Linux on z Systems

Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide

Version 5.0

SC33-8294-05

Linux on z Systems

Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide

Version 5.0

SC33-8294-05

Note
Before using this document, be sure to read the information in "Notices" on page 1033.

| This edition applies to the Common Cryptographic Architecture (CCA) API, Release 4.4 and 5.0 for Linux on z
| Systems, and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

About this document

The presented information describes how to use a variety of services for cryptography and data-security provided in the Common Cryptographic Architecture (CCA).

CCA functions described in this edition apply to the CCA host library (RPM) version 5.0, shortly referred to as CCA (Release) 5.0. The new host library 5.0 is tested on CCA firmware versions 4.4 and 5.0 and it works with any firmware version 4.x. Note that there are no CCA firmware or host library versions 4.5.

The CCA functions perform cryptographic operations using the following adapters in coprocessor mode:

- IBM® 4767 Crypto Express5 feature (CEX5C)
- IBM 4765 Crypto Express4 feature (CEX4C)
- IBM 4765 Crypto Express3 feature (CEX3C)
- IBM 4764 Crypto Express2 feature (CEX2C)

See “Concurrent installations” on page 1001 for details.

This publication is for planning and programming purposes only.

The CCA host software provides an application programming interface through which applications request secure, high-speed cryptographic services from the hardware cryptographic features.

This publication continues to document CCA Releases 4.0 up to 4.2.10. Where CCA Releases 4.4 and 5.0 have been changed or enhanced from the previously documented CCA Release 4.2.10, these changes are indicated with revision marks.

For the supported environments and product ordering information, see:
`http://www.ibm.com/security/cryptocards/`

Revision history

Track the changes of this document for each CCA Support Program release.

Edition July 2015, CCA Support Program Releases 4.4 and 5.0

This edition describes the IBM CCA Basic Services API for Release 5.0, including those for Release 4.4.

For Releases 4.4 and 5.0, changes to the CCA API include the following new verbs:

Table 1. New verbs for CCA Releases 4.4 and 5.0.

Verb entry-point name	Service name	Category
CSUALGQ	“Log Query (CSUALGQ)” on page 117	Using the CCA nodes and resource control verbs
CSNBDKG2	“Diversified Key Generate2 (CSNBDKG2)” on page 152	Managing AES and DES cryptographic keys

Table 1. New verbs for CCA Releases 4.4 and 5.0 (continued).

Verb entry-point name	Service name	Category
CSNDSXD	"Symmetric Key Export with Data (CSNDSXD)" on page 303	Managing AES and DES cryptographic keys
CSNBUKD	"Unique Key Derive (CSNBUKD)" on page 322	Managing AES and DES cryptographic keys
CSNBCTT2	"Cipher Text Translate2 (CSNBCTT2)" on page 356	Protecting data
CSNBMGN2	"MAC Generate2 (CSNBMGN2)" on page 382	Verifying data integrity and authenticating messages
CSNBMVR2	"MAC Verify2 (CSNBMVR2)" on page 390	Verifying data integrity and authenticating messages
CSNBAPG	"Authentication Parameter Generate (CSNBAPG)" on page 458	Financial services
CSNBPTRE	"Encrypted PIN Translate Enhanced (CSNBPTRE)" on page 495	Financial services
CSNBFPED	"FPE Decipher (CSNBFPED)" on page 509	Financial services
CSNBFPEE	"FPE Encipher (CSNBFPEE)" on page 516	Financial services
CSNBFPET	"FPE Translate (CSNBFPET)" on page 524	Financial services
CSNBPFO	"Recover PIN from Offset (CSNBPFO)" on page 540	Financial services
CSNBDDPG	"DK Deterministic PIN Generate (CSNBDDPG)" on page 558	Financial services for DK PIN methods[1)]
CSNBDMP	"DK Migrate PIN (CSNBDMP)" on page 565	Financial services for DK PIN methods
CSNBDPMT	"DK PAN Modify in Transaction (CSNBDPMT)" on page 571	Financial services for DK PIN methods
CSNBDPT	"DK PAN Translate (CSNBDPT)" on page 578	Financial services for DK PIN methods
CSNBDPC	"DK PIN Change (CSNBDPC)" on page 585	Financial services for DK PIN methods
CSNBDPV	"DK PIN Verify (CSNBDPV)" on page 597	Financial services for DK PIN methods
CSNBDPNU	"DK PRW Card Number Update (CSNBDPNU)" on page 601	Financial services for DK PIN methods
CSNBDPCG	"DK PRW CMAC Generate (CSNBDPCG)" on page 608	Financial services for DK PIN methods
CSNBDRPG	"DK Random PIN Generate (CSNBDRPG)" on page 611	Financial services for DK PIN methods
CSNBDRP	"DK Regenerate PRW (CSNBDRP)" on page 617	Financial services for DK PIN methods
CSNBXEA	"Code Conversion (CSNBXEA)" on page 743	Utility verbs[1)]
1) new verb category		

The following verbs provide new keywords:

Table 2. Updated verbs for CCA Releases 4.4 and 5.0.

Entry-point	Service name	Category
CSUACFQ	"Cryptographic Facility Query (CSUACFQ)" on page 77	Using the CCA nodes and resource control verbs

Table 2. Updated verbs for CCA Releases 4.4 and 5.0 (continued).

Entry-point	Service name	Category
CSNBCVG	"Control Vector Generate (CSNBCVG)" on page 134	Managing AES and DES cryptographic keys
CSNBDKG	"Diversified Key Generate (CSNBDKG)" on page 145	Managing AES and DES cryptographic keys
CSNBKEX	"Key Export (CSNBKEX)" on page 169	Managing AES and DES cryptographic keys
CSNBKGN	"Key Generate (CSNBKGN)" on page 171	Managing AES and DES cryptographic keys
CSNBKGN2	"Key Generate2 (CSNBKGN2)" on page 181	Managing AES and DES cryptographic keys
CSNBKIM	"Key Import (CSNBKIM)" on page 189	Managing AES and DES cryptographic keys
CSNBKYT2	"Key Test2 (CSNBKYT2)" on page 204	Managing AES and DES cryptographic keys
CSNBKTB	"Key Token Build (CSNBKTB)" on page 213	Managing AES and DES cryptographic keys
CSNBKTB2	"Key Token Build2 (CSNBKTB2)" on page 218	Managing AES and DES cryptographic keys
CSNBKTC	"Key Token Change (CSNBKTC)" on page 254	Managing AES and DES cryptographic keys
CSNBKTC2	"Key Token Change2 (CSNBKTC2)" on page 258	Managing AES and DES cryptographic keys
CSNBKTP	"Key Token Parse (CSNBKTP)" on page 260	Managing AES and DES cryptographic keys
CSNBKTP2	"Key Token Parse2 (CSNBKTP2)" on page 264	Managing AES and DES cryptographic keys
CSNBKTR2	"Key Translate2 (CSNBKTR2)" on page 276	Managing AES and DES cryptographic keys
CSNBRKA	"Restrict Key Attribute (CSNBRKA)" on page 290	Managing AES and DES cryptographic keys
CSNDSYX	"Symmetric Key Export (CSNDSYX)" on page 298	Managing AES and DES cryptographic keys
CSNDSYI2	"Symmetric Key Import2 (CSNDSYI2)" on page 316	Managing AES and DES cryptographic keys
CSNBPCU	"PIN Change/Unblock (CSNBPCU)" on page 532	Financial services
CSNBTRV	"Transaction Validation (CSNBTRV)" on page 552	Financial services
CSNDPKI	"PKA Key Import (CSNDPKI)" on page 640	Managing PKA cryptographic keys
CSNDKTC	"PKA Key Token Change (CSNDKTC)" on page 652	Managing PKA cryptographic keys
CSNDPKT	"PKA Key Translate (CSNDPKT)" on page 655	Managing PKA cryptographic keys
CSNDRKX	"Remote Key Export (CSNDRKX)" on page 664	Managing PKA cryptographic keys

Beginning with Release 4.4, a new AES key type SECMSG is supported, which is added to variable-length symmetric key tokens. See Table 242 on page 869.

Applications linked with prior CCA host software will continue to function if the CCA host software is upgraded in place. However, IBM always recommends full testing of upgrades before implementing production roll-out.

Edition January 2014, CCA Support Program Release 4.2.10

This edition describes the IBM CCA Basic Services API for Release 4.2.10.

For Release 4.2.10, changes to the CCA API include the following added support:

- DEV-ANY (AUTOSELECT) support added via CSUACRA, CSUACRD, environment variable CSU_DEFAULT_ADAPTER, for automatic (simple) load balancing.
- CCA key storage support for DES key tokens with Zero CVs.
- CEX4C support and recognition for all verbs and utilities.

Edition February 2012, CCA Support Program Release 4.2

This edition describes the IBM CCA Basic Services API for Release 4.2.

Important:

Two problems have been discovered with the CCA microcode related to the reenciphering of master keys. Although similar, the two problems are slightly different and exist in different levels of the microcode. These problems could lead to a loss of operational private keys after a master key change. Symmetric keys are not affected. Although it is expected few customers will be impacted this document describes the problems and how to recover.

For details, see http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/FLASH10764.

The PKA Key Token Change (CSNDKTC) verb has been changed to not permit the use of the **RTNMK** keyword for processor firmware levels that have this problem.

For Release 4.2, changes to the CCA API include:

- New verbs:
  - CVV Key Combine (CSNBCKC)
  - EC Diffie-Hellman (CSNDEDH)
  - Key Export to TR31 (CSNBT31X)
  - Key Token Parse2 (CSNBKTP2)
  - TR31 Key Import (CSNBT31I)
  - TR31 Key Token Parse (CSNBT31P)
  - TR31 Optional Data Build (CSNBT31O)
  - TR31 Optional Data Read (CSNBT31R)
- Added support to improve security and control for PIN decimalization tables, and changes to the required access control points for these verbs:
  - Clear PIN Generate (CSNBPGN)
  - Clear PIN Generate Alternate (CSNBCPA)
  - Encrypted PIN Generate (CSNBEPG)
  - Encrypted PIN Verify (CSNBPVR)
- Added support for double-length keys to these verbs:
  - CVV Generate (CSNBCSG)
  - CVV Verify (CSNBCSV)
- Added support for variable-length AES keys to these verbs:
  - Symmetric Algorithm Decipher (CSNBSAD)
  - Symmetric Algorithm Encipher (CSNBSAE)
- Added rule-array keywords **SHA-1** and **SHA-256** to these verbs:
  - Symmetric Key Generate (CSNDSYG)
  - Symmetric Key Import (CSNDSYI)
- Added key usage **NOT31XPT** and **T31XPTOK** to this verb:
  - Control Vector Generate (CSNBCVG)
- Updated control vector key combinations for fixed-length DES key tokens, for these verbs:
  - Control Vector Generate (CSNBCVG)
  - Key Token Build (CSNBKTB)
- Added new rule-array keyword **AES** to these verbs:

  - Key Part Import2 (CSNBKPI2)
  - Key Token Change2 (CSNBKTC2)
  - Symmetric Key Import2 (CSNDSYI2)
- Added new rule-array keyword **AESKW** to these verbs:
  - Symmetric Key Export (CSNDSYX)
  - Symmetric Key Import2 (CSNDSYI2)
- Added support for AES keys, and many other changes to this verb:
  - Key Generate2 (CSNBKGN2)
- Added new rule-array keywords **CLONE**, **OKEK-AES**, and **OKEK-DES** to this verb:
  - PKA Key Generate (CSNDPKG)
- Added new keywords **STATCRD2**, **NUM-DECT**, **STATDECT**, **STATVKPL**, and **STATVKPR** to this verb:
  - Cryptographic Facility Query (CSUACFQ)
- Put an important warning in this verb:
  - PKA Key Token Change (CSNDKTC)
- Added rule-array keywords **AES**, **DES**, **ENC-ZERO**, **SHA-256**, **TR-31**, **IKEK-PKA**, **IKEK-DES**, and **IKEK-AES** to this verb:
  - Key Test2 (CSNBKYT2)
- Added rule-array keywords **AES**, **DES**, **NOEX-AES**, **NOEX-DES**, **NOEX-RAW**, **NOEX-RSA**, **NOT31XPT**, **IKEK-AES**, **IKEK-DES**, and **IKEK-PKA** to this verb:
  - Restrict Key Attribute (CSNBRKA)

Edition March 2011, CCA Support Program Release 4.1.0

This edition describes the IBM CCA Basic Services API for Release 4.1.0.

For Release 4.1.0, changes to the CCA API include:

- Enhanced PIN security with the addition of ANSI X9.8 restriction capabilities

  Three new access control points are added to enhance PIN security by blocking PIN attacks. See the **Required commands** sections of the Clear PIN Generate Alternate (CSNBCPA), Encrypted PIN Translate (CSNBPTR), and Secure Messaging for PINs (CSNBSPN) verbs.
- Wrap CCA keys in Cipher-Block Chaining (CBC) mode

  A second key-wrapping method is added for DES that is a more secure version of Triple-DES ECB mode currently used by CCA. The enhanced version of key wrapping complies with current cryptographic standards that require key bundling. This new key-wrapping method can coexist with the CCA legacy ECB mode of wrapping Triple-DES keys. The two methods can coexist on the same or multiple systems.
- Elliptic Curve Cryptography (ECC) support

  New Elliptic Curve Cryptography (ECC) key generation, along with support for digital signature generation and verification using the Elliptic Curve Digital Signature Algorithm (ECDSA). This enhancement includes a new PKA key-token for housing ECC public-key cryptographic keys and a new asymmetric APKA master-key (32-byte AES key) for wrapping an ECC key-token, along with added support to the Master Key Process verb.
- Hashed Message Authentication Code (HMAC) support for key generation and processing, but not for key storage.
- These new verbs:

- HMAC Generate (CSNBHMG)
- HMAC Verify (CSNBHMV)
- Key Generate2 (CSNBKGN2)
- Key Part Import2 (CSNBKPI2)
- Key Test2 (CSNBKYT2)
- Key Token Build2 (CSNBKTB2)
- Key Token Change2 (CSNBKTC2)
- Key Translate2 (CSNBKTR2)
- Restrict Key Attribute (CSNBRKA)
- Symmetric Key Import2 (CSNDSYI2)

Edition April 2010, CCA Support Program Release 4.0.0

This edition describes the IBM CCA Basic Services API for Release 4.0.0.

For Release 4.0.0, changes to the CCA API include:

- Support for the IBM Crypto Express3 feature (CEX3C) in coprocessor mode
- A Java™ Native Interface (JNI) form for most of the verbs
- Central Processor Assist for Cryptographic Functions (CPACF) support
- These new verbs:
  - AES Key Record Create (CSNBAKRC)
  - AES Key Record Delete (CSNBAKRD)
  - AES Key Record List (CSNBAKRL)
  - AES Key Record Read (CSNBAKRR)
  - AES Key Record Write (CSNBAKRW)
  - Control Vector Translate (CSNBCVT)
  - Cryptographic Facility Version (CSUACFV)
  - Cryptographic Variable Encipher (CSNBCVE)
  - Key Test Extended (CSNBKYTX)
  - MDC Generate (CSNBMDG)
  - PKA Key Translate (CSNDPKT)
  - Prohibit Export Extended (CSNBPEXX)
  - Random Number Generate Long (CSNBRNGL)
  - Remote Key Export (CSNDRKX)
  - Retained Key Delete (CSNDRKD)
  - Retained Key List (CSNDRKL)
  - Symmetric Algorithm Decipher (CSNBSAD)
  - Symmetric Algorithm Encipher (CSNBSAE)
  - Trusted Block Create (CSNDTBC)

Who should use this document

This document is intended for application programmers who are responsible for writing application programs that use the security application programming interface (API) to access cryptographic functions.

Distribution-specific information

Common Cryptographic Architecture for Linux on z Systems™, Release 5.0, adds support for the CEX5S feature. In order to use the full set of CCA 5.0, a Linux on z Systems distribution with support for the CEX5S feature is required.

- The following previous distributions provide CEX5S toleration:
  - SLES – s390x, 64-bit only
    - SLES 12, SP0
    - SLES 11, SP4
  - RHEL – s390x, 64-bit only
    - RHEL 5.11
    - RHEL 6.6
    - RHEL 7.1
- CCA Linux on z Systems maintenance Release 4.2.10 adds support for CEX4C. These distributions introduced CEX4C exploitation:
  - SUSE Linux Enterprise Server 11 SP3 (SLES 11 SP3) 64-bit only
  - Red Hat Enterprise Linux 6 Update 4 (64-bit only)
- SUSE Linux Enterprise Server 11 SP2 (SLES 11 SP2) 64-bit only
- SUSE Linux Enterprise Server 10 SP4 (SLES 10 SP4) 64-bit only
- Red Hat Enterprise Linux 5 Update 7 (64-bit only)
- Red Hat Enterprise Linux 6 Update 2 (64-bit only)

The following Linux on z Systems distributions support CCA Release 4.1.0 host software for use with CEXC3. Support also extends to CEX2C within the functional scope of the CEX2C feature:

- SUSE Linux Enterprise Server 11 SP1 (SLES 11 SP1)
- SUSE Linux Enterprise Server 10 SP3 (SLES 10 SP3)
- Red Hat Enterprise Linux 5 Update 6
- Red Hat Enterprise Linux 6

For restrictions and recommendations about the usage of CCA prior to release 5.0 with Linux on z Systems distributions, refer to http://www.ibm.com/security/cryptocards/pciecc/ordersoftware.shtml. For the appropriate information for CCA 5.0, refer to http://www.ibm.com/security/cryptocards/pciecc2/overview.shtml.

Note:

1. CCA 4.2 host software supports all prior levels of CEX3C adapter firmware and may also be used for support of the CEX2C subset of functionality. Full CEX2C support is included in CCA Release 4.2.0 and 4.1.0. However, note that because of limits in the CEX2C hardware and firmware available, this is a limited subset of the CEX*C functions described in this document.
2. Only 64-bit versions of this software are provided. 31-bit support is not provided.

Restrictions

Known restrictions for current and prior releases to assist system administrators.

General note

The CCA host library (libcsulccas.so) was changed to allow more flexible preparation of requests to be sent to the adapter. The change allows very

large key block support, among other changes. The z90crypt device driver as it exists in currently available Linux distributions has the same limitation as older CCA host library code, and so a patch was prepared and submitted for maintenance releases to 'in-service' Linux distributions.

IBM is working with our distribution partners to include a patch to remove these noted restrictions in future distributions.

General description of the restriction

Verbs that may send or return a lot of data (of certain types, such as lists of key labels or key tokens) or large key tokens are limited by an issue in the current version of the z90crypt device driver buffer handling to a smaller amount of data or key token size than would normally be allowed.

The following scenarios clarify what to avoid to prevent this restriction from leading to errors.

Restriction scenario: Sending or requesting a large amount of certain types of data

CSNDRKL

This verb returns a list of labels or tokens for a specified set of retained keys. Specifying a large number for the key_labels_count or retained_keys_count parameter can result in more return data than the cryptographic device driver can handle. Because a key label is 64 bytes, do not specify key_labels_count values greater than 75.

Crossing this limit results in return code 8 reason code 1106 error, indicating the data is too large to be returned (it would be truncated).

CSNDRKX

This verb has the potential to send large objects for parameters certificate, **certificate_parms** and **extra_data**. Avoid using a combined value for these parameters that greatly exceeds 4096 Bytes. The actual value of the threshold varies with the size of other parameters and so cannot be specified exactly.

Crossing this limit results in error return code 8 reason code 343.

Restriction scenario: Processing extremely large key tokens

CSNBT31I, CSNBT31X, CSNBKYT2

These verbs handle TR-31 key blocks, which can be up to 9992 Bytes (if 9 KB or more of optional block sections have been added). Due to the z90crypt restriction, TR-31 key blocks should be built specifying no more than 4096 Bytes of optional block sections.

Crossing this limit results in error return code 8 reason code 343.

Terminology

The following terms are used in this document for CCA releases and features.

CEX2C

An IBM 4764 Crypto Express2 feature, configured in CCA coprocessor mode.

CEX3C

An IBM 4765 Crypto Express3 feature, configured in CCA coprocessor mode.

CEX4C
An IBM 4765 Crypto Express4 feature, configured in CCA coprocessor mode.

CEX5C
An IBM 4767 Crypto Express5 feature, configured in CCA coprocessor mode.

CEX*C
Either the CEX2C, CEX3C, CEX4C, CEX5C, or (if plural) any combination of these.

Hardware requirements

In order to make use of the verbs provided in the Common Cryptographic Architecture (CCA) API for Linux on z Systems, your hardware must meet certain minimum requirements.

This is the minimum supported hardware configuration:

- One CEXC3 feature, with one CEXC3 adapter mapped to the z/VM® image or LPAR that uses it. The CEXC3 must have CCA 4.2.0z or greater firmware loaded, configured in co-processor mode.
- If you plan to use a Trusted Key Entry (TKE) workstation, you must have a TKE V6.0 or higher workstation in order to see supported CEXC3s. They are not seen when using TKE V5 or earlier workstations. Note that a TKE V8.0 workstation is required to manage CEX5C adapters.

This is the maximum supported hardware configuration:

- IBM z13™
  - 16 CEX5S total in the machine
  - all 16 may also be mapped to an LPAR, up to 85 total LPARs
  - there is 1 adapter per feature code

This hardware configuration is also supported:

- IBM zEnterprise® EC12
  - One or two CEC3C or CEX4S adapter features, with up to 4 adapters mapped to a single z/VM image or LPAR.
- IBM zEnterprise 196
  - 2 CEX3C adapter features, with 4 CEX3C adapters mapped to a single z/VM image or LPAR

See “Concurrent installations” on page 1001 for details about a mixed environment of CEX2C and CEX*C. Note that CEX4Cs are only supported by TKE 7.2 or later if the Linux driver reports them as CEX4s. If you are running in toleration mode, and the Linux driver reports them as CEX3Cs, then TKE 6.0 is be able to manage them as CEX3Cs.

To determine whether a Crypto Express adapter available to Linux on z Systems is a CEX2C, a CEX3C, CEX4C, or a CEX5C, see “Listing CCA coprocessors” on page 11.

VM64656
Introduces Crypto Express3 support.

VM64727
Fixes problem with shared coprocessors.

VM64793
Introduces protected key CPACF support.

VM65007
Introduces Crypto Express4 support.

On z/VM 6.2, apply the following APAR fixes:

VM65007
Introduces Crypto Express4 support.

VM65577
Introduces Crypto Express5 support.

On z/VM 6.3, apply the following APAR fix:

VM65577
Introduces Crypto Express5 support.

How to use this document

Read an overview of the information in each section of this document.

For encryption, CCA supports Advanced Encryption Standard (AES), Data Encryption Standard (DES), public key cryptography (PKA or RSA), and Elliptic Curve Cryptography (ECC). These are very different cryptographic systems. Additionally, CCA provides APIs for generating and verifying Message Authentication Codes (MACs), Hashed Message Authentication Codes (HMACs), hashes, and PINS, as well as other cryptographic functions.

Part 1, "IBM CCA programming," on page 1 focuses on IBM CCA programming. It includes the following chapters:

- Chapter 1, "Introduction to programming for the IBM Common Cryptographic Architecture," on page 3 describes the programming considerations for using the CCA verbs. It also explains the syntax and parameter definitions used in verbs. Information about concurrency is also provided.
- Chapter 2, "Using AES, DES, and HMAC cryptography and verbs," on page 29 gives an overview of AES, DES, and HMAC cryptography, and provides general guidance information on how these verbs use different key types and key forms.
- Chapter 3, "Introducing PKA cryptography and using PKA verbs," on page 63 introduces Public Key Algorithm (PKA) support and describes programming considerations for using the CCA PKA and ECC verbs, such as the PKA key token structure and key management.
- Chapter 4, "TR-31 symmetric-key management," on page 71 introduces TR-31 support and how CCA uses an IBM-defined optional block in a TR-31 key block.

Part 2, "CCA verbs," on page 75 focuses on CCA verbs and includes the following topics:

- Chapter 5, "Using the CCA nodes and resource control verbs," on page 77 describes using the CCA resource control verbs.
- Chapter 6, "Managing AES and DES cryptographic keys," on page 129 describes the verbs for generating and maintaining DES and AES cryptographic keys, the Random Number Generate verb (which generates 8-byte random numbers), the Random Number Generate Long verb (which generates up to 8192 bytes of

random content), and the Secure Sockets Layer (SSL) security protocol. This chapter also describes utilities to build DES and AES tokens, generate and translate control vectors, and describes the PKA verbs that support DES and AES key distribution.

- Chapter 7, "Protecting data," on page 333 describes the verbs for enciphering and deciphering data.
- Chapter 8, "Verifying data integrity and authenticating messages," on page 369 describes the verbs for generating and verifying Message Authentication Codes (MACs), generating and verifying Hashed Message Authentication Codes (HMACs), generating Modification Detection Codes (MDCs), and generating hashes (SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, MD5, RIPEMD-160).
- Chapter 9, "Key storage mechanisms," on page 401 describes the use of key storage, key tokens, and associated verbs.
- Chapter 10, "Financial services," on page 443 describes the verbs for use in support of finance-industry applications. This includes several categories.
  - Verbs for generating, verifying, and translating personal identification numbers (PINS).
  - Verbs that generate and verify VISA card verification values and American Express card security codes.
  - Verbs to support smart card applications using the EMV (Europay MasterCard Visa) standards.
- Chapter 11, "Financial services for DK PIN methods," on page 557 describes the verbs for PIN methods and requirements for financial services specified by the German Banking Industry Committee, *Deutsche Kreditwirtschaft* (DK).
- Chapter 12, "Using digital signatures," on page 625 describes the verbs that support using digital signatures to authenticate messages.
- Chapter 13, "Managing PKA cryptographic keys," on page 635 describes the verbs that generate and manage PKA keys.
- Chapter 14, "TR-31 symmetric key management verbs," on page 681 describes the verbs used to manage TR-31 key blocks and TR-31 functions.
- Chapter 15, "Utility verbs," on page 743 describes the utility verb CSNBXEA which is provided for code conversion.

Part 3, "Reference information," on page 749 includes the following information:

- Chapter 16, "Return codes and reason codes," on page 751 explains the return and reason codes returned by the verbs.
- Chapter 17, "Key token formats," on page 769 describes the formats for AES, DES internal, external, and null key tokens, for PKA public, private external, and private internal key tokens containing Rivest-Shamir-Adleman (RSA) information, PKA null key tokens, ECC key tokens, HMAC key tokens, Transaction Validation Values (TVVs), and trusted blocks.
- Chapter 18, "Key forms and types used in the Key Generate verb," on page 893 describes the key forms and types used by the Key Generate verb.
- Chapter 19, "Control vectors and changing control vectors with the Control Vector Translate verb," on page 897 contains a table of the default control vector values that are associated with each key type and describes the control information for testing control vectors, mask array preparation, selecting the key-half processing mode, and an example of using the Control Vector Translate verb.
- Chapter 20, "PIN formats and algorithms," on page 913 describes the PIN notation, formats, extraction rules, and algorithms.

- Chapter 21, “Cryptographic algorithms and processes,” on page 927 describes various ciphering and key verification algorithms, as well as the formatting of hashes and keys.
- Chapter 22, “Access control points and verbs,” on page 953 lists the access control points and their corresponding verbs.
- Chapter 23, “Sample verb call routines,” on page 977 contains sample verb call routines, both in C and Java, that illustrates the practical application of CCA verb calls.
- Chapter 24, “Initial system set-up tips,” on page 987 includes tips to help you set up your system for the first time.
- Chapter 25, “CCA installation instructions,” on page 991 includes RPM installation, configuration, and uninstallation instructions.
- Chapter 26, “Coexistence of CEX5C and previous CEX*C features,” on page 1001 includes information about using CEX2C and CEX*C features in the same system, and other restrictions.
- Chapter 27, “Utilities,” on page 1003 describes the ivp.e and panel.exe utilities.
- Chapter 28, “Security API command and sub-command codes,” on page 1009 contains an alphabetical list of security API command and sub-command codes returned by the output **rule-array** for option STATDIAG of the Cryptographic Facility Query verb.
- Chapter 29, “openCryptoki support,” on page 1013 provides information about openCryptoki which is an open source implementation of the *Cryptoki* API as defined by the industry-wide PKCS #11 Cryptographic Token Interface Standard.
- Chapter 30, “List of abbreviations,” on page 1023 contains definitions of abbreviations used.

Where to find more information

You can find other CCA product publications that might be of use with applications and systems that you might develop for use with the IBM 4765 and IBM 4764.

While there is substantial commonality in the API supported by the CCA products, and while this document seeks to guide you to the subset supported by Linux on z Systems, other individual product publications might provide further insight into potential issues of compatibility.

IBM 4765 PCIe and IBM 4764 PCI-X Cryptographic Coprocessors
All of the IBM 4765-related and 4764-related publications can be obtained from the Library page that you can reach from the product Web site at http://www.ibm.com/security/cryptocards/:

IBM 4765 PCIe Cryptographic Coprocessor CCA Support Program Installation Manual **and the** *IBM 4764 PCI-X Cryptographic Coprocessor CCA Support Program Installation Manual*
Describe the installation of the CCA Support Program and the operation of the Cryptographic Node Management utility.

IBM 4765 PCIe Cryptographic Coprocessor Installation Manual **and the** *IBM 4764 PCI-X Cryptographic Coprocessor Installation Manual*
Describe the physical installation of the IBM 4765 and the IBM 4764, and also the battery-changing procedure.

Custom Programming for the IBM 4765 and the IBM 4764
The Library portion of the product Web site also includes programming information for creating applications that perform within the IBM 4765 and

the IBM 4764. See the reference to custom programming. The product Web site is located at http://www.ibm.com/security/cryptocards/.

Other documents referenced in this book are:

- *IBM CCA Basic Services Reference and Guide for the IBM 4765 PCIe and IBM 4764 PCI-X Cryptographic Coprocessors*
- *IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface Reference*, SC40-1675
- *z/OS Cryptographic Services ICSF: Trusted Key Entry PCIX Workstation User's Guide*, SA23-2211-05
- For Linux on z Systems:
  - *Device Drivers, Features, and Commands*, SC33-8411

    See one of these web sites for the version of this document that is correct for your distribution of Linux:
    - http://www.ibm.com/developerworks/linux/linux390/documentation_suse.html
    - http://www.ibm.com/developerworks/linux/linux390/documentation_red_hat.html

Cryptography publications

The following publications describe cryptographic standards, research, and practices relevant to the coprocessor.

- Accredited Standards Committee X9, Inc.: X9 TR-31 2010: *Interoperable Secure Key Exchange Block Specification for Symmetric Algorithms*
- American Express Travel Related Services Company, Inc.: *American Express Hardware Security Module (HSM): Function Requirements*, August, 2011.
- American National Standards Institute (ANSI). ANSI is the official U.S. representative to the International Organization for Standardization (ISO) and, via the U.S. National Committee, the International Electrotechnical Commission (IEC). ANSI is also a member of the International Accreditation Forum (IAF).
  - ANSI X9.8-1:2003 *Banking -- Personal Identification Number Management and Security -- Part 1: PIN Protection Principles and Techniques for Online PIN Verification in ATM & POS Systems.*
  - ANSI X9.9:1986 *Financial Institution Message Authentication (Wholesale).*
  - ANSI X9.19:1998 *Financial Institution Retail Message Authentication.*
  - ANSI X9.23:1998 *Financial Institution Encryption of Wholesale Financial Messages.*
  - ANSI X9.24-1:2009 *Retail Financial Services Symmetric Key Management -- Part 1: Using Symmetric Techniques.*
  - ANSI X9.24-2:2006 *Retail Financial Services Symmetric Key Management -- Part 2: Using Asymmetric Techniques for the Distribution of Symmetric Keys.*
  - ANSI X9.31:1998 *Digital Signature Using Reversible Public Key Cryptography for the Financial Services Industry.*
  - ANSI X9.52:1998 *Triple Data Encryption Algorithm Modes of Operation.*
  - ANSI X9.62:2005 *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA).*
  - ANSI X9.63:2001 *Public Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography.*
  - ANSI X9.102:2008 *Symmetric Key Cryptography for the Financial Services Industry -- Wrapping of Keys and Associated Data*.

- *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition*, Bruce Schneier, John Wiley & Sons, Inc., 1996, ISBN 0-471-12845-7 or ISBN 0-471-11709-9
- *ECC Brainpool Standard Curves and Curve Generation*, v.1.0, October 19, 2005

  Available at http://www.ecc-brainpool.org/download/Domain-parameters.pdf.
- *Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation* (RFC 5639), Manfred Lochter and Johannes Merkle, IETF Trust, March 2010. Available at http://www.rfc-editor.org/rfc/rfc5639.txt
- *EMV 2000 Integrated Circuit Card Specifications for Payment Systems, Book 2 -- Security and Key Management*, Version 4.0, EMVCo, LLC, December 2000.
- Federal Information Processing Standards (FIPS), issued by the U.S. National Institute of Standards and Technology (NIST, see http://www.nist.gov/itl/). The listed FIPS publications are available from this web site: FIPS PUBLICATIONS.
  - FIPS PUB 140-2 *Security Requirements for Cryptographic Modules*, May, 2001.
  - FIPS PUB 180-4 *Secure Hash Standard (SHS)*, May, 2012.
  - FIPS PUB 186-4 *Digital Signature Standard (DSS)*, July, 2013.
  - FIPS PUB 197 *Advanced Encryption Standard (AES)*, November, 2001.
  - FIPS PUB 198-1 *The Keyed-Hash Message Authentication Code (HMAC)*, July, 2008.
- International Organization for Standardization (ISO). ISO is the world's largest developer and publisher of International Standards. ISO is a network of the national standards institutes of many countries, one member per country, with a Central Secretariat in Geneva, Switzerland, that coordinates the system.
  - ISO 16609:2004 *Banking -- Requirements for Message Authentication Using Symmetric Techniques.*
  - ISO/DIS 9564-1 *Financial Services -- Personal Identification Number (PIN) Management and Security -- Part 1: Basic Principles and Requirements for PINs in Card-Based Systems.*
- International Organization for Standardization/International Electrotechnical Commission (ISO/IEC)
  - ISO/IEC 9796-1:1997 *Information Technology -- Security Techniques -- Digital Signature Schemes Giving Message Recovery -- Part 1: Mechanisms Using Redundancy.*
  - ISO/IEC 9796-2:2002 *Information Technology -- Security Techniques -- Digital Signature Schemes Giving Message Recovery -- Part 2: Integer Factorization Based Mechanisms.*
  - ISO/IEC 9797-1:1999 *Information technology -- Security techniques -- Message Authentication Codes (MACs) -- Part 1: Mechanisms Using a Block Cipher.*
  - ISO/IEC 11770-3:2008 *Information Technology -- Security Techniques -- Key Management -- Part 3: Mechanisms Using Asymmetric Techniques.*
- National Institute of Standards and Technology (NIST) Special Publications (SP), U.S. Dept. of Commerce
  - NIST SP 800-38A *Recommendation for Block Cipher Modes of Operations: Methods and Techniques*, 2001 Edition. Available at http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.
  - NIST SP 800-38B *Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication*, May 2005. Available at: http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf.

- NIST SP 800-56A *Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised)*, May 2013. Available at http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf.
- NIST SP 800-90A *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*, January 2012. Available at: http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf.
- NIST SP 800-108 *Recommendation for Key Derivation Using Pseudorandom Functions (Revised)*, October 2009 Edition. Available at: http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf.
- *The NIST SP 800-90A Deterministic Random Bit Generator Validation System (DRBGVS)*, February 2013. Available at: http://csrc.nist.gov/groups/STM/cavp/documents/drbg/DRBGVS.pdf.

- RSA Laboratories, Standard Initiatives, for information on the Public-Key Cryptographic Standards (PKCS): http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/index.htm, especially **PKCS #11**: PKCS #11: Cryptographic Token Interface Standard
- *SET Secure Electronic Transaction, Version 1.0*, May 31, 1997, Secure Electronic Transaction LLC.
- *Standards for Efficient Cryptography 2 (SEC 2): Recommended Elliptic Curve Domain Parameters* Version 2.0, Standards for Efficient Cryptography Group (SECG), Certicom Research, January 27, 2010. Available at http://www.secg.org/download/aid-784/sec2-v2.pdf
- *The MD5 Message-Digest Algorithm*, Internet Engineering Task Force Request for Comments (RFC) 1321, MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992. Available at http://www.ietf.org/rfc/rfc1321.txt.
- *Visa Integrated Circuit Card Specification*, Version 1.4.0, Visa International, October 31, 2001.
- *Visa Integrated Circuit Card Specification (VIS) 1.4.0 Corrections and Updates*, Visa International.

Do you have problems, comments, or suggestions?

Your suggestions and ideas can contribute to the quality and the usability of this document.

If you have problems using this document, or if you have suggestions for improving it, complete and mail the Reader's Comment Form found at the back of the document.

Part 1. IBM CCA programming

Read the information in this part of the document which introduces programming for the IBM CCA, AES, DES, and PKA cryptography.

The topics in this part explain how to use CCA nodes and AES, DES, and PKA verbs.

- Chapter 1, "Introduction to programming for the IBM Common Cryptographic Architecture," on page 3 describes the programming considerations for using the CCA verbs. It also explains the syntax and parameter definitions used in the verbs. Information about concurrency is also provided.
- Chapter 2, "Using AES, DES, and HMAC cryptography and verbs," on page 29 gives an overview of AES, DES, and ECC cryptography and provides general guidance information on how these verbs use different key types and key forms.
- Chapter 3, "Introducing PKA cryptography and using PKA verbs," on page 63 introduces Public Key Algorithm (PKA) and Elliptic Curve Cryptography (ECC) support, and describes programming considerations for using the CCA PKA verbs, such as the PKA key token structure and key management.
- Chapter 4, "TR-31 symmetric-key management," on page 71 introduces X9 TR-31 (Technical Report 31) support, and provides details about the TR-31 key block.

Chapter 1. Introduction to programming for the IBM Common Cryptographic Architecture

By using the IBM CCA application programming interface (API), you can obtain cryptographic and other services from the CEX*C feature and CCA.

The following subtopics are discussed:

Available Common Cryptographic Architecture (CCA) verbs

CCA products provide a variety of cryptographic processes and data-security techniques.

Your application program can call verbs (sometimes called services) to perform the following functions:

Data confidentiality
: Encrypt and decrypt information, typically using the AES or DES algorithms in Cipher Block Chaining (CBC) mode to enable data confidentiality.

Data integrity
: Hash data to obtain a digest, or process the data to obtain a Message Authentication Code (MAC) or keyed hash MAC (HMAC), that is useful in demonstrating data integrity.

Nonrepudiation
: Generate and verify digital signatures using either the RSA algorithm or the ECDSA algorithm, to demonstrate data integrity and form the basis for nonrepudiation.

Authentication
: Generate, encrypt, translate, and verify finance industry personal identification numbers (PINs) and American Express, MasterCard, and Visa card security codes with a comprehensive set of finance-industry-specific services.

Key management
: Manage the various AES, DES, ECC, and RSA keys necessary to perform the mentioned cryptographic operations.

Java interaction
: Interact with the Java Native Interface (JNI). Some of the CCA verbs have a specific version that can be used for JNI work.

CCA management
: Control the initialization and operation of CCA.

This publication groups the many available verbs by topics. Each topic lists the verbs in alphabetical order by verb pseudonym.

Common Cryptographic Architecture functional overview

You use the CCA security API to access a common cryptographic architecture.

Figure 1 provides a conceptual framework for positioning the CCA security API, which you use to access a common cryptographic architecture. Application programs make procedure calls to the CCA security API to obtain cryptographic and related I/O services. The CCA security API is designed so that a call can be issued from essentially any high-level programming language. The call, or request, is forwarded to the cryptographic services access layer and receives a synchronous response. Your application program loses control until the access layer returns a response after processing your request.

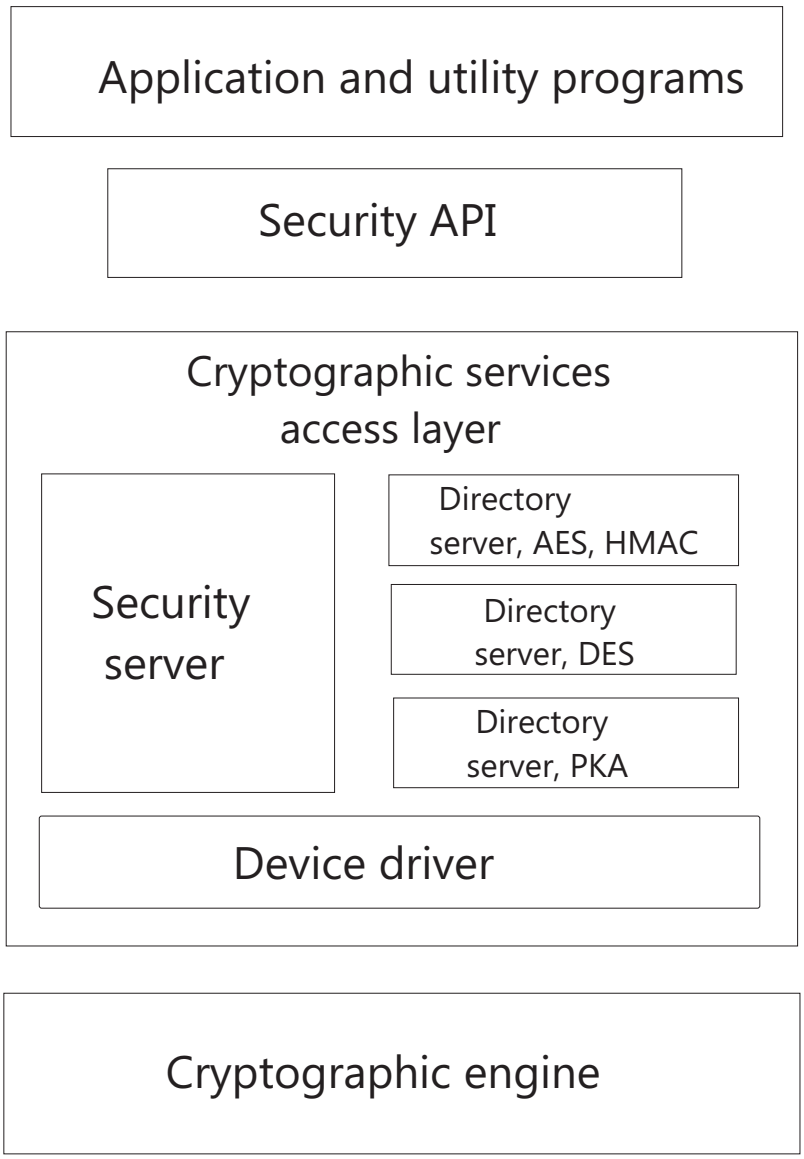


Figure 1. CCA security API, access layer, and cryptographic engine

The products that implement the CCA support consist of both hardware and software components.

CCA software support: The software consists of application development and runtime software components.

- The application development software primarily consists of language bindings that can be included in new applications to assist in accessing services available at the API. Language bindings are provided for the C and Java programming languages.
- The runtime software can be divided into the following categories:
  - Service-requesting programs, including application and utility programs.
  - The security API, an agent function that is logically part of the calling application program or utility.
  - The cryptographic services access layer: an environment-dependent request routing function, key-storage support services, and device driver to access one or more hardware cryptographic engines.
  - The cryptographic engine software that gives access to the cryptographic engine hardware.

    The cryptographic engine is implemented in the hardware of the CEX*C coprocessor. Security-sensitive portions of CCA are implemented in the cryptographic engine software running in the protected coprocessor environment.
  - Utility programs and tools provide support for administering CCA secret keys, interacting with CCA managed symmetric and public key cryptography key storage, and configuring the software support.

You can create application programs that employ the CCA security API or you can purchase applications from IBM or other sources that use the products. This document is the primary source of information for designing systems and application programs that use the CCA security API with the cryptographic coprocessors.

Cryptographic engine: The CCA architecture defines a cryptographic subsystem that contains a cryptographic engine operating within a protected boundary. The coprocessor's tamper-resistant, tamper-responding environment provides physical security for this boundary and the CCA architecture provides the logical security needed for the full protection of critical information.

CEX2C and CEX3C Coprocessors: The coprocessor provides a secure programming and hardware environment wherein AES, DES, RSA, Elliptic Curve, and HMAC processes are performed. Each cryptographic coprocessor includes a general-purpose processor, non-volatile storage, and specialized cryptographic electronics. These components are encapsulated in a protective environment to enhance security. The IBM CCA Support Program enables applications to employ a set of AES, DES, RSA, Elliptic Curve, and HMAC-based cryptographic services utilizing the coprocessor hardware. Services include:

- DES, AES, RSA, Elliptic Curve, and HMAC key-pair generation
- DES, AES, RSA, Elliptic Curve, and HMAC host-based key record management
- Digital signature generation and verification
- Cryptographic key wrapping and unwrapping
- Data encryption, decryption and MAC generation/verification
- PIN processing for the financial services industry
- Other services, including DES key-management based on control-vector-enforced key separation.

CEX4C Coprocessor: The coprocessor provides the same cryptographic functions as the CEX3C coprocessor.

CEX5C Coprocessor: The coprocessor provides the same functions as the CEX4C coprocessor with more algorithms moved from hardware-enhanced to fully hardware-accelerated. CMAC, VFPE, AESKW, and other algorithms are added.

CCA: Common Cryptographic Architecture (CCA) is the basis for a consistent cryptographic product family. Applications employ the CCA security API to obtain services from, and to manage the operation of, a cryptographic system that meets CCA specifications.

CCA access control: Each CCA node has an access-control system enforced by the hardware and protected software. The robust UNIX style access controls integrated into the Linux operating system are used to protect the integrity of the underlying CCA hardware environment. The specialized processing environment provided by the cryptographic engine can be kept secure because selected services are provided only when certain requirements are met or a Trusted Key-Entry console is used to enable access. The access-control decisions are performed within the secured environment of the cryptographic engine and cannot be subverted by rogue code that might run on the main computing platform.

Coprocessor certification: After quality checking a newly manufactured coprocessor, IBM loads and certifies the embedded software. Following the loading of basic, authenticated software, the coprocessor generates an RSA key-pair and retains the private key within the cryptographic engine. The associated public key is signed by a certification key securely held at the manufacturing facility and then the certified device key is stored within the coprocessor. The manufacturing facility key has itself been certified by a securely held key unique to the CEX*C product line.

The private key within the coprocessor, known as the device private key, is retained in the coprocessor. From this time on, if tampering is detected or if the coprocessor batteries are removed or lose power in the absence of bus power, the coprocessor sets all security-relevant keys and data items to zero. This process is irreversible and results in the permanent loss of the factory-certified device key, the device private key, and all other data stored in battery-protected memory. Security-sensitive data stored in the coprocessor flash memory is encrypted. The key used to encrypt such data is itself retained in the battery-protected memory.

CCA master key: When using the CCA architecture, working keys, including session keys and the RSA and ECC private keys used at a node to form digital signatures or to unwrap other keys, are generally stored outside the cryptographic-engine protected environment. These working keys are wrapped (DES triple-encrypted or AES encrypted) by the CCA master key. The master key is held in the clear (not enciphered) within the cryptographic engine.

The number of keys usable with a CCA subsystem is thus restricted only by the host server storage, not by the finite amount of storage within the coprocessor secure module. In addition, the working keys can be used by additional CCA cryptographic engines which have the same master key. This CCA characteristic is useful in high-availability and high-throughput environments where multiple cryptographic processors must function in parallel.

Establishing a CCA master key: To protect working keys, the master key must be generated and initialized in a secure manner. One method uses the internal

random-number generator for the source of the master key. In this case, the master key is never external to the node as an entity and no other node has the same master key unless master-key cloning is authorized and in use (unless, out of all the possible values, another node randomly generates the same master-key data). If an uncloned coprocessor loses its master key - for example, the coprocessor detects tampering and destroys the master key, - there is no way to recover the working keys that it wrapped. The number of possible values is:

- For DES and RSA master keys, 2^{168}
- For AES and APKA master keys, 2^{256}

Another master-key-establishment method enables authorized users to enter multiple, separate key parts into the cryptographic engine. As each part is entered, that part is XORed with the contents of the new master-key register. When all parts have been accumulated, a separate command is issued to promote the contents of the current master-key register to the old master-key register and to promote the contents of the new master-key register to the current master-key register. The length of the key parts is:

- For DES and RSA master keys, 168 bits
- For AES and APKA master keys, 256 bits

CCA verbs: An application or utility program obtains service from the CCA Support Program by issuing service requests (verb calls or procedure calls) to the runtime subsystem (see Chapter 23, “Sample verb call routines,” on page 977). To fulfill these requests, the Support Program in turn obtains service from the coprocessor software and hardware.

The available services are collectively described as the CCA security API. All the software and hardware accessed through theCCA security API should be considered an integrated subsystem. A command processor performs the verb request within the cryptographic engine.

Commands and access control, roles, profiles: In order to ensure that only designated individuals (or programs) can run commands such as master-key loading, each command processor that performs sensitive processing interrogates one or more *control-point* values within the cryptographic engine access-control system for permission to perform the request.

The access-control system includes one or more roles. Each role defines the permissible control points for users of that role. In the IBM z Systems environment, all application programs run using the permissions defined in the DEFAULT role for their domain. The DEFAULT role can only be modified using the TKE workstation. For a description of the functions that are permitted by the default version of the DEFAULT role, see *z/OS Cryptographic Services ICSF: Trusted Key Entry PCIX Workstation User's Guide*.

How application programs obtain service

Application programs and utility programs obtain services from the security product by issuing service requests to the runtime subsystem of software and hardware. Use a procedure call according to the rules of your application language.

The available services are collectively described as the security API. All the software and hardware accessed through the security API should be considered an integrated subsystem.

When the cryptographic services access layer receives requests concurrently from multiple application programs, it serializes the requests and returns a response for each request. There are other multiprocessing implications arising from the existence of a common master-key and a common key-storage facility. These implications are covered by separate topics of this publication.

The way application programs and utilities are linked to the API services depends on the computing environment. In the Linux environment, the operating system dynamically links application security API requests to the subsystem shared object library code. Compile application programs that use CCA and link the compiled programs to the CCA library. The library and its default distribution location is **/usr/lib64/libcsulcca.so**.

Together, the security API shared library and the environment-dependent request routing mechanism act as an agent on behalf of the application and present a request to the server. Requests can be issued by one or more programs. Each request is processed by the server as a self-contained unit of work. The programming interface can be called concurrently by applications running as different processes. The security API can be used by multiple threads in a process and is thread safe.

In each server environment, a device driver provided by IBM supplies low-level control of the hardware and passes the request to the hardware device. Requests can require one or more I/O commands from the security server to the device driver and hardware.

The security server and a directory server manage key storage. Applications can store locally used cryptographic keys in a key-storage facility. This is especially useful for long-life keys. Keys stored in key storage are referenced using a key label. Before deciding whether to use the key-storage facility or to let the application retain the keys, consider system design trade-off factors, such as key backup, the impact of master-key changing, the lifetime of a key, and so forth.

Overlapped processing and load balancing

You can maximize throughput by organizing your application or applications to make multiple, overlapping calls to the CCA API.

Calls to the CCA security API are synchronous, that is, your program loses control until the verb completes. Multiple processing-threads can make concurrent calls to the API.

You can maximize throughput by organizing your application or applications to make multiple, overlapping calls to the CCA API. You can also increase throughput by employing multiple coprocessors, each with CCA. You can maximize throughput by organizing your application (or applications) to make multiple, overlapping calls to the CCA API. You can also increase throughput by employing multiple coprocessors, each with CCA. Another way to maximize throughput is to make use of the AUTOSELECT option for automatic load-balancing. See “Multi-coprocessor selection capabilities” on page 9.

Within the coprocessor, the CCA software is organized into multiple threads of processing. This multiprocessing design is intended to enable concurrent use of the coprocessor's main engine, PCIe communications, DES and Secure Hash Algorithm-1 (SHA-1) engine, and modular-exponentiation engine.

Host-side key caching

Calls to the CCA security API are synchronous, that is, your program loses control until the verb completes. Multiple processing-threads can make concurrent calls to the API.

CCA provides caching of key records obtained from key storage within the CCA host code. However, the host cache is unique for each host process. If different host processes access the same key record, an update to a key record caused in one process does not affect the contents of the key cache held for other processes. Caching of key records within the key-storage system can be suppressed so all processes access the most current key-records. To suppress caching of key records, use the SET command to set the environment variable CSUCACHE to NO. If this environment variable is not set, or is set to anything other than NO, caching of key records will not be suppressed. The CSUCACHE environment variable does not impact CPACF translated key caching.

Multi-coprocessor selection capabilities

Multi-coprocessor selection capabilities allow you to employ more than one CCA coprocessor.

When more than one CCA coprocessor is installed, an application program can control which CCA coprocessor to use. It can explicitly select a CCA coprocessor, it can switch on the AUTOSELECT option, or it can optionally employ the default CCA coprocessor.

AUTOSELECT option

If switched on, the AUTOSELECT option overrides an explicit CCA coprocessor selection and default CCA coprocessor selection for all verbs (except those listed in "Verbs that ignore AUTOSELECT" on page 10).

When the AUTOSELECT option is switched on, the CCA coprocessor to be used by a verb will be selected by the operating system (the Linux device driver) from the set of available CCA coprocessors, including any coprocessors loaded with CCA user defined function (UDX) code. The Linux device driver chooses a CCA coprocessor based on a policy for load balancing.

To switch on the AUTOSELECT option, use Cryptographic Resource Allocate verb (CSUACRA). Alternatively, the AUTOSELECT option can be switched on at program start by setting the environment variable CSU_DEFAULT_ADAPTER to the value `"DEV-ANY"`. For example:

```
export CSU_DEFAULT_ADAPTER=DEV-ANY
```

To switch off the AUTOSELECT option, use Cryptographic Resource De-allocate verb (CSUACRD).

Master key coherence for AUTOSELECT

When using the AUTOSELECT option, all CCA coprocessors accessible by the operating system must have the same state. In particular, they must be configured with the same master key as appropriate for the services in use. For example, if your application uses only DES functions and you enable AUTOSELECT, then the SYM-MK should be the same across all accessible CCA coprocessors. If you use any RSA functions from PKA verbs, then the ASYM-MKs must be the same. For

AES usage the AES-MKs must match, and for ECC the APKA-MK must match.

Verbs that ignore AUTOSELECT

The following verbs ignore the AUTOSELECT option and use the explicitly selected or default CCA coprocessor instead. These verbs act as if the AUTOSELECT option does not exist, acting exactly as they did in prior releases in which AUTOSELECT was not present.

Table 3. Verbs that ignore AUTOSELECT

Verb	Explanation
CSUACFQ	When querying CCA coprocessor state it is important to retrieve data from the explicitly queried CCA coprocessor.
CSUARNT	When testing adapter health it is also important to receive explicit results.
CSNDRKL, CSNDRKD	Listing and managing the retained keys for a CCA coprocessor requires dealing with explicitly selected CCA coprocessors. Note that it is specifically not recommended to use any retained key functions (such as choosing to use retained keys with PKA verbs) when AUTOSELECT is active. By their very nature retained keys are tied to the specific CCA coprocessor where they were created
CSNBMKP	Managing the Master Keys for a CCA coprocessor also requires explicit allocation and de-allocaton to achieve expected results.
HOST-only verbs	Some verbs are not impacted by AUTOSELECT simply because they never use the CCA coprocessor. These verbs take actions that do not involve secrets guarded by the card Master Keys. These include: CSUACFV, CSNBKSI, CSNBCVG, CSNBKTB, CSNBKTB2, CSNBKTP, CSNBKTP2, CSNBT31O, CSNBT31R, CSNBT31P, CSNBAKRC, CSNBAKRD, CSNBAKRL, CSNBAKRR, CSNBAKRW, CSNBKRC, CSNBKRD, CSNBKRL, CSNBKRR, CSNBKRW, CSNDKRC, CSNDKRD, CSNDKRL, CSNDKRR, CSNDKRW, CSNDPKB, CSNDPKX
CPACF using verbs	Some verbs use the CPACF according to configuration of appropriate environment variables. These verbs will ignore AUTOSELECT if told to use CPACF. These include: CSNBOWH, CSNBDEC, CSNBENC, CSNBSAE, CSNBSAD

Explicit CCA coprocessor selection

To explicitly select a CCA coprocessor, use the Cryptographic Resource Allocate verb (CSUACRA). This verb allocates a CCA coprocessor loaded with the CCA software. When a CCA coprocessor is allocated and the AUTOSELECT option is not on, CCA requests are routed to it until it is de-allocated Similarly, when a CCA coprocessor is allocated and one of the verbs that ignore the AUTOSELECT option is used, then CCA requests are routed to it until it is de-allocated .

To de-allocate an allocated coprocessor, use the Cryptographic Resource De-allocate verb (CSUACRD). When a coprocessor is not allocated (either before an allocation occurs or after the cryptographic resource is de-allocated), requests are routed to the default coprocessor unless the AUTOSELECT option is on.

Note: The scope of the Cryptographic Resource Allocate and the Cryptographic Resource Deallocate verbs is to a thread. A multi-threaded application program can use all of the installed CCA coprocessors simultaneously. A program thread can use only one of the installed coprocessors at any given time, but it can switch to a

different installed coprocessor as needed. To perform the switch, a program thread must deallocate an allocated cryptographic resource, if any, and then it must allocate the desired cryptographic resource. The Cryptographic Resource Allocate verb fails if a cryptographic resource is already allocated.

Listing CCA coprocessors

With the first call to CCA from a process, CCA associates coprocessor designators CRP01, CRP02, and so on with specific coprocessors. The host determines the total number of coprocessors installed through a call to the coprocessor device driver. Adding, removing, or relocating coprocessors can alter the number associated with a specific coprocessor. The host then polls each coprocessor in turn to determine which ones contain the CCA firmware. As each coprocessor is evaluated, the CCA host associates the identifiers CRP01, CRP02, and so forth to the CCA coprocessors. CCA coprocessors loaded with a UDX extension are also assigned a CRP*nn* identifier.

For a specific device driver, names such as these are used: CRP*nn*, card*nn*, AP*nn*, and so forth, where the *nn* values normally do not match (for example, some start with 0, others start with 1).

To determine the coprocessor type of a card (one out of CEX2C through CEX5C), use one of these methods:

- Invoke the Cryptographic Facility Query verb (see "Determining if a card is a CEX2C, CEX3C, CEX4C, or a CEX5C" on page 77).
- Use the **lszcrypt** command (see "Confirming your cryptographic devices" on page 988).
- Use the **hwtype** attribute of your cryptographic devices in sysfs (see the version of *Device Drivers, Features, and Commands*, that applies to your distribution).
- Run **panel.exe -x** using the panel.exe utility installed with the RPM, to get a quick summary of cards available and their status. See "The panel.exe utility" on page 1003.
- Run **ivp.e**, another utility installed with the RPM, which gives more detailed information about each card available. See Chapter 27, "Utilities," on page 1003.

Note that the mapping of logical card identifiers such as CRP01 and CRP02 to physical cards in your machine is not defined. This is because the mapping can change depending on the machine and its configuration. If your application needs to identify specific coprocessor cards, you can do one of the following:

- Use the Cryptographic Facility Query verb (see "Cryptographic Facility Query (CSUACFQ)" on page 77) with the STATCARD or STATCRD2 *rule_array* keyword
- Use the panel.exe utility program with option -x, in order to read a card's serial number (see "The panel.exe utility" on page 1003).

Default CCA coprocessor

The selection of a default device occurs with the first CCA call to a coprocessor. When the default device is selected, it remains constant throughout the life of the thread. Changing the value of the environment variable after a thread uses a coprocessor does not affect the assignment of the default coprocessor. If a thread with an allocated coprocessor ends without first de-allocating the coprocessor, excess memory consumption results. It is not necessary to deallocate a

cryptographic resource if the process itself ends. It is suggested only if individual threads end while the process continues to run.

You can alter the default designation by explicitly setting the CSU_DEFAULT_ADAPTER environment variable. This is accomplished by issuing following command:

```
export CSU_DEFAULT_ADAPTER=CRPxx
```

Replace CRP*xx* with the identifier for the resource you wish to use, such as CRP02. The rpm limits the maximum value of *xx* such that CRP01 through CRP08 are the valid identifiers for the default coprocessor (limited by the number of coprocessors actually configured).

When cards of multiple types (CEX*C) are active in the same system, note the following:

- The CCA library detects CEX*C adapters and intermingle them in the CRP*nn* adapter instance list. This is a list of all available adapters, in the order that they were discovered by the device driver.
- The default adapter will be the lowest numbered CEX*C instance, that is, the newest adapter level, found by the device driver. For example, if the ordered discovery list is: [CEX3C, CEX4C, CEX3C, CEX4C], the default adapter will be the first CEX4C.
- A user can specify the proper CRP*nn* number to allocate and work with any card (CEX*C) .
- For a specific device driver, names such as these are used: CRP*nn*, card*nn*, AP*nn*, and so forth, where the *nn* values normally do not match (for example, some start with 0, others start with 1).

CPACF support

Central Processor Assist for Cryptographic Functions (CPACF) must be configured and enabled on the system before you can use it.

CPACF support has these features:

- "Environment variables that affect CPACF usage"
- "Access control points that affect CPACF protected key operations" on page 13
- "CPACF operation (protected key)" on page 14
- "CCA library CPACF preparation at startup" on page 16
- "Interaction between the *default card* and use of Protected Key CPACF" on page 17

Environment variables that affect CPACF usage

The CSU_HCPUACLR and CSU_HCPUAPRT environment variables control whether the CPACF is used for certain CCA functions.

These variables are overridden by the explicit use of the Cryptographic Resource Allocate (CSUACRA) and Cryptographic Resource Deallocate (CSUACRD) verbs to enable or disable these access patterns. To avoid confusion, the environment variables are given similar names to the keywords used by Cryptographic Resource Allocate (CSUACRA) and Cryptographic Resource Deallocate (CSUACRD).

Note: The default values listed here are valid even if these environment variables are not defined. Their settings represent default policy decisions made in the library code.

CSU_HCPUACLR

Use the CSU_HCPUACLR variable to allow CPACF for clear key operations and hashing algorithms.

Set CSU_HCPUACLR to 1 in a profile setup file or with this command:

```
export CSU_HCPUACLR=1
```

Setting this variable to any other value (except for the case where the variable has not been set, as noted above) results in disabling the use of the CPACF for clear key operations and hashing algorithms. The default is 1, meaning that the function is enabled.

Affected verbs

- MDC Generate (CSNBMDG)
- One-Way Hash (CSNBOWH)
- Symmetric Algorithm Decipher (CSNBSAD) (clear key AES)
- Symmetric Algorithm Encipher (CSNBSAE) (clear key AES)

CSU_HCPUAPRT

Use the CSU_HCPUAPRT variable to use CPACF for protected key (translated secure key) operations.

Set CSU_HCPUAPRT to 1 in a profile setup file or with this command:

```
export CSU_HCPUAPRT=1
```

Setting this variable to any other value (except for the case where the variable has not been set, as noted above) results in disabling the use of the CPACF for protected key (translated secure key) operations. The default is 0, meaning that the function is disabled.

Affected verbs

- Decipher (CSNBDEC)
- Encipher (CSNBENC)
- MAC Generate (CSNBMGN)
- MAC Verify (CSNBMVR)
- Symmetric Algorithm Decipher (CSNBSAD) (clear key AES)
- Symmetric Algorithm Encipher (CSNBSAE) (clear key AES)

Access control points that affect CPACF protected key operations

There are two access points that enable the protected key feature.

These two access points are:

Symmetric Key Encipher/Decipher - Encrypted DES keys
This is bit X'0295', and is set ON by default.

This ACP enables translating DES keys for use with the CPACF. Without this bit set ON, the call to the CEX*C to rewrap the key under the CPACF

wrapping key will fail with a return code 8 and reason code 90, which will in turn imply disabling the use of this function by the host user. This error will not be returned to the user, instead the operation will be sent to the CEX*C. Because the default value of the bit is ON, it is assumed that the user will know that it is set OFF on purpose. A return code 8 and reason code 90 will cause no further requests to go to the CEX*C verb that translates keys, in an effort to preserve normal path performance.

Symmetric Key Encipher/Decipher - Encrypted AES keys
This is bit X'0296', and is set ON by default.

This ACP enables translating AES keys for use with the CPACF. Without this bit set ON, the call to the CEX*C to rewrap the key under the CPACF wrapping key will fail with a return code 8 and reason code 90, which will in turn imply disabling the use of this function by the host user. This error will not be returned to the user, instead the operation will be sent to the CEX*C. Because the default value of the bit is ON, it is assumed that the user will know that it is set OFF on purpose. A return code 8 and reason code 90 will cause no further requests to go to the CEX*C verb that translates keys, in an effort to preserve normal path performance.

CPACF operation (protected key)

These are details for Central Processor Assist for Cryptographic Functions (CPACF) usage by the host library.

Note that at system power-on, the CPACF generates a new Key Encryption Key (KEK, kek-t) for wrapping translated keys.

Figure 2 on page 15 illustrates the CPACF layer as it relates to the security access API and cryptographic engine. The CPACF exploitation layer examines commands received by the security server to see if they can be redirected to the CPACF. If so, this layer makes preparations (including translating secure keys to protected keys), and then call the CPACF directly. If all preparations and the CPACF operations are successful, the results are returned as a normal return through the security server. For any errors, the command is redirected back through the security server to the normal path, using the allocated CEX*C for the thread making the call.

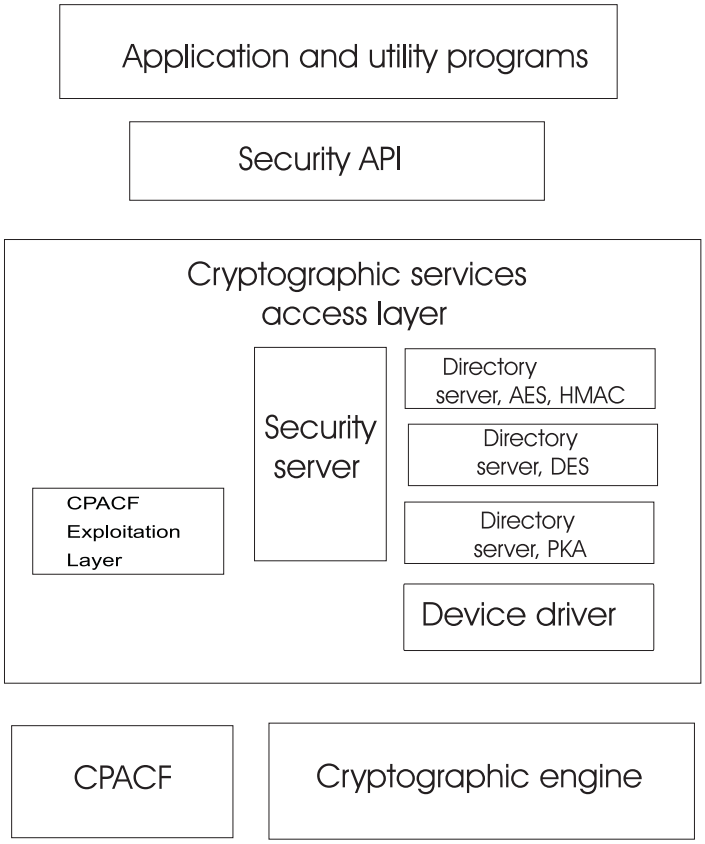


Figure 2. CPACF

Clear key or No key: For operations that do not use keys (such as hash algorithms) or operations that use keys that are not encrypted under the card master key, (called *clear keys*), no translation is necessary and the CPACF is used immediately.

Protected key: The device driver and the other layers are used for protected key support, for translating keys. This relationship is similar to the 'directory server' relationship: a translation layer invisible to the customer. After translation the 'translated-key' is stored in an invisible runtime cache so that the next use of the key can avoid the translation step. For protected key usage, a CEX*C feature must be available and allocated for use by the thread.

CPACF service actions and running applications: The CPACF is an independent hardware unit, like the CEX*C itself, and can be independently configured available or unavailable while an S/390 Linux instance is running by service technicians performing service actions. If the CPACF is cycled it will generate a new wrapping key for translated keys, invalidating all of the keys in the CCA library key translation cache. Therefore, it is never advisable to attempt such a service action while there are system instances with applications running that use the CPACF.

If such an action is undertaken, applications should be stopped and restarted so that the **libcsulcca.so** is unloaded from memory and reloaded. This will cause the key cache to be cycled. A more complete measure would be to reboot system images. If these precautions are disregarded and a CPACF service action is

undertaken as described, application crashes may ensue with a SIGSEGV error. This could occur due to translated keys wrapped under outdated CPACF wrapping keys being used.

A normal system-wide power cycle will cause the CPACF to generate a new wrapping key by design, however, this action also of course cycles all of the hosted system LPARs and VM system images so there is no problem; translated keys are not cached in permanent storage.

Using keys with CPACF, protected key

Follow the steps in this procedure to use keys with CPACF, protected key.

Procedure

1. An eligible CCA verb call (see lists in "Access control points that affect CPACF protected key operations" on page 13) specifying a key token or key identifier for a key token that is a normal internal CCA key token, called key-e here, comes into the CCA library.
2. The CCA library verifies that a CEX*C is available for key translation. If not, then the standard **no-available-device** error is returned.
3. The CCA library tries to find an already translated version (key-t) that matches the key-e passed into the CCA library.
   - The user application (CCA library in this case) must cache translated key-t objects in RAM, using the key-e tokens as references.
4. If a key-t is not found for the key-e used:

   The CCA library translates the key-e to a key-t for use with the CPACF using CCA secure services, then caches the key pair.
5. At this point, either a fresh key-t has been obtained, or a key-t was found in RAM cache for the operation.
6. The CCA library directs the operation to the CPACF using the key-t.

Results

The **panel.exe -m** command displays all the supported CPACF functions. This is especially useful on a z/VM system, to make sure that the protected key functions are available. For details, see "The panel.exe utility" on page 1003.

Using keys with CPACF, clear key or no key

Follow the steps in this procedure to use keys with CPACF, clear key or no key.

Procedure

1. An eligible CCA verb call (see lists in "Access control points that affect CPACF protected key operations" on page 13) comes into the CCA library.
2. No CEX*C is necessary, so no check for availability or Cryptographic Resource Allocate (CSUACRA) call will be implied.
3. The CCA library prepares an appropriate CPACF clear key (key-c) structure using the clear key passed to the CCA verb (key-v).
4. The CCA library directs the operation to the CPACF using the key-c.

CCA library CPACF preparation at startup

When the CCA library first starts up, it must take some initialization steps to prepare for using the Central Processor Assist for Cryptographic Functions (CPACF).

Procedure

1. Check configuration options to see if either is set to 'on', allowing some use of the CPACF. If neither is on, skip the rest of initialization
2. Check for existence and configuration of the CPACF.

Interaction between the *default card* and use of Protected Key CPACF

While the CPACF can be used to encrypt and decrypt data in the absence of a CEX*C, for protected key operations a CEX*C is still necessary and it must be the allocated or default adapter for the thread doing the processing.

Note that the service that translates the keys is not available on the CEX2C, for any CCA firmware version. Therefore the default adapter must be CEX3C or newer in order to use protected key operations with CPACF.

Using the AUTOSELECT option and the use of protected key CPACF

While the CPACF can be used to encrypt and decrypt data in the absence of a CEX*C, for protected key operations a CEX*C is still necessary, because the users' key tokens are translated with a service only available on the CEX*C for use with the CPACF.

Therefore, when enabling the AUTOSELECT option, all CCA coprocessor adapters available to the operating system must be CEX3C or newer coprocessors (no CEX2C). See "Multi-coprocessor selection capabilities" on page 9 for more information.

Security API programming fundamentals

You obtain CCA cryptographic services from the coprocessor through procedure calls to the CCA security application programming interface (API).

Most of the services provided are considered an implementation of the IBM Common Cryptographic Architecture (CCA). Most of the extensions that differ from other IBM CCA implementations are in the area of the access-control services. If your application program is used with other CCA products, compare the product literature for differences.

Your application program requests a service through the security API by using a procedure call for a verb. The term *verb* implies an action that an application program can initiate. Other publications might use the term *callable service* instead. The procedure call for a verb uses the standard syntax of a programming language, including the entry-point name of the verb and the parameters of the verb. Each verb has an entry-point name and a fixed-length parameter list.

The security API is designed for use with high-level languages, such as C, COBOL, or RPG and for low-level languages, such as assembler language. It is also designed to enable you to use the same verb entry-point names and variables in the various supported environments. Therefore, application code you write for use in one environment generally can be ported to additional environments with minimal change.

Verbs, variables, and parameters

Certain information is included for each verb, including the entry-point name and the parameter list.

Each verb has an entry-point name and a fixed-length parameter list. Part 2, "CCA verbs," on page 75 describes each verb, and includes the following information for each verb:

- Pseudonym
- Entry-point name
- Description
- Format
- Parameters
- Restrictions
- Required commands
- Usage notes
- Related information
- JNI version

Pseudonym
Also known as a general-language name or verb name. This name describes the function that the verb performs, such as Key Generate.

Entry-point name
Also known as a computer-language name. This name is used in your program to call the verb. Each verb's 7 or 8 character entry-point name begins with one of the following prefixes:

CSNB Generally, the AES and DES verbs

CSND Public key cryptography verbs, including RSA and Elliptic Curve

CSUA Cryptographic-node and hardware-control verbs

The last three or four letters in the entry-point name after the prefix identify the specific verb in a group and are often the first letters of the principal words in the verb pseudonym.

When verbs are described throughout this publication, they are sometimes referred to by the pseudonym, and at other times by the pseudonym followed by the entry-point name in parenthesis. An example of this is: Key Generate (CSNBKGN).

The verb prefixes used here are different from those used by IBM's Integrated Cryptographic Service Facility (ICSF).

Description
The verb is described in general terms. Be sure to read the parameter descriptions because these add additional detail.

Format
The format section for each verb lists the entry-point name followed by the list of parameters for the verb. You must code all the parameters, and they must be in the order listed.

```
entry-point name(
                 return_code,
                 reason_code,
                 exit_data_length,
                 exit_data,
                 parameter_5,
                 parameter_6,
                 ...
                 parameter_n)
```

Parameters

All information exchanged between your application program and a verb is through the variables identified by the parameters in the procedure call. These parameters are pointers to the variables contained in application program storage that contain information to be exchanged with the verb. Each verb has a fixed-length parameter list and though all parameters are not always used by the verb, they must be included in the call.

The first four parameters are the same for all of the verbs. For a description of these parameters, see "Parameters common to all verbs" on page 20. For the description of the remaining parameters, see the definitions with the individual verbs.

In the description for each parameter, data flow direction and data type are indicated, as follows.

Direction: *direction*
Type: *data type*

direction The parameter descriptions use the following terms to identify the flow of information:

Input The application program sends the variable to the verb (to the called routine).

Output
The verb returns the variable to the application program.

Input/Output
The application program sends the variable to the verb or the verb returns the variable to the application program, or both.

data type Data identified by a verb parameter can be a single value or a one-dimensional array. If a parameter identifies an array, each data element of the array is of the same data type. If the number of elements in the array is variable, a preceding parameter identifies a variable that contains the actual number of elements in the associated array. Unless otherwise stated, a variable is a single value, not an array.

For each verb, the parameter descriptions use the following terms to describe the type of variable:

Integer
A CCA integer (CCAINT). On Linux on z Systems, this is defined as the system type *long*. On other platforms this has been defined as a 4-byte (32-bit), signed, two's-complement binary number. (CCA for Linux on z Systems has always defined the CCA integer as *long*).

String A series of bytes where the sequence of the bytes must be maintained. Each byte can take on any bit configuration. The string

consists only of the data bytes. No string terminators, field-length values, or typecasting parameters are included. Individual verbs can restrict the byte values within the string to characters or numerics.

Character data must be encoded in the native character set of the computer where the data is used. Exceptions to this rule are noted where necessary.

Array An array of values, which can be integers or strings. Only one-dimensional arrays are permitted. For information about the parameters that use arrays, see "The **rule_array** and other keyword parameters" on page 21.

Restrictions
Any restrictions are noted.

Required commands
Any access control points required to use the verb are described here.

Usage notes
Usage notes about this verb are listed.

Related information
Any related information is noted.

JNI version
If the verb has a Java Native Interface version, it is described.

Commonly encountered parameters

Some parameters are common to all verbs and other parameters are used with many of the verbs.

- "Parameters common to all verbs"
- "The **rule_array** and other keyword parameters" on page 21
- "Key tokens, key labels, and key identifiers" on page 21

Parameters common to all verbs

A parameter is an address pointer to the associated variable in application program storage.

The first four parameters (**return_code**, **reason_code**, **exit_data_length**, and **exit_data**) are the same for all verbs:

return_code
The return code specifies the general result of the verb. Chapter 16, "Return codes and reason codes," on page 751 lists the return codes.

reason_code
The reason code specifies the result of the verb that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Chapter 16, "Return codes and reason codes," on page 751 lists the reason codes.

exit_data_length
A pointer to an integer value containing the length of the string (in bytes) that is returned by the **exit_data** value. This parameter should point to a value of zero, to ensure compatibility with any future extension or other operating environment.

exit_data
The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

Restriction: The **exit_data_length** and **exit_data** variables must be declared in the parameter list. The **exit_data_length** parameter should be set to B'0'.

The rule_array and other keyword parameters

The **rule_array** parameter and some other parameters use keywords to transfer information.

Generally, a **rule_array** consists of a variable number of data elements that contain keywords that control specific details of the verb process. Almost all keywords, in a **rule_array** or otherwise, are eight bytes in length, and should be uppercase, left-aligned, and padded on the right with space characters. Not all implementations fold lowercase characters to uppercase so you should always code the keywords in uppercase.

The number of keywords in a **rule_array** is specified by a **rule_array_count** variable, an integer that defines the number of 8-byte elements in the array.

In some cases, a **rule_array** is used to convey information other than keywords between your application and the server. This is, however, an exception. For a list of key types that are passed in the **rule_array** keyword, see Table 5 on page 42.

Key tokens, key labels, and key identifiers

Essentially all cryptographic operations employ one or more keys. In CCA, keys are retained within a structure called a *key token*.

A verb parameter can point to a variable that contains a key token. Generally you do not need to be concerned with the details of a key token. You can deal with it as an entity.

Key tokens are described as either internal, operational, or external, as follows:

Internal
A key token that contains an encrypted key for local use. The cryptographic engine decrypts an internal key to use the key in a local operation. When a key is entered into the system, it is always encrypted if it appears outside the protected environment of the cryptographic engine. The engine has a special key-encrypting key, called a master key. This key is held within the engine to wrap and unwrap locally used keys.

Operational
An internal key token that is complete and ready for use and contains a key that is encrypted under a master key. During entry of a key, the internal key-token can have a flag set indicating the key information is incomplete.

External
A key token that contains a key that is either in the clear or is encrypted by some key-encrypting key other than the master key. Generally, when a key is to be transported from place to place or is to be held for a significant period of time, the key must be encrypted with a transport key. A key wrapped by a (transport) key-encrypting key is designated as being external.

RSA and ECC public-keys are not encrypted values and, when not accompanied by private-key information, are retained in an external key-token.

Internal key tokens can be stored in a file maintained by the directory server. These key tokens are referenced by use of a key label. A *key label* is an alphanumeric string you place in a variable and reference with a verb parameter.

Parameter descriptions specify how you can provide a key using these terms:

Key token
The parameter must contain a proper key-token structure.

Key label
The parameter must contain a key-label string used to locate a key record in key storage.

Key identifier
The parameter must contain either a key token or a key label. The first byte in the parameter indicates whether it contains a key token or a key label.

X'00' indicates a DES null key-token.

range X'01' - X'1F'
indicates that the variable is processed as a key token.

range X'20' - X'FE'
indicates that the variable is processed as a key label. There are additional restrictions on the value of a key label.

X'FF' raises an error condition when passed to the API.

How to compile and link CCA application programs

The CCA RPM includes C libraries that you can use to build CCA C applications.

One of these libraries also supports a Java Native Interface (JNI) that you can use to build CCA Java applications. The CCA RPM also includes Java libraries with front end classes for the JNI.

The C libraries and their default distribution locations are:

/usr/lib64/libcsulcca.so.*
This library contains all CCA verbs apart from the Master Key Process (CSNBMKP) verb. This library also contains the C support for the JNI.

/usr/lib64/libcsulccamk.so.*
This library contains the Master Key Process (CSNBMKP) verb and is required for applications that use the master key process (see "Using the Master Key Process (CSNBMKP) verb" on page 23).

CCA 5.0 includes two versions of the Java libraries:

New version introduced with CCA 4.2

As of CCA 4.2, new Java libraries that use the Java package infrastructure are included. These Java libraries and their default distribution locations are:

/opt/IBM/CCA/cnm/CNM.jar
This library contains the data classes used by the JNI (see "Entry points and data types used in the JNI" on page 24) and most JNI verb front end classes.

/opt/IBM/CCA/cnm/CNMMK.jar
This library contains the JNI front end class for the Master Key Process (CSNBMKP) verb.

Deprecated version - removed starting with CCA 5.0

Note that the CCA 5.0 rpm does no longer contain the files mentioned in this section. Hoewever, CCA 4.2 still includes the Java libraries of earlier CCA versions to support your existing applications. These libraries contain all verbs available with CCA 4.2 but future additions will not be available in these libraries. The default distribution locations of these deprecated libraries are:

/opt/IBM/CCA/cnm/HIKM.zip
This library contains the data classes used by the JNI (see "Entry points and data types used in the JNI" on page 24) and most JNI verb front end classes.

/opt/IBM/CCA/cnm/HIKMMK.zip
This library contains the JNI front end class for the Master Key Process (CSNBMKP) verb.

See "Methods for calling the CCA JNI" on page 24 for more information about the two versions of the Java libraries.

The CCA RPM also includes C and Java sample programs that help you develop your application (see "Building C applications using the CCA libraries" and "JNI sample modules and sample code" on page 25).

Using the Master Key Process (CSNBMKP) verb

`/usr/lib64/libcsulccamk.so` contains the Master Key Process (CSNBMKP) verb.

Any use of the `libcsulccamk.so` library is restricted because the library is installed so that only the `root` user (user ID of 0) and members of the group `cca_admin` have read access. The `cca_admin` group is added by the CCA RPM installation procedure. This is done to limit the ability of an untrusted user to copy the library with the purpose of reverse-engineering the master-key access methods inside it.

Furthermore, use of some specific access methods through the Master Key Process (CSNBMKP) verb are restricted to a corresponding Linux group membership of the user trying to make that access. Table 272 on page 997 contains a list of the groups and their functions.

Users without the required group membership are denied use. For more information, see **Master key load** (Step 7 on page 995).

Building C applications using the CCA libraries

Perform these steps to build a C program from a make file.

Procedure

1. Change to the directory that contains the make file.
2. Issue this command to compile the program:

```
make -f <makefile>
```

For example, to use the make file of “Sample program in C” on page 977, issue:

```
make -f makefile.lnx
```

Using the CCA JNI

You can use the Java native interface to work with the CCA.

Methods for calling the CCA JNI

For CCA 5.0, there is only one method for calling the CCA JNI. The deprecated method used with prior CCA versions is no longer available for CCA 5.0.

For more information, see topic “JNI sample modules and sample code” on page 25 about available sample programs.

Method using the Java package infrastructure:

This method was introduced with CCA 4.2.

When using this method, you need `import` statements in the Java source files. The new JNI JAR files used by this method are `CNM.jar` and, for the master key process, `CNMMK.jar`.

This is the preferred method to use when developing new applications.

See “Building the Java byte code” on page 26 for details about compiling and running Java applications with this method.

Deprecated method used with prior CCA versions -- removed for CCA 5.0:

With CCA versions before 4.2, the CCA JNI was called using the `HIKM.zip` JAR file and, for the master key process, the `HIKMMK.zip` JAR file.

To support existing applications, these JAR files are included in CCA 4.2, and they provide access to all CCA 4.2 functions.

This method was deprecated for CCA 4.2 and has been removed starting with CCA 5.0.

Entry points and data types used in the JNI

The Java entry points of CCA verbs are similar to the C entry points, except that a letter `J` is appended to the entry point name.

For example, CSNBKGN is the C entry point for the Key Generate verb, and CSNBKGNJ is the Java entry point for this verb. The detailed verb descriptions in Part 2, “CCA verbs,” on page 75 include a section for the JNI interface.

These two data types are defined and used in the JNI:

hikmNativeInteger
: 64-bit native signed integer (type long), matching the C interface

Byte *
: general pointer type to unsigned byte

A file named `hikmNativeInteger.html` provides information about this class. The default location of this file is `/opt/IBM/CCA/doc` directory.

JNI sample modules and sample code

To illustrate the two JNI access methods and also how to use the CCA JNI to call CCA verbs, sample modules are provided.

mac.java
: illustrates JNI calls versus C calls to CCA.

  This sample program calls the same CCA verbs as the sample C language program named `mac.c`.

  This sample program uses the package infrastructure of the new JNI access method. For more information about this sample program, see "Sample program in Java" on page 981.

RNGpk.java
: illustrates the method available for calling the JNI.

  - Java class implementations (`*.java` files) that call CCA JNI functions need an import line, such as this:

    ```
    import com.ibm.crypto.cca.jni.*;
    ```

  - The Java classpath must point to `CNM.jar`.

The default location of the sample code is `/opt/IBM/CCA/samples` for both, SUSE and Red Hat distributions.

Preparing your Java environment

Before you can compile and run Java applications that use the CCA Java Native Interface (JNI), you must install a Java version that is supported by your distribution.

Install Java from the distribution installation media or from other authorized sources for that distribution.

The CCA JNI has been tested with these Java versions:

- Java 1.5.0 for Red Hat Enterprise Linux 6.6
- Java 1.7.0 for SUSE Linux Enterprise Server 11 SP3
- Java 1.7.0 for SUSE Linux Enterprise Server 12
- Java 1.7.0 for Red Hat Enterprise Linux 5.11
- Java 1.8.0 for Red Hat Enterprise Linux 7.1

Later versions of Java, as provided with a distribution, might work, although they have not been tested.

For compiling and running applications that use the CCA JNI, you need access to the **java** and **javac** executables. Use one of the following methods to ensure that you can call the command without preconditions:

- Add the path to the **java** and **javac** executable to your PATH environment variable.
- Create soft links from the **java** and **javac** executables from wherever they are located to a directory that is in your PATH environment variable by default, such as `/usr/bin`.

Note: IBM Java 1.7.0 interaction

Some users reported an issue with IBM Java, support item IV52646. The following exception occurs:

```
Exception in thread "main" java.lang.UnsatisfiedLinkError: nio
(/usr/lib64/jvm/java-1.7.0-ibm-1.7.0/jre/lib/s390x/libnio.so:
symbol NET_Bind, version SUNWprivate_1.1 not defined in file libnet.so
with link time reference)
```

This occurs because the s390x version of the `libnio.so` library is being loaded when a java program is run. The correct version is:

```
/usr/lib64/jvm/java-1.7.0-ibm-1.7.0/jre/lib/s390/libnio.so
```

To ensure that the application uses the correct version, do the following in the application environment as a test, and then add an appropriate adjustment to the application environment file (such as .bashrc or .bash_profile):

```
export LD_LIBRARY_PATH=/usr/lib64/jvm/java-1.7.0-ibm-1.7.0/jre/lib/s390x:$LD_LIBRARY_PATH
```

Building Java applications using the CCA JNI

Call the CCA JNI using the Java package infrastructure.

This method was introduced with CCA 4.2 and is the preferred method for new applications.

Building the Java byte code:

Perform these steps to compile a Java source file.

Procedure

1. Ensure that your LD_LIBRARY_PATH variable points to the CCA libraries. This example points to the default location:
   ```
   export LD_LIBRARY_PATH=/usr/lib64
   ```
2. Change to the directory that contains the source code file of the program you want to compile.
3. Issue this command to compile a Java source code file <*program*>.java:
   ```
   javac -classpath <fullpath>/CNM.jar <program>.java
   ```
   where <*fullpath*> is the location of CNM.jar.

   The following example uses the default path and the sample program of "Sample program in Java" on page 981:
   ```
   javac -classpath /opt/IBM/CCA/cnm/CNM.jar RNGpk.java
   ```
   If the program uses the Master Key Process (CSNBMKP) verb, you must also include CNMMK.jar in the class path (see "Using the Master Key Process (CSNBMKP) verb" on page 23). Issue this command to compile such programs:
   ```
   javac -classpath <fullpath>/CNM.jar:<fullpath>/CNMMK.jar <program>.java
   ```

   Tip: Instead of using the **-classpath** option, you can set the CLASSPATH variable to point to CNM.jar and, if required, CNMMK.jar.

Running the Java byte code:

Perform these steps to run a compiled Java program.

Procedure

1. Change to the directory that contains the compiled Java program you want to run.
2. Issue this command to run a program <*program*>.class:
   ```
   java -classpath <fullpath>/CNM.jar:. <program>.class
   ```

where <*fullpath*> is the location of CNM.jar. The period (.) at the end of the class path ensures that Java can find <*program*>.class in the current directory.

The following example uses the default path and the compiled sample program of "Sample program in Java" on page 981:

```
javac -classpath /opt/IBM/CCA/cnm/CNM.jar:. RNGpk.class
```

If the program uses the Master Key Process (CSNBMKP) verb in addition to other verbs, you must also include CNMMK.jar in the class path (see "Using the Master Key Process (CSNBMKP) verb" on page 23). Issue this command to run such programs:

```
java -classpath <fullpath>/CNM.jar:<fullpath>/CNMMK.jar:. <program>.class
```

Tip: Instead of using the **-classpath** option, you can set the CLASSPATH variable to point to CNM.jar and, if required, CNMMK.jar.

Chapter 2. Using AES, DES, and HMAC cryptography and verbs

You can use AES, DES, and HMAC cryptographic functions that CCA provides. CCA also provides cryptographic key functions, and you can use CCA to build key tokens.

The CEX*C protects data from unauthorized disclosure or modification. This coprocessor protects data stored within a system, stored in a file off a system on magnetic tape, and sent between systems. The coprocessor also authenticates the identity of customers in the financial industry and authenticates messages from originator to receiver. The coprocessor uses cryptography to perform these functions.

The CCA API for the coprocessor provides access to cryptographic functions through verbs. A verb is a routine that receives control using a function call from an application program. Each verb performs one or more cryptographic functions, including:

- Generating and managing cryptographic keys
- Enciphering and deciphering data with encrypted keys using either the U.S. National Institute of Standards and Technology (NIST) Data Encryption Standard (DES) or Advanced Encryption Standard (AES)
- Re-enciphering text from encryption under one key to encryption under another key
- Encoding and decoding data with clear keys
- Generating random numbers
- Ensuring data integrity and verifying message authentication
- Generating, verifying, and translating personal identification numbers (PINs) that identify a customer on a financial system

Functions of the AES, DES and HMAC cryptographic keys

The CCA API provides functions to create, import, and export AES, DES, and HMAC keys.

Key separation

The cryptographic coprocessor controls the use of keys by separating them into unique types, allowing you to use a specific type of key only for its intended purpose.

For example, a key used to protect data cannot be used to protect a key.

A CCA system has only one DES or AES master key. However, to provide for key separation, the cryptographic coprocessor automatically encrypts each type of key in a fixed-length token under a unique variation of the master key. Each variation of the master key encrypts a different type of key. Although you enter only one master key, you have a unique master key to encrypt all other keys of a certain type.

Master key variant for fixed-length tokens

Whenever the master key is used to encipher a key, the cryptographic coprocessor produces a variation of the master key according to the type of key that the master key will encipher.

These variations are called *master key variants*. The cryptographic coprocessor creates a master key variant by XORing a fixed pattern, called a *control vector*, onto the master key. A unique control vector is associated with each type of key. For example, all the different types of data-encrypting, PIN, MAC, and transport keys each use a unique control vector which is XORed with the master key in order to produce the variant. The different key types are described in "Types of keys" on page 36.

Each master key variant protects a different type of key. It is similar to having a unique master key protect all the keys of a certain type.

The master key, in the form of master key variants, protects keys operating on the system. A key can be used in a cryptographic function only when it is enciphered under a master key. When systems want to share keys, transport keys are used to protect keys sent outside of systems. When a key is enciphered under a transport key, the key cannot be used in a cryptographic function. It must first be brought on to a system and enciphered under the system's master key, or exported to another system where it will then be enciphered under that system's master key.

Transport key variant for fixed-length tokens

Like the master key, the coprocessor creates variations of a transport key to encrypt a key according to its type.

This allows for key separation when a key is transported off the system. A *transport key variant*, also called *key-encrypting key variant*, is created the same way a master key variant is created. The transport key's clear value is XORed with a control vector associated with the key type of the key it protects.

Note: To exchange keys with systems that do not recognize transport key variants, the coprocessor allows you to encrypt selected keys under a transport key itself, not under the transport key variant. For more information, see NOCV Importers and Exporters on page "NOCV importers and exporters" on page 39.

Key forms

A key that is protected under the master key is in *operational form*, which means the coprocessor can use it in cryptographic functions on the system.

When you store a key with a file or send it to another system, the key is enciphered under a transport key rather than the master key. The transport key is a key shared by your system and another system for the purpose of securely exchanging other keys. When CCA enciphers a key under a transport key, the key is not in operational form and cannot be used to perform cryptographic functions.

When a key is enciphered under a transport key, the sending system considers the key in *exportable form*. The receiving system considers the key in *importable form*. When a key is re-enciphered from under a transport key to under a system's master key, it is in operational form again.

Enciphered keys appear in three forms. The form you need depends on how and when you use a key.

- **Operational** key form is used at the local system. Many verbs can *use* an operational key form.

  The Key Generate, Key Import, Data Key Import, Clear Key Import, and Multiple Clear Key Import verbs can *create* an operational key form.
- **Exportable** key form is transported to another cryptographic system. It can be passed only to another system. The CCA verbs cannot use it for cryptographic functions. The Key Generate, Data Key Export, and Key Export verbs produce the exportable key form.
- **Importable** key form can be transformed into operational form on the local system. The Key Import verb (CSNBKIM) and the Data Key Import verb (CSNBDKM) can *use* an importable key form. Only the Key Generate verb (CSNBKGN) can *create* an importable key form.

For more information about the key types, see "Functions of the AES, DES and HMAC cryptographic keys" on page 29. See Chapter 18, "Key forms and types used in the Key Generate verb," on page 893 for more information about key form.

Symmetric key (DES, AES) flow

The conversion from one key to another key is considered to be a one-way flow. An operational key form cannot be turned back into an importable key form. An exportable key form cannot be turned back into an operational or importable key form. The flow of CCA key forms can be in only one direction:

```
IMPORTABLE —to→ OPERATIONAL —to→ EXPORTABLE
```

Key token

CCA supports two types of symmetric key tokens, fixed-length and variable-length.

An AES or DES fixed-length token is a 64-byte field composed of a key value and control information in the control vector. An HMAC key token is a variable-length token composed of a key value and control information. The control information is assigned to the key when the coprocessor creates the key. The key token can be either an internal key token, an external key token, or a null key token. Through the use of key tokens, CCA can do the following:

- Support continuous operation across a master key change
- Control use of keys in cryptographic services

If the first byte of the key identifier is X'01', the key identifier is interpreted as an **internal key token**. An internal key token is a token that can be used only on the CCA system that created it or another CCA system with the same host master key. It contains a key that is encrypted under the master key.

An application obtains an internal key token by using one of the verbs such as those listed below. The verbs are described in detail in Chapter 6, "Managing AES and DES cryptographic keys," on page 129.

- AES Key Record Read
- Clear Key Import
- Data Key Import
- DES Key Record Read
- Key Generate

- Key Generate2
- Key Import
- Key Part Import
- Key Part Import2
- Key Token Build
- Key Token Build2
- Multiple Clear Key Import
- Symmetric Key Import2

The master key could be dynamically changed between the time that you invoke a verb, such as the Key Import verb, to obtain a key token, and the time that you pass the key token to the Encipher verb. When a change to the master key occurs, the coprocessor will still successfully use the key, because it stores a copy of the old master key as well as the new one.

Attention: If an internal key token held in user storage is not used while the master key is changed twice, the internal key token is no longer usable. A return code of 0 with a reason code of 10001 notifies you that the master key used to decrypt the key used in your operation was an old master key, as a reminder that you should use one of the Key Token Change verbs to re-encipher your key under the current or new master key (as desired, see verbs for description).

If the first byte of the key identifier is X'02', the key identifier is interpreted as an **external key token**. By using the external key token, you can exchange keys between systems. It contains a key that is encrypted under a key-encrypting key.

An external key token contains an encrypted key and control information to allow compatible cryptographic systems to:

- Have a standard method of exchanging keys
- Control the use of keys through the control vector
- Merge the key with other information needed to use the key

An application obtains the external key token by using one of the verbs such as those listed below. They are described in detail in Chapter 6, “Managing AES and DES cryptographic keys,” on page 129.

- Key Generate
- Key Export
- Data Key Export
- Symmetric Key Export

If the first byte of the key identifier is X'00', the key identifier is interpreted as a **null key token**. Use the null key token to import a key from a system that cannot produce external key tokens. That is, if you have an 8 or 16-byte key that has been encrypted under an importer key, but is not imbedded within a token, place the encrypted key in a null key token and then invoke the Key Import verb to get the key in operational form.

For debugging information, see Chapter 17, “Key token formats,” on page 769 for the format of internal, external, or null key tokens.

Key wrapping

CCA supports two methods of wrapping the key value in a fixed-length symmetric key token for DES and AES keys: the original ECB wrapping and an enhanced CBC wrapping method, which is ANSI X9.24 compliant.

These methods use the 64-byte token. HMAC keys use a variable length token with associated data and the payload wrapping method. Variable-length tokens are wrapped by using the AESKW wrapping method that is defined in ANSI X9.102.

AES key wrapping

The key value in AES tokens are wrapped using the AES algorithm and cipher block chaining (CBC) mode of encryption.

The key value is left-aligned in a 32-byte block, padded on the right with zero, and encrypted.

The enhanced wrapping of an AES key (*K) using an AES *MK is defined as:

$e_{*MK}(*K) = e_{cbcMK}(*K)$

where:

e_{*k}(m) or e_{*kek}(*k)
: message **m** is encrypted (**e**) with key ***k** or key ***k** is encrypted with key encrypting key ***kek**

ecbc$_{*k}$(m)
: message **m** is encrypted (**e**) with key ***k** using the cipher block chaining (**cbc**) mode of operation

DES key wrapping

The key value in a DES key token are wrapped using one of two possible methods.

The two methods are:

Original method
: The key value in DES tokens are encrypted using triple-DES encryption, and key parts are encrypted separately.

Enhanced method
: The key value for keys is bundled with other token data and encrypted using triple-DES encryption and cipher block chaining mode. The enhanced method applies only to DES key tokens. The enhanced method of symmetric key wrapping is designed to be ANSI X9.24 compliant. This method was introduced with CCA 4.1.0.

ECB wrapping of DES keys (original method)

In ECB wrapping, a double length key (*K) is wrapped using a double-length key-encrypting key (*KEK).

The definition is as follows:

$e_{*KEK}(KL) \,||\, e_{*KEK}(KR) = e_{KEKL}(d_{KEKR}(e_{KEKL}(KL))) \,||\, e_{KEKL}(d_{KEKR}(e_{KEKL}(KR)))$

where:

KL is the left 64 bits of *K

KR is the right 64 bits of *K

KEKL is the left 64 bits of *KEK

KEKR is the right 64 bits of *KEK

| | means concatenation

d_{*k}(m) or e_{*k}(m)
means that message **m** is decrypted (**d**) or encrypted (**e**) with key ***k**.

Enhanced CBC wrapping of DES keys

The enhanced CBC wrapping method uses triple-DES encryption, an internal chaining of the key value, and CBC mode.

The enhanced wrapping of a double-length key (*K) using a double-length key-encrypting key (*KEK) is defined as:

```
e*KEK(*KL) = ecbcKEKL(dcbcKEKR(ecbcKEKL(KLPRIME || KR)))
KLPRIME = KL XOR SHA1(KR)
```

Where:

KL Is the left 64 bits of *K

KR Is the right 64 bits of *K

KLPRIME
Is the 64-bit modified value of KL

KEKL Is the left 64 bits of *KEK

KEKR Is the right 64 bits of *KEK

SHA1(X)
Is the 160-bit SHA-1 hash of X

| | Means concatenation

XOR Means bitwise exclusive OR

ecbc Means encryption using cipher block chaining mode

dcbc Means decryption using cipher block chaining mode

Wrapping key derivation for enhanced wrapping of DES keys

The wrapping key is exactly the same key that is used by the legacy wrapping method (the only method used by CCA 4.0.0), with one exception.

Instead of using the base key itself (master key or key-encrypting key), a key that is derived from that base key is used. The derived key will have the control vector applied to it in the standard CCA manner, and then use the resulting key to wrap the new-format target key token.

The reason for using a derived key is to ensure that no attacks against this wrapping scheme are possible using the existing CCA functions. For example, it was observed that an attack was possible by copying the wrapped key into an ECB CCA key token, if the wrapping key was used instead of a derivative of that key.

The key will be derived using a method defined in the U.S. National Institute of Standards and Technology (NIST) standard SP 800-108, *Recommendation for Key Derivation Using Pseudorandom Functions* (October, 2009). Derivation will use the

method *KDF in counter mode* using pseudo-random function (PRF) HMAC-SHA256. This method provides sufficient strength for deriving keys for any algorithm used.

The HMAC algorithm is defined as:

```
HMAC(K, text) = H((K0 XOR opad ) || H((K0 XOR ipad) || text))
```

Where:

H Is an approved hash function.

K Is a secret key shared between the originator and the intended receivers.

K0 The key K after any necessary preprocessing to form a key of the proper length.

ipad Is the constant X'36' repeated to form a string the same length as K0

opad Is the constant X'5C' repeated to form a string the same length as K0

text Is the text to be hashed.

|| Means concatenation

XOR Means bitwise exclusive OR

If the key K is equal in length to the input block size of the hash function (512 bits for SHA-256), K0 is set to the value of K. Otherwise, K0 is formed from K by hashing or padding.

The Key Derivation Function (KDF) specification calls for inputs optionally including two byte strings, Label and Context. The Context will not be used. The Label will contain information on the usage of this key, to distinguish it from other derivations that CCA may use in the future for different purposes. Because the security of the derivation process is rooted in the security of the derivation key and in the HMAC and Key Derivation Functions (KDF) themselves, it is not necessary for this label string to be of any particular minimum size. The separation indicator byte of X'00' specified in the NIST document will follow the label.

The label value will be defined so that it is unique to derivation for this key wrapping process. This means that any future designs that use the same KDF must use a different value for the label. The label will be the 16 byte value consisting of the following ASCII characters:

```
ENHANCEDWRAP2010 (X'454E4841 4E434544 57524150 32303130')
```

The parameters for the counter mode KDF defined in NIST standard SP 800-108 are:

Fixed values:

h Length of output of PRF, 256 bits

r Length of the counter, in bits, 32. The counter will be an unsigned 4-byte value.

Inputs:

- KI (input key) - The key we are deriving from.
- Label - The value shown above (ASCII ENHANCEDWRAP2010).
- Separator byte - X'00' following the label value.
- Context - A null string. No context is used.

- L - The length of the derived key to be produced, rounded up to the next multiple of 256.
- PRF - HMAC-SHA256.

Variable length token (AESKW method)

The wrapping method for the variable-length key tokens with AESKW is defined in standard ANSI X9.102.

The wrapping of the payload of a variable length key (*K) using an AES *MK is defined as:

```
e*MK(*K) = eAESKW*MK(P)
P = ICV || Pad length || Hash length || Hash options || Data hash || *K || Padding
```

Where:

ICV Is the 6 byte constant X'A6A6A6A6A6A6'.

Pad length
Is the length of the padding in bits.

Hash length
Is the length of the Data Hash in bytes.

Hash options
Is a 4-byte field.

Data hash
Is the hash of the associated data block.

Padding
Is the number of bytes of X'00' used to make the overall length of P a multiple of 16.

eAESKW
Means encryption using the AESKW method.

Control vector

A unique control vector exists for each type of CCA key.

For an internal key token, the coprocessor XORs the master key with the control vector associated with the type of key the master key will encipher. The control vector ensures that an operational key is used only in cryptographic functions for which it is intended. For example, the control vector for an input PIN-encrypting key ensures that such a key can be used only in the Encrypted PIN Translate and Encrypted PIN Verify functions.

Types of keys

The cryptographic keys are grouped into the following categories based on the functions that they perform.

Symmetric keys master key (SYM-MK)
The SYM-MK master key is a triple-length (192-bit) key that is used only to encrypt other DES keys on the coprocessor. The administrator installs and changes the SYM-MK master key using the panel.exe utility, the clear key entry panels, the z/OS® clear key entry panels, or the optional Trusted Key Entry (TKE) workstation. The master key always remains within the secure boundary of the coprocessor. It is used only to encipher and decipher keys that are in operational form.

Note: If the coprocessor is shared with z/OS, the SYM-MK key must be a double-length (128-bit) key. This means that the first 64 bits and the last 64 bits of the key must be identical. If the master key is loaded by z/OS CCA or from a TKE workstation, it will automatically be a double-length key.

AES keys master key (AES-MK)
The AES-MK master key is a 256-bit key that is used to encrypt other AES keys and HMAC keys on the coprocessor. The administrator installs and changes the AES-MK master key using the panel.exe utility, the clear key entry panels, the z/OS clear key entry panels, or the optional Trusted Key Entry (TKE) workstation. The master key always remains within the secure boundary of the coprocessor. It is used only to encipher and decipher keys that are in operational form.

Asymmetric keys master key (ASYM-MK)
The ASYM-MK is a triple-length (192-bit) key that is used to protect RSA private keys on the coprocessor. The administrator installs and changes the ASYM-MK master key using the panel.exe utility, the clear key entry panels, the z/OS clear key entry panels, or the optional Trusted Key Entry (TKE) workstation. The master key always remains within the secure boundary of the coprocessor. It is used only to encipher and decipher keys that are in operational form.

AES CIPHER keys
The AES cipher keys are 128-bit, 192-bit, and 256-bit keys that protect data privacy. If you intend to use a cipher key for an extended period, you can store it in key storage so that it will be re-enciphered if the master key is changed.

AES PKA master key (APKA-MK)
The APKA-MK key, introduced to CCA beginning with Release 4.1.0, is used to encrypt and decrypt the Object Protection Key (OPK) that is itself used to wrap the key material of an Elliptic Curve Cryptography (ECC) key. ECC keys are asymmetric. The APKA-MK is a 256-bit (32-byte) value. The administrator installs and changes the APKA-MK master key using the panel.exe utility, the clear key entry panels, the z/OS clear key entry panels, or the optional Trusted Key Entry (TKE) workstation.

Data-encrypting keys
The data-encrypting keys are single-length DES (64-bit), double-length DES (128-bit), or triple-length DES (192-bit) keys, or 128-bit, 192-bit or 256-bit AES keys that protect data privacy. Single-length DES data-encrypting keys can also be used to encode and decode data and authenticate data sent in messages. If you intend to use a data-encrypting key for an extended period of time, you can store it in the CCA key storage file so that it will be re-enciphered if the master key is changed.

You can use single-length DES data-encrypting keys in the Encipher and Decipher verbs to manage data, and also in the MAC Generate and MAC Verify verbs. Double-length DES and triple-length DES data-encrypting keys can be used in the Encipher and Decipher verbs for more secure data privacy. DATAC is also a double-length DES data encrypting key.

AES data-encrypting keys can be used in services similar to DES data-encrypting key services.

DES CIPHER keys
These consist of CIPHER, ENCIPHER, and DECIPHER keys. They are single and double length DES keys for enciphering and deciphering data.

Cipher text translation keys
These ciphertext translation keys consist of CIPHERXI, CIPHERXL, and CIPHERXO keys. They protect data that is transmitted through intermediate systems when the originator and receiver do not share a common key. Data that is enciphered under one ciphertext translation key is re-enciphered under another ciphertext translation key on the intermediate node. During this process, the data never appears in the clear. These keys are double-length.

HMAC keys
HMAC keys are variable-length symmetric keys. The length is in the range of 80 - 2024. HMAC keys are used to generate and verify HMACs using the FIPS-198 algorithm, with the HMAC Generate and HMAC Verify verbs.

- Operational keys will be encrypted under the AES master key
- HMAC keys can be imported and exported under an RSA key.
- HMAC keys will be stored in the AES key storage file. The AES master key must be active.

For more information about HMAC keys and verb processing, see Chapter 8, "Verifying data integrity and authenticating messages," on page 369.

MAC keys
The MAC keys are single-length DES (64-bits - DATAM, DATAMV, MAC, and MACVER,) and double-length DES (128-bits - DATAM, DATAMV, MAC, and MACVER) keys used for the verbs that generate and verify MACs.

PIN keys
The personal identification number (PIN) is a basis for verifying the identity of a customer across financial industry networks. PIN keys are used in cryptographic functions to generate, translate, and verify PINs, and protect PIN blocks. They are all double-length DES (128 bits) keys. PIN keys are used in the Clear PIN Generate, Encrypted PIN Verify, and Encrypted PIN Translate verbs.

For installations that do not support double-length DES 128-bit keys, effective single-length DES keys are provided. For a single-length DES key, the left key half of the key equals the right key half.

"Processing personal identification numbers" on page 49 gives an overview of the PIN algorithms you need to know to write your own application programs.

AES transport keys (or key-encrypting keys)
Transport keys are also known as key-encrypting keys. They are used to protect AES and HMAC keys when you distribute them from one system to another.

There are two types of AES transport keys:

Exporter key-encrypting key
This type of key protects keys of any type that are sent from your system to another system. The exporter key at the originator is the same key as the importer key of the receiver.

Importer key-encrypting key
This type of key protects keys of any type that are sent from another system to your system. It also protects keys that you store

externally in a file that you can import to your system at another time. The importer key at the receiver is the same key as the exporter key at the originator.

DES transport keys (or key-encrypting keys)

Transport keys are also known as key-encrypting keys. They are double-length (128 bits) DES keys used to protect keys when you distribute them from one system to another.

There are several types of DES transport keys:

Exporter or OKEYXLAT key-encrypting key

This type of key protects keys of any type that are sent from your system to another system. The exporter key at the originator is the same key as the importer key of the receiver.

Importer or IKEYXLAT key-encrypting key

This type of key protects keys of any type that are sent from another system to your system. It also protects keys that you store externally in a file that you can import to your system later. The importer key at the receiver is the same key as the exporter key at the originator.

NOCV importers and exporters

These keys are key-encrypting keys used to exchange keys with systems that do not recognize key-encrypting key variants. There are some requirements and restrictions for the use of NOCV key-encrypting keys:

- The use of NOCV IMPORTERs and EXPORTERs is controlled by access control points in the coprocessor's role-based access control system.
- Only programs in system or supervisor state can use the NOCV key-encrypting key in the form of tokens in verbs. Any program can use NOCV key-encrypting keys with label names from the key storage.
- Access to NOCV key-encrypting keys should be carefully controlled, because use of these keys can reduce security in your key management process.
- NOCV key-encrypting key can be used to encrypt single or double length DES keys with standard CVs for key types DATA, DATAC, DATAM, DATAMV, DATAXLAT, EXPORTER, IKEYXLAT, IMPORTER, IPINENC, single-length MAC, single-length MACVER, OKEYXLAT, OPINENC, PINGEN and PINVER.
- NOCV key-encrypting keys can be used with triple length DATA keys. Because DATA keys have 0 CVs, processing will be the same as if the key-encrypting keys are standard key-encrypting keys (not the NOCV key-encrypting key).

Note: A key-encrypting key should be as strong or stronger than the key that it is wrapping.

You use key-encrypting keys to protect keys that are transported using any of the following verbs: Data Key Export, Key Export, Key Import, Clear Key Import, Multiple Clear Key Import, Key Generate, Key Generate2, Key Translate and Key Translate2.

For installations that do not support double-length key-encrypting keys, effective single-length keys are provided. For an effective single-length key, the clear key value of the left key half equals the clear key value of the right key half.

Key-generating keys
Key-generating keys are double-length keys used to derive other keys. This is often used in smart card applications.

Clear keys
A clear key is the base value of a key, and is not encrypted under another key. Encrypted keys are keys whose base value has been encrypted under another key.

To convert a clear key to an encrypted data key in operational form, use either the Clear Key Import verb or the Multiple Clear Key Import verb.

Table 4 describes the key types.

Table 4. Key types

Key type	Description
AESDATA	Data encrypting key. Use the AES 128-bit, 192-bit, or 256-bit key to encipher and decipher data.
AESTOKEN	Can contain an AES key.
CIPHER	**AES** This 128-bit, 192-bit, or 256-bit key is used to encrypt or decrypt data. It can be used in the Symmetric Algorithm Decipher and Symmetric Algorithm Encipher verbs. **DES** This single or double-length key is used to encrypt or decrypt data. It can be used in the Encipher and Decipher verbs Used only to encrypt or decrypt data. This is a single or double length key and can be used in the Encipher or Decipher verbs.
CIPHERXI	Usable with the Cipher Text Translate2 verb (translate inbound key only)
CIPHERXL	Usable with the Cipher Text Translate2 verb (translate inbound or outbound key)
CIPHERXO	Usable with the Cipher Text Translate2 verb (translate outbound key only)
CLRAES	Data encrypting key. The key value is not encrypted. Use this AES 128-bit, 192-bit, or 256-bit key to encipher and decipher data.
CVARDEC	The cryptographic variable decipher service, which is available in some CCA implementations, uses a **CVARDEC** key to decrypt plaintext by using the Cipher Block Chaining (CBC) method. This is a single-length key.
CVARENC	The cryptographic variable encipher service, which is available in some CCA implementations, uses a **CVARENC** key to encrypt plaintext by using the Cipher Block Chaining (CBC) method. This is a single-length key.
CVARPINE	Used to encrypt a PIN value for decryption in a PIN-printing application. This is a single-length key.
CVARXCVL	Used to encrypt special control values in DES key management. This is a single-length key.
CVARXCVR	Used to encrypt special control values in DES key management. This is a single-length key.
DATA	Data encrypting key. Use this DES single-length, double-length, or triple-length key to encipher and decipher data. Use the AES 128-bit, 192-bit, or 256-bit key to encipher and decipher data.
DATAC	Used to specify a **DATA**-class key that will perform in the Encipher and Decipher verbs, but not in the MAC Generate or MAC Verify verbs. This is a double-length key. Only available with a CEX*C.
DATAM	Key-encrypting keys that have a control vector with this attribute formerly could only be used to transport keys with a key type of **DATA**, **CIPHER**, ENCIPHER, **DECIPHER**, **MAC**, and **MACVER**. The meaning of this keyword has been discontinued and its usage is allowed for backward compatibility reasons only.

Table 4. Key types (continued)

Key type	Description
DATAMV	Used to specify a **DATA**-class key that performs in the MAC Verify verb, but not in the MAC Generate, Encipher, or Decipher verbs.
DATAXLAT	Data translation key. Use this single-length key to reencipher text from one DATA key to another.
DECIPHER	Used only to decrypt data. **DECIPHER** keys cannot be used in the Encipher (CSNBENC) verb. This is a single-length key. This is a single or double length key and can be used in the Decipher verb.
DKYGENKY	Used to generate a diversified key based on the key-generating key. This is a double-length key.
ENCIPHER	Used only to encrypt data. **ENCIPHER** keys cannot be used in the Decipher (CSNBDEC) verb. This is a single-length key. This is a single or double length key and can be used in the Encipher verb.
EXPORTER	Exporter key-encrypting key. Use this double-length DES key or 128-bit, 192-bit or 256-bit AES key to convert a key from operational form into exportable form.
HMAC	Variable-length HMAC generation key. Use this key to generate or verify a Message Authentication Code using the keyed-hash MAC algorithm.
HMACVER	Variable-length HMAC verification key. Use this key to verify a Message Authentication Code using the keyed-hash MAC algorithm.
IKEYXLAT	Used to decrypt an input key in the Key Translate and Key Translate2 verbs. This is a double-length key.
IMPORTER	Importer key-encrypting key. Exporter key-encrypting key. Use this double-length DES key or 128-bit, 192-bit or 256-bit AES key to convert a key from importable form into operational form.
IMP-PKA	Double-length limited-authority importer key used to encrypt PKA private key values in PKA external tokens.
IPINENC	Double-length input PIN-encrypting key. PIN blocks received from other nodes or automatic teller machine (ATM) terminals are encrypted under this type of key. These encrypted PIN blocks are the input to the Encrypted PIN Translate, Encrypted PIN Verify, and Clear PIN Generate Alternate verbs.
KEYGENKY	Used to generate a key based on the key-generating key. This is a double-length key.
MAC	Single, double-length, or variable-length MAC generation key. Use this key to generate a message authentication code.
MACVER	Single, double-length, or variable-length MAC verification key. Use this key to verify a message authentication code.
OKEYXLAT	Used to encrypt an output key in the Key Translate and Key Translate2 verbs. This is a double-length key.
OPINENC	Output PIN-encrypting key. Use this double-length output key to translate PINs. The output PIN blocks from the Encrypted PIN Translate, Encrypted PIN Generate, and Clear PIN Generate Alternate verbs are encrypted under this type of key.
PINGEN	PIN generation key. Use this double-length key to generate PINs.
PINVER	PIN verification key. Use this double-length key to verify PINs.
SECMSG	Used to encrypt PINs or keys in a secure message. This is a double-length key.
TOKEN	A key token that might contain a key.

Table 5 on page 42 lists key subtypes passed in the *rule_array* keyword.

Table 5. Key subtypes specified by the rule_array keyword

rule_array **keyword**	**Description**
AMEX-CSC	A MAC key that can be used for the AMEX CSC transaction validation process MAC calculation method, used with the Transaction Validation (CSNBTRV) verb.
ANSIX9.9	A MAC key that can be used for the ANSI X9.9 MAC calculation method, either for MAC Generate (CSNBMGN), MAC Verify (CSNBMVR), or Transaction Validation (CSNBTRV). Other Control Vector bits could limit these usages.
ANY	Key-encrypting keys that have a control vector with this attribute can be used to transport any type of key. The meaning of this keyword has been discontinued, and its usage is allowed for backward compatibility reasons only.
ANY-MAC	Can be used with any function or MAC calculation method that uses a MAC key, such as MAC Generate (CSNBMGN), MAC Verify (CSNBMVR), or Transaction Validation (CSNBTRV). This is the default configuration for a MAC key control vector.
CVVKEY-A	Can be used as 'Key A' in either the CVV Generate (CSNBCSG) or CVV Verify (CSNBCSV) verbs, as controlled by the CVV generation and verification Control Vector bits (bits 20 and 21 respectively).
CVVKEY-B	Can be used as 'Key B' in either the CVV Generate (CSNBCSG) or CVV Verify (CSNBCSV) verbs, as controlled by the CVV generation and verification Control Vector bits (bits 20 and 21 respectively).
DATA	Data encrypting key. Use this 8-byte, 16-byte or 24-byte DES key or 16-byte, 24-byte or 32-byte AES key to encipher and decipher data.
EPINGENA	Legacy key subtype, used to turn on bit 19 of a PIN Generating Key Control Vector. The default PIN Generating Key type will have this bit on. No PIN generating or processing behavior is currently influenced by this key subtype parameter. **EPINGENA** is no longer supported, although the bit retains this definition for compatibility There is no Encrypted Pin Generate Alternate verb
LMTD-KEK	Key-encrypting keys that have a control vector with this attribute formerly could only be used to exchange keys with key-encrypting keys that carry NOT-KEK, PIN, or DATA key-type ciphering restrictions. The usage of this keyword has been discontinued and its usage is allowed for backward compatibility reasons only.
NOT-KEK	Key-encrypting keys that have a control vector with this attribute formerly could not be used to transport key-encrypting keys. The meaning of this keyword has been discontinued and its usage is allowed for backward compatibility reasons only.
PIN	Key-encrypting keys that have a control vector with this attribute formerly could only be used to transport keys with a key type of **PINVER**, **IPINENC**, and **OPINENC**. The usage of this keyword has been discontinued and its usage is allowed for backward compatibility reasons only.

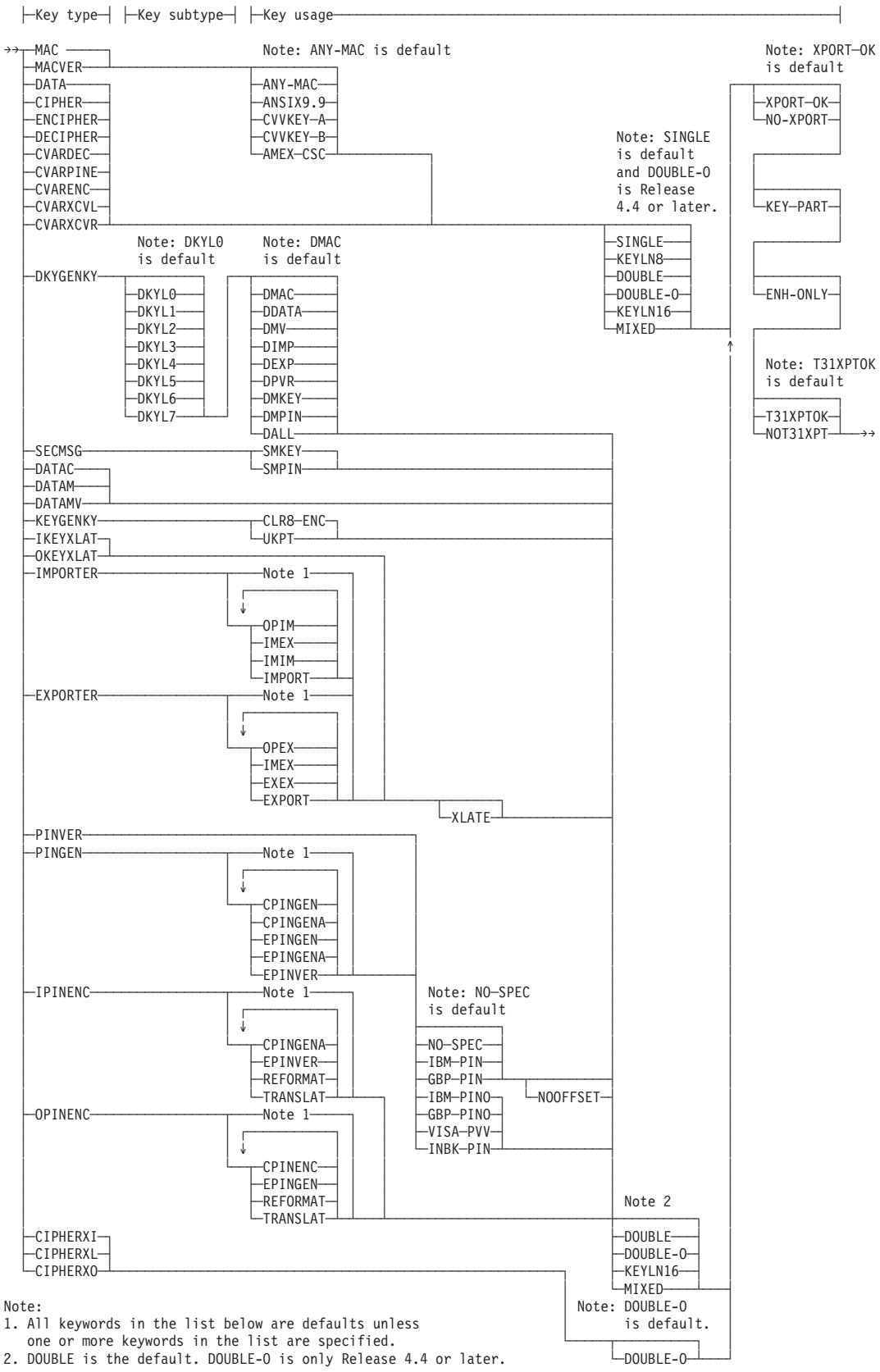


Figure 3. Control Vector Generate and Key Token Build CV keyword combinations for fixed-length DES key tokens

Verbs for managing AES and DES key storage files

CCA provides API functions to allow application programs to manage the AES and DES key storage file, where key tokens are stored when the program references them by key label name.

The following verbs are used to manage the AES and DES key storage files:

- AES Key Record Create (CSNBAKRC)
- AES Key Record Delete (CSNBAKRD)
- AES Key Record List (CSNBAKRL)
- AES Key Record Read (CSNBAKRR)
- AES Key Record Write (CSNBAKRW)
- DES Key Record Create (CSNBKRC)
- DES Key Record Delete (CSNBKRD)
- DES Key Record List (CSNBKRL)
- DES Key Record Read (CSNBKRR)
- DES Key Record Write (CSNBKRW)

Verbs for managing the PKA key storage file and PKA keys in the cryptographic engine

The PKA key storage file is a repository for RSA keys, similar to the AES and DES key storage files.

An application can store keys in the key storage file and refer to them by label when using any of the verbs which accept RSA key tokens as input. The following verbs are used to manage the PKA key storage file, or PKA keys stored in the cryptographic engine:

- PKA Key Record Create (CSNDKRC)
- PKA Key Record Delete (CSNDKRD)
- PKA Key Record List (CSNDKRL)
- PKA Key Record Read (CSNDKRR)
- PKA Key Record Write (CSNDKRW)
- Retained Key Delete (CSNDRKD)
- Retained Key List (CSNDRKL)

EC Diffie-Hellman key agreement models

You can specify an EC Diffie-Hellman agreement model, or obtain the shared secret value without deriving a key.

Token agreement scheme

The caller must have both the required key tokens and both party's identifiers, including a randomly generated nonce.

Combine the exchanged nonce and Party Info into the party identifier. (Both parties must combine this information in the same format.) Then call the EC Diffie-Hellman verb, where “EC” means Elliptic Curve. Specify a skeleton token or the label of a skeleton token as the output key identifier to be used as a container

for the computed symmetric key material. Note, both parties must specify the same key type in their skeleton key tokens.

- Specify rule-array keyword **DERIV01** to denote the Static Unified Model key agreement scheme.
- Specify an ECC token as the private key identifier containing this party's ECC public-private key pair.
- Optionally specify a private KEK key identifier, if the key pair is in an external key token.
- Specify an ECC token as the public key identifier containing the other party's ECC public key part.
- Specify a skeleton token as the output key identifier to be used as a container for the computed symmetric key material.
- Optionally specify an output KEK key identifier, if the output key is to be in an external key token.
- Specify the combined party info (including nonce) as the party identifier.
- Specify the desired size of the key to be derived (in bits) as the key bit length.

Obtaining the raw "Z" value

The caller must then derive the final key material using a method of their choice. Do not specify any party info.

- Specify rule array keyword **PASSTHRU** to denote no key agreement scheme.
- Specify an ECC token as the private key identifier containing this party's ECC public-private key pair.
- Optionally specify a private KEK key identifier, if the key pair is in an external key token.
- Specify an ECC token as the public key identifier containing the other party's ECC public key part.
- The output key identifier will be populated with the resulting shared secret material.

Improved remote key distribution

New methods have been added for securely transferring symmetric encryption keys to remote devices, such as Automated Teller Machines (ATMs), PIN-entry devices, and point of sale terminals.

These methods can also be used to transfer symmetric keys to another cryptographic system of any type, such as a different kind of Hardware Security Module (HSM) in an IBM or non-IBM computer server.

Note: This improved remote key distribute support is only available on IBM z9® and later.

This change replaces expensive human operations with network transactions that can be processed quickly and inexpensively. This method makes significant interoperability improvements to related cryptographic key-management functions.

In “Remote key loading” on page 46, an ATM scenario illustrates the operation of the new methods. Other uses of this method are also possible.

Remote key loading

Remote key loading is the process of installing symmetric encryption keys into a remotely located device from a central administrative site.

This encompasses two phases of key distributions:

- Distribution of initial key encrypting keys (KEKs) to a newly installed device. A KEK is a type of symmetric encryption key that is used to encrypt other keys so that they can be securely transmitted over unprotected paths.
- Distribution of operational keys or replacement KEKs, enciphered under a KEK currently installed in the device.

Access control points are assigned to roles to control keyword usage in the services provided for ATM remote key loading. Table 6 lists the access control points used by the ATM remote key loading function.

Table 6. Access Control Points Used by ATM remote key loading

Verb name	Entry point	Offset	Access Control Point name and comments
Trusted Block Create	CSNDTBC	X'030F'	Trusted Block Create - Create Block in inactive form
Trusted Block Create	CSNDTBC	X'0310'	Trusted Block Create - Activate an inactive block
PKA Key Import	CSNDPKI	X'0311'	PKA Key Import - Import an external trusted block Convert a trusted block from external to internal format
PKA Key Import	CSNDPKI	X'0104'	PKA Key Import
Remote Key Export	CSNDRKX	X'0312'	Remote Key Export - Gen or export a non-CCA node key
Key Generate Remote Key Export	CSNBKGN CSNDRKX	X'00DB'	Key Generate - SINGLE-R Replication of a single-length source key (which is either an RKX token or a CCA token) if the output symmetric encryption result is to be a CCA token, and the CV in the trusted block's Common Export Key Parameters TLV Object is 16 bytes with key form bits 'fff' set to X'010' for the left half and X'001' for the right half.
Key Import	CSNBKIM	X'027B'	Key Import - Unrestricted The importer key identifier in the initial code release must have unique halves.
Key Export	CSNBKEX	X'0276'	Key Export - Unrestricted The transport key identifier in the initial code release must have unique halves.

Old remote key loading example

Use an ATM as an example of the remote key loading process: a new ATM has none of the purchaser's keys installed when it is delivered from the manufacturer.

The process of getting the first key securely loaded is difficult.

The installation of the first key on the ATM has typically been done by loading the first KEK into each ATM manually, in multiple clear text key parts. Using dual control for key parts, two separate people must carry key part values to the ATM, then load each key part manually. After they are inside the ATM, the key parts are combined to form the actual KEK. In this manner, neither of the two people has

the entire key, protecting the key value from disclosure or misuse. This method is labor-intensive and error-prone, making it expensive.

New remote key loading methods

New remote key loading methods have been developed to overcome some of the shortcomings of the old manual key loading methods.

These new methods define acceptable techniques using public key cryptography to load keys remotely. Using these new methods, initial KEKs can be loaded without sending people to the remote device. This will reduce labor costs, be more reliable, and be much less expensive to install and change keys.

The new cryptographic features provide new methods for the creation and use of the special key forms needed for remote key distribution of this type. In addition, the new cryptographic features provide ways to solve long-standing barriers to secure key exchange with non-IBM cryptographic systems.

After an ATM is in operation, new keys can be installed as needed, by sending them enciphered under a KEK installed previously. This is straightforward in concept, but the cryptographic architecture in ATMs is often different from that of the host system that is sending the keys, and it is difficult to export the keys in a form understood by the ATM. For example, cryptographic architectures often enforce key-usage restrictions in which a key is bound to data describing limitations on how it can be used (for encrypting data, for encrypting keys, for operating on Message Authentication Codes (MACs), and so forth). The encoding of these restrictions and the method used to bind them to the key itself differs among cryptographic architectures, and it is often necessary to translate the format to that understood by the target device prior to a key being transmitted. It is difficult to do this without reducing security in the system; typically it is done by making it possible to arbitrarily change key-usage restrictions.

The methods described here provide a mechanism through which the system owner can securely control these translations, preventing the majority of attacks that could be mounted by modifying usage restrictions.

A data structure called a *trusted block* is defined to facilitate the remote key loading methods. The trusted block is the primary vehicle supporting these new methods. See "Trusted blocks" on page 877.

Verbs supporting Secure Sockets Layer (SSL)

The Secure Sockets Layer (SSL) protocol, developed by Netscape Development Corporation, provides communications privacy over the Internet. Client/server applications can use the SSL protocol to provide secure communications and prevent eavesdropping, tampering, or message forgery.

CCA provides verbs that support the RSA-encryption and RSA-decryption of PKCS 1.2-formatted symmetric key data to produce symmetric session keys. These session keys can then be used to establish an SSL session between the sender and receiver. The verbs provide SSL support:

- PKA Decrypt (CSNDPKD)
- PKA Encrypt (CSNDPKE)

Enciphering and deciphering data

To protect data, CCA can use the Data Encryption Standard (DES) or Advanced Encryption Standard (AES) algorithms to encipher or decipher data or keys.

Enciphering data protects it from disclosure to people who do not have authority to access it. Using algorithms that make it difficult and expensive for an unauthorized user to derive the original clear data within a practical time period assures privacy.

The DES algorithm is documented in the Federal Information Processing Standard #46. The AES algorithm is documented in the Federal Information Processing Standard #192. These verbs perform the enciphering and deciphering functions:

- Decipher (CSNBDEC)
- Encipher (CSNBENC)
- Symmetric Algorithm Decipher (CSNBSAD)
- Symmetric Algorithm Encipher (CSNBSAE)

Managing data integrity and message authentication

To ensure the integrity of transmitted messages and stored data, CCA provides DES-based Message Authentication Code (MAC) functions and several hashing functions, including Modification Detection Code (MDC), SHA-1, RIPEMD-160 and MD5.

See Chapter 12, "Using digital signatures," on page 625 for an alternate method of message authentication using digital signatures.

The choice of verb depends on the security requirements of the environment in which you are operating. If you need to ensure the authenticity of the sender and also the integrity of the data, consider Message Authentication Code processing. If you need to ensure the integrity of transmitted data in an environment where it is not possible for the sender and the receiver to share a secret cryptographic key, consider hashing functions.

Processing message authentication code

The process of verifying the integrity and authenticity of transmitted messages is called *message authentication*.

Message authentication code (MAC) processing allows you to verify that a message was not altered or a message was not fraudulently introduced onto the system. You can check that a message you have received is the same one sent by the message originator. The message itself can be in clear or encrypted form. The comparison is performed within the cryptographic coprocessor. Because both the sender and receiver share a secret cryptographic key used in the MAC calculation, the MAC comparison also ensures the authenticity of the message.

In a similar manner, MACs can be used to ensure the integrity of data stored on the system or on removable media, such as tape.

CCA key typing makes it possible to give one party a key that can only be used to generate a MAC, and to give another party a corresponding key that can only be used to verify the MAC. This ensures that the second party cannot impersonate the first by generating MACs with their version of the key.

The coprocessor provides support for both single-length and double-length MAC generation and MAC verification keys. With the ANSI X9.9-1 single key algorithm, use the single-length MAC and MACVER keys.

CCA provides support for the use of data-encrypting keys in the MAC Generate and MAC Verify verbs, and also the use of a MAC generation key in the MAC Verify verb. This support permits CCA MAC verbs to interface more smoothly with non-CCA key distribution system.

HMAC codes are computed using the FIPS-198 Keyed-Hash Message Authentication Code method. See Chapter 8, "Verifying data integrity and authenticating messages," on page 369.

These verbs are used to process MACs:

- MAC Generate (CSNBMGN)
- MAC Verify (CSNBMVR)

Hashing functions

Hashing functions are provided by two verbs.

These verbs are:

- MDC Generate (CSNBMDG)
- One-Way Hash (CSNBOWH)

Processing personal identification numbers

The process of validating personal identities in a financial transaction system is called *personal authentication*.

The personal identification number (PIN) is the basis for verifying the identity of a customer across the financial industry networks. The financial industry needs functions to generate, translate, and verify PINs. These functions prevent unauthorized disclosures when organizations handle personal identification numbers.

The coprocessor supports the following algorithms for generating and verifying personal identification numbers:

- IBM 3624
- IBM 3624 PIN offset
- IBM German Bank Pool
- IBM German Bank Pool PIN Offset (GBP-PINO)
- VISA PIN validation value
- Interbank

You can translate PIN blocks from one format to another without the PIN being exposed in cleartext form. The coprocessor supports the following formats:

- ANSI X9.8
- ISO formats 0, 1, 2, 3
- VISA formats 1, 2, 3, 4
- IBM 4704 Encrypting PINPAD format
- IBM 3624 formats

- IBM 3621 formats
- ECI formats 1, 2, 3

With the capability to translate personal identification numbers into different PIN block formats, you can use personal identification numbers on different systems.

Secure messaging

The following verbs assist applications in encrypting secret information such as clear keys and PIN blocks in a secure message.

These verbs execute within the secure boundary of the cryptographic coprocessor:

- Secure Messaging for Keys (CSNBSKY)
- Secure Messaging for PINs (CSNBSPN)

Trusted Key Entry support

The Trusted Key Entry (TKE) workstation provides a secure method of initializing and administering cryptographic coprocessors.

It is an optional z Systems feature, but it is mandatory if z/OS and CCA are not available on your system. Initialization of the coprocessor can be done through CCA for both the z/OS and Linux environments, either with or without TKE.

TKE Version 6.0 or higher is required in order to administer the CEX*C coprocessor features earlier than CEX5C. You can use the TKE workstation to load DES master keys, PKA master keys, and operational keys in a secure way. TKE Version 6.0 and 7.0 can also set AES master keys on a CEX*C coprocessor earlier than CEX5C. For CEX5C, TKE 8.0 is required.

You can load keys remotely and for multiple coprocessors, which can be in a single machine or in multiple machines. The TKE workstation eases the administration for using one coprocessor as a production machine and as a test machine at the same time, while maintaining security and reliability.

The TKE workstation can be used for enabling and disabling access control points for verbs executed on the cryptographic coprocessor. See Chapter 22, “Access control points and verbs,” on page 953 for additional information.

For complete details about the TKE workstation, see *z/OS Cryptographic Services ICSF: Trusted Key Entry PCIX Workstation User's Guide*.

Typical sequences of CCA verbs

View some sample sequences in which CCA verbs might be called.

Combination A (DATA keys only)

1. Random number generate
2. Clear key import or multiple clear key import
3. Encipher/decipher
4. Data key export or key export (optional step)

Combination B

1. Random number generate

2. Secure key import or multiple secure key import
3. Any service
4. Data key export for DATA keys, or key export in the general case (optional step)

Combination C

1. Key generate (OP form only)
2. Any service
3. Key export (optional)

Combination D

1. Key generate (OPEX form)
2. Any service

Combination E

1. Key Generate (IM form only)
2. Key Import
3. Any service
4. Key Export (optional)

Combination F

1. Key Generate (IMEX form)
2. Key Import
3. Any service

Combination G

1. Key Generate
2. AES or DES Key Record Create
3. AES or DES Key Record Write
4. Any service (passing label of the key just generated)

Combination H

1. Key Import
2. AES or DES Key Record Create
3. AES or DES Key Record Write
4. Any service (passing label of the key just generated)

Notes

1. An example of "any service" is CSNBENC.
2. These combinations exclude verbs that can be used on their own; for example, Key Export or encode, or using the Key Generate verb to generate an exportable key.
3. These combinations do not show key communication, or the transmission of any output from an CCA verb.

The key forms are described in Chapter 18, "Key forms and types used in the Key Generate verb," on page 893 and "Key Generate (CSNBKGN)" on page 171.

Using the CCA node and master key management verbs

Use these verbs for the CCA node and master key management functions.

- Cryptographic Facility Query (CSUACFQ)
- Cryptographic Facility Version (CSUACFV)
- Cryptographic Resource Allocate (CSUACRA)
- Cryptographic Resource Deallocate (CSUACRD)
- Cryptographic Variable Encipher (CSNBCVE)
- Data Key Export (CSNBDKX)
- Data Key Import (CSNBDKM)
- Diversified Key Generate (CSNBDKG)
- EC Diffie-Hellman (CSNDEDH)
- Key Export (CSNBKEX)
- Key Generate (CSNBKGN)
- Key Import (CSNBKIM)
- Key Part Import (CSNBKPI)
- Key Storage Initialization (CSNBKSI)
- Key Test (CSNBKYT)
- Key Test Extended (CSNBKYTX)
- Key Token Build (CSNBKTB)
- Key Token Change (CSNBKTC)
- Key Token Parse (CSNBKTP)
- Key Token Parse2 (CSNBKTP2)
- Key Translate (CSNBKTR)
- Master Key Process (CSNBMKP)
- Multiple Clear Key Import (CSNBCKM)
- Prohibit Export (CSNBPEX)
- Prohibit Export Extended (CSNBPEXX)
- Random Number Generate (CSNBRNG)
- Random Number Generate Long (CSNBRNGL)
- Random Number Tests (CSUARNT)
- Symmetric Key Export (CSNDSYX)
- Symmetric Key Generate (CSNDSYG)
- Symmetric Key Import (CSNDSYI)
- Symmetric Key Import2 (CSNDSYI)

Summary of the CCA nodes and resource control verbs

A table of the CCA nodes and resource control verbs, including references to the verb descriptions.

Table 7. Summary of CCA nodes and resource control verbs

Entry point	Verb name	Description	Topic/Page
Chapter 5, "Using the CCA nodes and resource control verbs," on page 77			

Table 7. Summary of CCA nodes and resource control verbs (continued)

Entry point	Verb name	Description	Topic/Page
CSUACFQ	Cryptographic Facility Query	Retrieves information about the coprocessor and the CCA application program in that coprocessor.	"Cryptographic Facility Query (CSUACFQ)" on page 77
CSUACFV	Cryptographic Facility Version	Retrieve the Security Application Program Interface (SAPI) version and build date.	"Cryptographic Facility Version (CSUACFV)" on page 108
CSUACRA	Cryptographic Resource Allocate	Allocates specific CCA coprocessor for use by the thread or process, depending on the scope of the verb.	"Cryptographic Resource Allocate (CSUACRA)" on page 109
CSUACRD	Cryptographic Resource Deallocate	De-allocates a specific CCA coprocessor that is allocated by the thread or process, depending on the scope of the verb.	"Cryptographic Resource Deallocate (CSUACRD)" on page 112
CSNBKSI	Key Storage Initialization	This verb initializes a key-storage file using the current symmetric or asymmetric master-key. The initialized key storage does not contain any preexisting key records. The name and path of the key storage data and index file are established differently in each operating environment. Note that HMAC keys are not supported for key storage.	"Key Storage Initialization (CSNBKSI)" on page 114
CSUALGQ	Log Query	This verb retrieves system log (syslog) message data and CCA log message data from the coprocessor. Syslog data is available for one of the five latest boot cycles (current boot cycle and up to four previous boot cycles). CCA log message data is optionally available during the current boot cycle. The verb supports service personnel and developers in testing and debugging issues.	"Log Query (CSUALGQ)" on page 117
CSNBMKP	Master Key Process	Operates on the three master-key registers: new, current, and old. This verb is used to clear the new and the old master-key registers, generate a random master-key value in the new master-key register, XOR a clear value as a key part into the new master-key register, and set the master key, which transfers the current master-key to the old master-key register and the new master-key to the current master-key register.	"Master Key Process (CSNBMKP)" on page 122
CSUARNT	Random Number Tests	Invokes the USA NIST FIPS PUB 140-1 specified cryptographic operational tests. These tests, selected by a *rule_array* keyword, consist of known-answer tests of DES, RSA, and SHA-1 processes and, for random numbers, monobit test, poker test, runs test, and log-run test.	"Random Number Tests (CSUARNT)" on page 127

Summary of the AES, DES, and HMAC verbs

Hash Message Authentication Code (HMAC) support was added in CCA Release 4.1.0. All of the HMAC verbs and features listed in this summary require CCA 4.1.0 or CCA 4.2.0 in order to run.

Table 8 lists the AES, DES, and HMAC verbs described in this document. The table also references the chapter that describes the verb.

Table 8. Summary of CCA AES, DES, and HMAC verbs

Entry point	Verb name	Description	Topic/Page
Chapter 6, "Managing AES and DES cryptographic keys," on page 129			
CSNBCKI	Clear Key Import	Imports an 8-byte clear DATA key, enciphers it under the master key, and places the result into an internal key token. This verb converts the clear key into operational form as a DATA key.	"Clear Key Import (CSNBCKI)" on page 130
CSNBCKM	Multiple Clear Key Import	Imports a single-length, double-length, or triple-length clear DATA key that is used to encipher or decipher data. It accepts a clear key and enciphers the key under the host master key, returning an encrypted DATA key in operational form in an internal key token.	"Multiple Clear Key Import (CSNBCKM)" on page 131
CSNBCVG	Control Vector Generate	Builds a control vector from keywords specified by the *key_type* and *rule_array* parameters.	"Control Vector Generate (CSNBCVG)" on page 134
CSNBCVT	Control Vector Translate	Changes the control vector used to encipher an external DES key.	"Control Vector Translate (CSNBCVT)" on page 136
CSNBCVE	Cryptographic Variable Encipher	Encrypts plaintext using a CVARENC key to produce ciphertext using the Cipher Block Chaining (CBC) method.	"Cryptographic Variable Encipher (CSNBCVE)" on page 140
CSNBDKX	Data Key Export	Re-enciphers a DATA key from encryption under the master key to encryption under an exporter key-encrypting key, making it suitable for export to another system.	"Data Key Export (CSNBDKX)" on page 142
CSNBDKM	Data Key Import	Imports an encrypted source DES single-length or double-length DATA key and creates or updates a target internal key token with the master key enciphered source key.	"Data Key Import (CSNBDKM)" on page 143
CSNBDKG	Diversified Key Generate	Generates a key based upon the key-generating key, the processing method, and the parameter data that is supplied. The control vector of the key-generating key also determines the type of target key that can be generated.	"Diversified Key Generate (CSNBDKG)" on page 145
CSNBDKG2	Diversified Key Generate2	Generates an AES key based on a function of a key-generating key, the process rule, and data that you supply.	"Diversified Key Generate2 (CSNBDKG2)" on page 152

Table 8. Summary of CCA AES, DES, and HMAC verbs (continued)

Entry point	Verb name	Description	Topic/Page
CSNDEDH	EC Diffie-Hellman	Creates symmetric key material from a pair of Elliptic Curve Cryptography (ECC) keys using the Elliptic Curve Diffie-Hellman (ECDH) protocol.	"EC Diffie-Hellman (CSNDEDH)" on page 158
CSNBKEX	Key Export	Re-enciphers a key from encryption under a master key variant to encryption under the same variant of an exporter key-encrypting key, making it suitable for export to another system.	"Key Export (CSNBKEX)" on page 169
CSNBKGN	Key Generate	Generates a 64-bit, 128-bit, 192-bit, or 256-bit odd parity key, or a pair of keys; and returns them in encrypted forms (operational, exportable, or importable). Key Generate does not produce keys in plaintext.	"Key Generate (CSNBKGN)" on page 171
CSNBKGN2	Key Generate2	Generates either one or two HMAC keys. This verb does not produce keys in clear form and all keys are returned in encrypted form. When two keys are generated, each key has the same clear value, although this clear value is not exposed outside the secure cryptographic feature. This verb returns variable-length CCA key tokens and uses the AESKW wrapping method. Operational keys will be encrypted under the AES master key.	"Key Generate2 (CSNBKGN2)" on page 181
CSNBKIM	Key Import	Re-enciphers a key from encryption under an importer key-encrypting key to encryption under the master key. The re-enciphered key is in the operational form.	"Key Import (CSNBKIM)" on page 189
CSNBKPI	Key Part Import	Combines the clear key parts of any key type and returns the combined key value in an internal key token or an update to the CCA key storage file.	"Key Part Import (CSNBKPI)" on page 192
CSNBKPI2	Key Part Import2	Combines the clear key parts of any HMAC key type from an internal variable-length symmetric key-token, and returns the combined key value in an internal variable-length symmetric key-token or an update to the CCA key storage file.	"Key Part Import2 (CSNBKPI2)" on page 196
CSNBKYT	Key Test	Generates or verifies (depending on keywords in the *rule_array*) a secure verification pattern for keys. This verb requires the tested key to be in the clear or encrypted under the master key.	"Key Test (CSNBKYT)" on page 199

Table 8. Summary of CCA AES, DES, and HMAC verbs (continued)

Entry point	Verb name	Description	Topic/Page
CSNBKYT2	Key Test2	Generates or verifies (depending on keywords in the *rule_array*) a secure cryptographic verification pattern for keys contained in a variable-length symmetric key-token. The key to test can be in the clear or encrypted under a master key. Requires the tested key to be in the clear or encrypted under the master key.	"Key Test2 (CSNBKYT2)" on page 204
CSNBKYTX	Key Test Extended	This verb is essentially the same as Key Test, except for the following: • In addition to operating on internal keys and key parts, this verb also operates on external keys and key parts. • This verb does not operate on clear keys, and does not accept *rule_array* keywords CLR-A128, CLR-A192, CLR-A256, KEY-CLR, and KEY-CLRD.	"Key Test Extended (CSNBKYTX)" on page 208
CSNBKTB	Key Token Build	Builds an internal or external token from the supplied parameters. You can use this verb to build CCA key tokens for all key types that CCA supports. The resulting token can be used as input to the Key Generate, and Key Part Import verbs.	"Key Token Build (CSNBKTB)" on page 213
CSNBKTB2	Key Token Build2	Builds variable-length internal or external key tokens for all key types that the coprocessor supports. The key token is built based on parameters that you supply. The resulting token can be used as input to the Key Generate2, and Key Part Import2 verbs. A clear key token built by this verb can be used as input to the Key Test2 verb. This verb supports internal HMAC tokens, both as clear key tokens and as skeleton tokens containing no key.	"Key Token Build2 (CSNBKTB2)" on page 218
CSNBKTC	Key Token Change	Re-enciphers a DES key from encryption under the old master key to encryption under the current master key, and to update the keys in internal DES key-tokens.	"Key Token Change (CSNBKTC)" on page 254
CSNBKTC2	Key Token Change2	Re-enciphers a variable-length HMAC key from encryption under the old master key to encryption under the current master key. This verb also updates the keys in internal HMAC key-tokens.	"Key Token Change2 (CSNBKTC2)" on page 258

Table 8. Summary of CCA AES, DES, and HMAC verbs (continued)

Entry point	Verb name	Description	Topic/Page
CSNBKTP	Key Token Parse	Disassembles a key token into separate pieces of information. This verb can disassemble an external key-token or an internal key-token in application storage.	"Key Token Parse (CSNBKTP)" on page 260
CSNBKTP2	Key Token Parse2	Disassembles a variable-length symmetric key-token into separate pieces of information. The verb can disassemble an external or internal variable-length symmetric key-token in application storage. The verb returns some of the key-token information in a set of variables identified by individual parameters, and returns the remaining information as keywords in the rule array.	"Key Token Parse2 (CSNBKTP2)" on page 264
CSNBKTR	Key Translate	Uses one key-encrypting key to decipher an input key and then enciphers this key using another key-encrypting key within the secure environment.	"Key Translate (CSNBKTR)" on page 274
CSNBKTR2	Key Translate2	Uses one key-encrypting key to decipher an input key and then enciphers this key using another key-encrypting key within the secure environment. This verb differs from the Key Translate verb in that Key Translate2 can process both fixed-length and variable-length symmetric key tokens.	"Key Translate2 (CSNBKTR2)" on page 276
CSNDPKD	PKA Decrypt	Uses an RSA private key to decrypt the RSA-encrypted key value and return the clear key value to the application.	"PKA Decrypt (CSNDPKD)" on page 280
CSNDPKE	PKA Encrypt	Encrypts a supplied clear key value under an RSA public key. The supplied key can be formatted using the PKCS 1.2 or ZERO-PAD methods prior to encryption.	"PKA Encrypt (CSNDPKE)" on page 283
CSNBPEX	Prohibit Export	Modifies the control vector of a CCA key token so that the key cannot be exported. This verb operates only on internal key tokens.	"Prohibit Export (CSNBPEX)" on page 287
CSNBPEXX	Prohibit Export Extended	Modifies an external DES key-token so that the key can no longer be exported after it has been imported. This verb operates only on internal key tokens.	"Prohibit Export Extended (CSNBPEXX)" on page 288
CSNBRKA	Restrict Key Attribute	Modifies an operational variable-length key so that it cannot be exported.	"Restrict Key Attribute (CSNBRKA)" on page 290

Table 8. Summary of CCA AES, DES, and HMAC verbs (continued)

Entry point	Verb name	Description	Topic/Page
CSNBRNG	Random Number Generate	Generates an 8-byte cryptographic-quality random number suitable for use as an encryption key or for other purposes. The output can be specified in three forms of parity: RANDOM, ODD, and EVEN.	"Random Number Generate (CSNBRNG)" on page 294
CSNBRNGL	Random Number Generate Long	Generates a cryptographic-quality random number suitable for use as an encryption key or for other purposes, ranging from 1 - 8192 bytes in length. The output can be specified in three forms of parity: RANDOM, ODD, and EVEN.	"Random Number Generate Long (CSNBRNGL)" on page 295
CSNDSYX	Symmetric Key Export	Transfer an application-supplied symmetric key (a DATA key) from encryption under the AES, DES or HMAC master key to encryption under an application-supplied RSA public key. The application-supplied DATA key must be an AES, DES or HMAC internal key token, or the label of an AES or DES key token in the CCA key storage file. The Symmetric Key Import and Symmetric Key Import2 verb can import the PKA-encrypted key form at the receiving node. Support for HMAC key was added beginning with CCA 4.1.0.	"Symmetric Key Export (CSNDSYX)" on page 298
CSNDSXD	Symmetric Key Export with Data	Export a symmetric key, along with some application supplied data, encrypted using an RSA key.	"Symmetric Key Export with Data (CSNDSXD)" on page 303
CSNDSYG	Symmetric Key Generate	Generate a symmetric key (a DATA key) and return the key in two forms: DES-encrypted and encrypted under an RSA public key. The DES-encrypted key can be an internal token encrypted under a host DES master key, or an external form encrypted under a KEK. (You can use the Symmetric Key Import verb to import the PKA-encrypted form.)	"Symmetric Key Generate (CSNDSYG)" on page 307
CSNDSYI	Symmetric Key Import	Import a symmetric AES or DES DATA key enciphered under an RSA public key into operational form enciphered under a DES master key.	"Symmetric Key Import (CSNDSYI)" on page 312

Table 8. Summary of CCA AES, DES, and HMAC verbs (continued)

Entry point	Verb name	Description	Topic/Page
CSNDSYI2	Symmetric Key Import2	Use this verb to import an HMAC key that has been previously formatted and enciphered under an RSA public key by the Symmetric Key Export verb. The formatted and RSA-enciphered key is contained in an external variable-length symmetric key-token. The key is deciphered using the associated RSA private-key. The recovered HMAC key is re-enciphered under the AES master-key. The re-enciphered key is then returned in an internal variable-length symmetric key-token. The key algorithm for this verb is HMAC.	"Symmetric Key Import2 (CSNDSYI2)" on page 316
CSNBUKD	Unique Key Derive	Performs the key derivation process as defined in ANSI X9.24 Part 1. The process derives keys from two values: the base derivation key and the derivation data. Rule array keywords determine the types and number of keys derived on a particular call.	
Chapter 7, "Protecting data," on page 333			
CSNBDEC	Decipher	Deciphers data using cipher block chaining mode of DES. The result is called plaintext.	"Decipher (CSNBDEC)" on page 335
CSNBENC	Encipher	Enciphers data using the cipher block chaining mode of DES. The result is called ciphertext.	"Encipher (CSNBENC)" on page 339
CSNBSAD	Symmetric Algorithm Decipher	Deciphers data using the AES cipher block chaining mode.	"Symmetric Algorithm Decipher (CSNBSAD)" on page 344
CSNBSAE	Symmetric Algorithm Encipher	Enciphers data using the AES cipher block chaining mode	"Symmetric Algorithm Encipher (CSNBSAE)" on page 350
CSNBCTT2	Cipher Text Translate2	Deciphers encrypted data (ciphertext) under one ciphertext translation key and re-enciphers it under another ciphertext translation key without having the data appear in the clear outside the cryptographic coprocessor.	"Cipher Text Translate2 (CSNBCTT2)" on page 356
Chapter 8, "Verifying data integrity and authenticating messages," on page 369			
CSNBHMG	HMAC Generate	Generates a keyed hash message authentication code (HMAC) for the text string provided as input. See Chapter 8, "Verifying data integrity and authenticating messages," on page 369.	"HMAC Generate (CSNBHMG)" on page 371
CSNBHMV	HMAC Verify	Verifies a keyed hash message authentication code (HMAC) for the text string provided as input. See Chapter 8, "Verifying data integrity and authenticating messages," on page 369.	"HMAC Verify (CSNBHMV)" on page 374

Table 8. Summary of CCA AES, DES, and HMAC verbs (continued)

Entry point	Verb name	Description	Topic/Page
CSNBMGN	MAC Generate	Generates a 4, 6, or 8-byte Message Authentication Code (MAC) for a text string that the application program supplies. The MAC is computed using either the ANSI X9.9-1 algorithm or the ANSI X9.19 optional double key algorithm and padding could be applied according to the EMV specification.	"MAC Generate (CSNBMGN)" on page 378
CSNBMGN2	MAC Generate2	Generates a keyed hash message authentication code (HMAC) or a ciphered message authentication code (CMAC) for the message string provided as input. A MAC key with key usage that can be used for generate is required to calculate the MAC.	"MAC Generate2 (CSNBMGN2)" on page 382
CSNBMVR	MAC Verify	Verifies a 4, 6, or 8-byte Message Authentication Code (MAC) for a text string that the application program supplies. The MAC is computed using either the ANSI X9.9-1 algorithm or the ANSI X9.19 optional double key algorithm and padding could be applied according to the EMV specification. The computed MAC is compared with a user-supplied MAC.	"MAC Verify (CSNBMVR)" on page 386
CSNBMVR2	MAC Verify2	Verifies a keyed hash message authentication code (HMAC) or a ciphered message authentication code (CMAC) for the message text provided as input. A MAC key with key usage that can be used for verify is required to verify the MAC.	"MAC Verify2 (CSNBMVR2)" on page 390
CSNBMDG	MDC Generate	Creates a 128-bit hash value (Modification Detection Code) on a data string whose integrity you intend to confirm.	"MDC Generate (CSNBMDG)" on page 393
CSNBOWH	One-Way Hash	Generates a one-way hash on specified text.	"One-Way Hash (CSNBOWH)" on page 397
Chapter 10, "Financial services," on page 443			
CSNBAPG	Authentication Parameter Generate	Generates an authentication parameter (AP) and returns it encrypted using the key supplied in an input parameter.	"Authentication Parameter Generate (CSNBAPG)" on page 458
CSNBCPE	Clear PIN Encrypt	Formats a PIN into a PIN block format and encrypts the results. You can also use this verb to create an encrypted PIN block for transmission. With the RANDOM keyword, you can have the verb generate random PIN numbers.	"Clear PIN Encrypt (CSNBCPE)" on page 461

Table 8. Summary of CCA AES, DES, and HMAC verbs (continued)

Entry point	Verb name	Description	Topic/Page
CSNBPGN	Clear PIN Generate	Generates a clear personal identification number (PIN), a PIN verification value (PVV), or an offset using one of the following algorithms: • IBM 3624 (IBM-PIN or IBM-PINO) • IBM German Bank Pool (GBP-PIN or GBP-PINO) • VISA PIN validation value (VISA-PVV) • Interbank PIN (INBK-PIN)	"Clear PIN Generate (CSNBPGN)" on page 464
CSNBCPA	Clear PIN Generate Alternate	Generates a clear VISA PIN validation value (PVV) from an input encrypted PIN block. The PIN block might have been encrypted under either an input or output PIN encrypting key. The IBM-PINO algorithm is supported to produce a 3624 offset from a customer selected encrypted PIN. The PIN block must be encrypted under either an input PIN-encrypting key (IPINENC) or output PIN-encrypting key (OPINENC).	"Clear PIN Generate Alternate (CSNBCPA)" on page 468
CSNBCSG	CVV Generate	Generates a VISA Card Verification Value (CVV) or a MasterCard Card Verification Code (CVC) as defined for track 2.	"CVV Generate (CSNBCSG)" on page 472
CSNBCKC	CVV Key Combine	Combine two single-length operational DES keys that are suitable for use with the CVV (card-verification value) algorithm into one operational TDES key.	"CVV Key Combine (CSNBCKC)" on page 476
CSNBCSV	CVV Verify	Verifies a VISA Card Verification Value (CVV) or a MasterCard Card Verification Code (CVC) as defined for track 2.	"CVV Verify (CSNBCSV)" on page 481
CSNBEPG	Encrypted PIN Generate	Generates and formats a PIN and encrypts the PIN block.	"Encrypted PIN Generate (CSNBEPG)" on page 484
CSNBPTR	Encrypted PIN Translate	Re-enciphers a PIN block from one PIN-encrypting key to another and, optionally, changes the PIN block format. UKPT keywords are supported. You must identify the input PIN-encrypting key that originally enciphers the PIN. You also need to specify the output PIN-encrypting key that you want the verb to use to encipher the PIN. If you want to change the PIN block format, specify a different output PIN block format from the input PIN block format.	"Encrypted PIN Translate (CSNBPTR)" on page 489

Table 8. Summary of CCA AES, DES, and HMAC verbs (continued)

Entry point	Verb name	Description	Topic/Page
CSNBPTRE	Encrypted PIN Translate Enhanced	Reformats a PIN into a different PIN-block format using an enciphered PAN field. You can use this verb in an interchange-network application, or to change the PIN block to conform to the format and encryption key used in a PIN-verification database.	"Encrypted PIN Translate Enhanced (CSNBPTRE)" on page 495
CSNBPVR	Encrypted PIN Verify	Verifies a supplied PIN using one of the following algorithms: • IBM 3624 (IBM-PIN or IBM-PINO) • IBM German Bank Pool (GBP-PIN or GBP-PINO) • VISA PIN validation value (VISA-PVV) • Interbank PIN (INBK-PIN) UKPT keywords are supported.	"Encrypted PIN Verify (CSNBPVR)" on page 504
CSNBFPED	FPE Decipher	Decrypts payment card data for the Visa Data Secure Platform (VDSP) processing.	"FPE Decipher (CSNBFPED)" on page 509
CSNBFPEE	FPE Encipher	Encrypts payment card data for the Visa Data Secure Platform (VDSP) processing.	"FPE Encipher (CSNBFPEE)" on page 516
CSNBFPET	FPE Translate	Translates payment data from encryption under one key to encryption under another key with a possibly different format.	"FPE Translate (CSNBFPET)" on page 524
CSNBPCU	PIN Change/Unblock	Supports the PIN change algorithms specified in the VISA Integrated Circuit Card Specification; available only on an IBM z890 or IBM z990 with May 2004 or later version of Licensed Internal Code (LIC).	"PIN Change/Unblock (CSNBPCU)" on page 532
CSNBPFO	Recover PIN from Offset	Calculates the encrypted customer-entered PIN from a PIN generating key, account information, and an IBM-PINO Offset.	"Recover PIN from Offset (CSNBPFO)" on page 540
CSNBSKY	Secure Messaging for Keys	Encrypts a text block, including a clear key value decrypted from an internal or external DES token.	"Secure Messaging for Keys (CSNBSKY)" on page 544
CSNBSPN	Secure Messaging for PINs	Encrypts a text block, including a clear PIN block recovered from an encrypted PIN block.	"Secure Messaging for PINs (CSNBSPN)" on page 548
CSNBTRV	Transaction Validation	Supports the generation and validation of American Express card security codes; available only on an IBM z890 or IBM z990 with May 2004 or later version of Licensed Internal Code (LIC).	"Transaction Validation (CSNBTRV)" on page 552

Chapter 3. Introducing PKA cryptography and using PKA verbs

Read the provided introduction to Public Key Algorithms (PKA) and Elliptic Curve Cryptography (ECC). When you use the CCA PKA verbs, take note of these programming considerations, such as the PKA key token structure and key management.

You can use PKA support to exchange symmetric algorithm secret keys securely, and to compute digital signatures for authenticating messages to users.

The preceding chapters focused on AES or DES cryptography or secret-key cryptography. This cryptography is symmetric (senders and receivers use the same key, which must be exchanged securely in advance, to encipher and decipher data).

Public key cryptography does not require exchanging a secret key. It is asymmetric (the sender and receiver each have a pair of keys, a public key and a different but corresponding private key).

PKA key algorithms

Public key cryptography uses a key pair consisting of a public key and a private key.

The PKA public key uses one of the following algorithms:

Rivest-Shamir-Adleman (RSA)
The RSA algorithm is the most widely used and accepted of the public key algorithms. It uses three quantities to encrypt and decrypt text: a public exponent (PU), a private exponent (PR), and a modulus (M). Given these three and some cleartext data, the algorithm generates ciphertext as follows:

$$ciphertext = cleartext^{PU} \pmod{M}$$

Similarly, the following operation recovers cleartext from ciphertext:

$$cleartext = ciphertext^{PR} \pmod{M}$$

Elliptic Curve Digital Signature Algorithm (ECDSA)
The ECDSA algorithm uses elliptic curve cryptography (an encryption system based on the properties of elliptic curves) to provide a variant of the Digital Signature Algorithm.

PKA master keys

On the Cryptographic Coprocessor, PKA keys are protected by the Asymmetric-Keys Master Key (ASYM-MK).

The ASYM-MK is a triple-length DES key used to protect PKA private keys. On the Cryptographic Coprocessor, the ASYM-MK protects RSA private keys.

Starting with the IBM zEnterprise 196 configured with a CEX3C, there are two PKA master keys: the ASYM-MK mentioned above, and the 256-bit AES PKA Master Key (APKA-MK), used to protect ECC private keys stored in ECC key tokens.

In order for PKA verbs to function on the processor, the hash pattern of the ASYM-MK must match the hash pattern of the SYM-MK on the Cryptographic Coprocessor Feature. The administrator installs the PKA master keys on the Cryptographic Coprocessor Feature and the ASYM-MK on the coprocessor by using either the pass phrase initialization routine, the Clear Master Key Entry panels, or the optional Trusted Key Entry (TKE) workstation.

Operational private keys

Operational private keys are protected under two layers of DES encryption.

They are encrypted under an Object Protection Key (OPK) that in turn is encrypted under the ASYM-MK. You dynamically generate the OPK for each private key at import time or when the private key is generated on a CEX*C. CCA provides a public key storage file for the storage of application PKA keys. Although you cannot change PKA master keys dynamically, the PKA Key Token Change verb can be run to change a private PKA token (RSA or ECC) from encryption under the old ASYM-MK (or APKA-MK) to encryption under the current ASYM-MK (or APKA-MK).

PKA verbs

The CEX*C featrues provide application programming interfaces to PKA functions.

These PKA functions are:

- RSA digital signature functions
- Key management and key generation functions
- DES key distribution functions
- Data encryption functions
- ECC digital signature functions
- ECC key management and key generation functions
- ECC-based and RSA-based services for:
  - DES and AES key derivation
  - Diffie-Hellman key agreement for DES and AES
  - Key distribution functions for DES and AES keys

The CEX3C feature, which became available in March 2010 for the IBM System z10® and newer models, provides all the functions provided by the CEX2C and adds application programming interfaces to the following PKA functions:

- ECC digital signature functions
- ECC key management and key generation functions
- ECC-based and RSA-based services for:
  - DES and AES key derivation
  - Diffie-Hellman key agreement for DES and AES
  - Key distribution functions for DES and AES keys

Verbs supporting digital signatures

CCA provides verbs that support digital signatures.

These verbs are:

- Digital Signature Generate (CSNDDSG)
- Digital Signature Verify (CSNDDSV)

PKA key management

You can generate RSA and ECC keys using the CCA PKA Key Generate verb.

- Using the Transaction Security System PKA Key Generate verb, or a comparable product from another vendor.

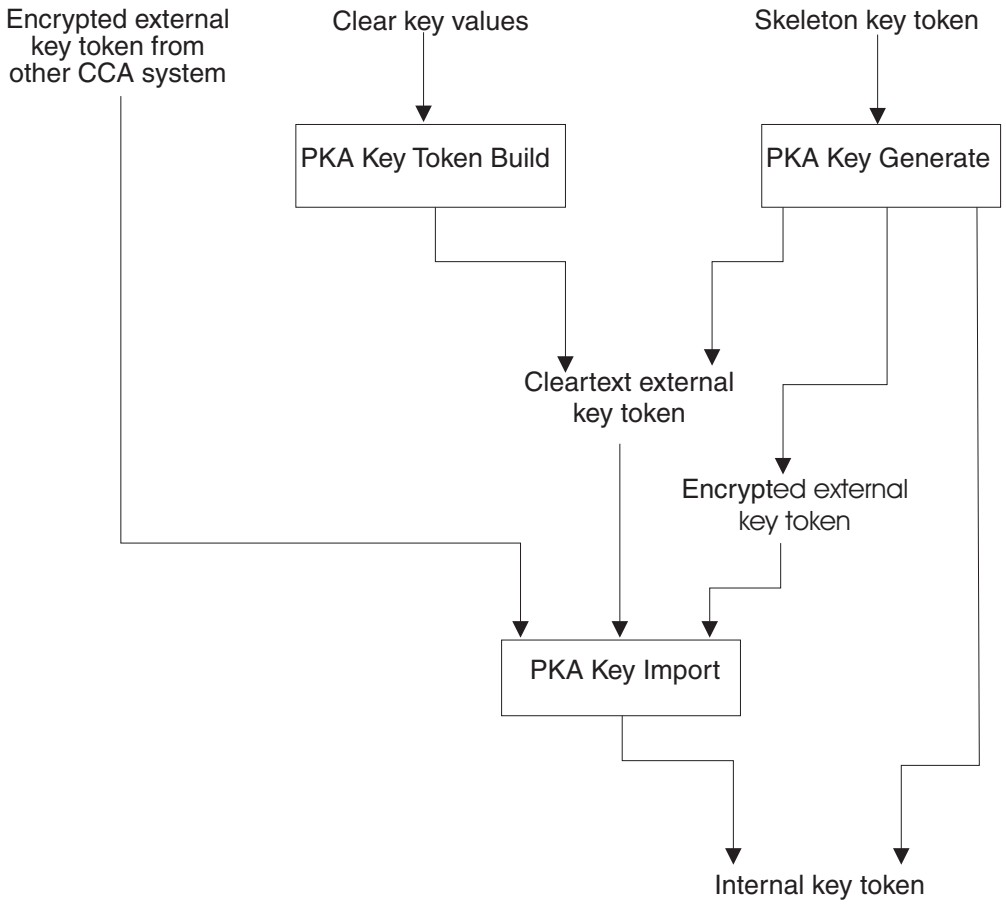


Figure 4. PKA key management

You can use the PKA Key Generate verb to generate internal and external PKA tokens. You can also generate RSA keys on another system and then import them to the cryptographic coprocessor. To input a clear RSA key, create the token with the PKA Key Token Build verb and import it using the PKA Key Import verb. To input an encrypted RSA key, use the PKA Key Import verb.

In either case, use the PKA Key Token Build verb to create a skeleton key token as input (see "PKA Key Token Build (CSNDPKB)" on page 644).

The PKA Key Import verb uses the clear token from the PKA Key Token Build verb or a clear or encrypted token from the CCA system to securely import the key token into operational form for the coprocessor to use. CCA does not permit the export of the imported PKA key.

The PKA Public Key Extract verb builds a public key token from a private key token.

Application RSA public and private keys can be stored in the PKA key storage file.

Verbs for PKA key management

CCA provides the following verbs for PKA key management:

- PKA Key Generate (CSNDPKG)
- PKA Key Import (CSNDPKI)
- PKA Key Token Build (CSNDPKB)
- PKA Key Token Change (CSNDKTC)
- PKA Key Translate (CSNDPKT)
- PKA Public Key Extract (CSNDPKX)
- Remote Key Export (CSNDRKX)
- Trusted Block Create (CSNDTBC)

PKA key tokens

PKA key tokens contain RSA or ECC private or public keys.

PKA tokens are variable length because they contain either RSA or ECC key values, which are variable in length. Consequently, length parameters precede all PKA token parameters. The maximum allowed size is 3500 bytes. PKA key tokens consist of a token header, any required sections, and any optional sections. Optional sections depend on the token type. PKA key tokens can be public or private, and private key tokens can be internal or external. Therefore, there are three basic types of tokens, each of which can contain either RSA or ECC information:

- A public key token
- A private external key token
- A private internal key token

Public key tokens contain only the public key. Private key tokens contain the public and private key pair. Table 9 summarizes the sections in each type of token.

Table 9. Summary of PKA key token sections

Section	Public external key token	Private external key token	Private internal key token
Header	X	X	X
RSA or ECC private key information		X	X
RSA or ECC public key information	X	X	X
Key name (optional)		X	X
Internal information			X

As with DES key tokens, the first byte of a PKA key token contains the token identifier which indicates the type of token.

A first byte of X'1E' indicates an external token with a cleartext public key and optionally a private key that is either in cleartext or enciphered by a transport key-encrypting key. An external key token is in importable key form. It can be sent on the link.

A first byte of X'1F' indicates an internal token with a cleartext public key and a private key that is enciphered by the PKA master key and ready for internal use. An internal key token is in operational key form. A PKA private key token must be in operational form for the coprocessor to use it. (PKA public key tokens are used directly in the external form.)

Formats for public and private external and internal RSA and ECC key tokens begin in "RSA public key token" on page 775.

Key identifier for PKA key token

A *key identifier* for a PKA key token is a variable length (maximum allowed size is 2500 bytes) area that contains either a key label or a key token.

- A **key label** identifies keys that are in the PKA key storage file.
- A **key token** can be either an internal key token, an external key token, or a null key token. Key tokens are generated by an application (for example, using the PKA Key Generate verb), or received from another system that can produce external key tokens.

  An **internal key token** can be used only on the local system, because the PKA master key encrypts the key value. Internal key tokens contain keys in operational form only.

  An **external key token** can be exchanged with other systems because a transport key that is shared with the other system encrypts the key value. External key tokens contain keys in either exportable or importable form.

  A **null key token** consists of eight bytes of binary zeros. The PKA Key Record Create verb can be used to write a null token to the key storage file. This record can subsequently be identified as the target token for the PKA Key Import or PKA Key Generate verb.

The term *key identifier* is used when a parameter could be one of the above items, and indicates that different inputs are possible. For example, you might want to specify a specific parameter as either an internal key token or a key label. The key label is, in effect, an indirect reference to a stored internal key token.

Key label

If the first byte of the key identifier is greater than X'20' but less than X'FF', the field is considered to be holding a **key label**.

The contents of a key label are interpreted as the identifier of a key entry in the PKA storage file. The key label is an indirect reference to an internal key token.

If the first byte of the key identifier is X'FF', the identifier is not valid. If the first byte is less than X'20', the identifier is treated as a key token as described below.

A key label is specified on verbs with the *key_identifier* parameter as a 64-byte character string, left-aligned, and padded on the right with blanks. In most cases, the verb does not check the syntax of the key label other than the first byte.

A key label has the following form:

Offset	Length	Data
00 - 63	64	Key label name

Key token

A key token is a variable length (maximum allowed size is 3500 bytes) field composed of key value and control information.

PKA keys can be either public or private RSA, or ECC keys. Each key token can be either an internal key token (the first byte of the key identifier is X'1F'), an external key token (the first byte of the key identifier is X'1E'), or a null PKA private key token (the first byte of the key identifier is X'00').

For descriptions of the PKA key tokens and for debugging information, see Chapter 17, “Key token formats,” on page 769.

Internal key token

An *internal key token* is a token that can be used only on the system that created it or another system with the same PKA master key.

It contains a key that is encrypted under the PKA master key.

An application obtains an internal key token by using one of the verbs such as those listed below. The verbs are described in detail in Chapter 13, “Managing PKA cryptographic keys,” on page 635.

- PKA Key Generate
- PKA Key Import

The PKA Key Token Change verb can re-encipher private internal tokens from encryption under the old ASYM-MK to encryption under the current ASYM-MK. PKA key storage Re-encipher/Activate options are available to re-encipher RSA and ECC internal tokens in the PKA key storage when the SYM-MK/ASYM-MK (or APKA-MK) keys are changed.

PKA master keys cannot be changed dynamically.

External key token

If the first byte of the key identifier is X'1E', the key identifier is interpreted as an *external key token*.

An external PKA key token contains key (possibly encrypted) and control information. By using the external key token, you can exchange keys between systems.

An application obtains the external key token by using one of the verbs such as those listed below. They are described in detail in Chapter 13, “Managing PKA cryptographic keys,” on page 635.

- PKA Public Key Extract
- PKA Key Token Build
- PKA Key Generate

Null key token

If the first byte of the key identifier is X'00', the key identifier is interpreted as a *null key token*.

Summary of the PKA verbs

Use this table of the PKA verbs to map them to their corresponding verb names and descriptions.

The PKA verb names start with CSND. This table also references the topic that describes the verb.

Table 10. Summary of PKA verbs

Entry point	Verb name	Description	Topic/Page
Chapter 12, "Using digital signatures," on page 625			
CSNDDSG	Digital Signature Generate	Generates a digital signature using an RSA or ECC private key.	"Digital Signature Generate (CSNDDSG)" on page 625
CSNDDSV	Digital Signature Verify	Verifies a digital signature using an RSA or ECC public key.	"Digital Signature Verify (CSNDDSV)" on page 629
Chapter 13, "Managing PKA cryptographic keys," on page 635			
CSNDPKG	PKA Key Generate	Generates an RSA key pair.	"PKA Key Generate (CSNDPKG)" on page 635
CSNDPKI	PKA Key Import	Imports a key token containing either a clear key or an RSA or ECC key enciphered under a transport key.	"PKA Key Import (CSNDPKI)" on page 640
CSNDPKB	PKA Key Token Build	Creates an external PKA key token containing a clear private RSA key. Using this token as input to the PKA Key Import verb returns an operational internal token containing an enciphered private key. Using PKA Key Token Build on a clear public RSA key, returns the public key in a token format that other PKA verbs can directly use. PKA Key Token Build can also be used to create a skeleton token for input to the PKA Key Generate verb for the generation of an internal RSA key token.	"PKA Key Token Build (CSNDPKB)" on page 644
CSNDKTC	PKA Key Token Change	Changes PKA key tokens from encipherment with the old asymmetric-keys master key to encipherment with the current asymmetric-keys master key. This verb changes only private internal tokens.	"PKA Key Token Change (CSNDKTC)" on page 652

Table 10. Summary of PKA verbs (continued)

Entry point	Verb name	Description	Topic/Page
CSNDPKT	PKA Key Translate	Translates PKA key tokens from encipherment under the old Asymmetric-Keys Master Key to encipherment under the current Asymmetric-Keys Master Key. This verb changes only Private Internal PKA Key Tokens.	"PKA Key Translate (CSNDPKT)" on page 655
CSNDPKX	PKA Public Key Extract	Extracts a PKA public key token from a supplied PKA internal or external private key token. Performs no cryptographic verification of the PKA private token.	"PKA Public Key Extract (CSNDPKX)" on page 662
CSNDRKX	Remote Key Export	Secure transport of DES keys using asymmetric techniques from a security module (for example, the CEX*C) to a remote device such as an Automated Teller Machine (ATM).	"Remote Key Export (CSNDRKX)" on page 664
CSNDTBC	Trusted Block Create	Creates an external trusted block under dual control. A trusted block is an extension of CCA PKA key tokens using new section identifiers.	"Trusted Block Create (CSNDTBC)" on page 676

Chapter 4. TR-31 symmetric-key management

X9 TR-31 is defined in X9 TR-31 2010: *Interoperable Secure Key Exchange Block Specification for Symmetric Algorithms*.

The provided information is an extension of Chapter 6, "Managing AES and DES cryptographic keys," on page 129. For additional information on symmetric keys, including DES control vectors, see Chapter 6, "Managing AES and DES cryptographic keys," on page 129.

The TR-31 key block is a format defined by the ANSI Standards Committee to support interchange of symmetric keys in a secure manner and with key attributes included in the exchanged data. Currently, this format supports only DES keys. AES keys are not supported.

TR-31 is a Technical Report. This is different from a standard, which is a mandatory set of rules that must be followed. A Technical Report is not mandatory, but provides guidance to those who are using the standards. In this case, TR-31 is a companion to the standard X9.24-1, which defines requirements for key management performed using symmetric key techniques. TR-31 shows a method that complies with the various requirements that are defined in X9.24-1, and because no other specific method has been defined by the standards committee, the TR-31 method is becoming the apparent standard through which financial organizations will exchange keys.

Prior to TR-31, there were problems with the interchange of symmetric keys. In the banking environment, it is very important that each symmetric key have a specific set of attributes attached to it, specifying such things as the cryptographic operations for which that key can be used. CCA implements these attributes in the form of the control vector (CV), but other vendors implement attributes in their own proprietary ways. Thus, if you are exchanging keys between CCA systems, you can securely pass the attributes using CCA functions and data structures. If, however, that same key were sent to a non-CCA system, there would be no secure way to do that. This is because the two cryptographic architectures have no common key format that could be used to pass both the key and its attributes. As a result, the normal approach has been to strip the attributes and send just the encrypted key, then attach attributes again at the receiving end.

The above scenario has major security problems because it allows an insider to obtain the key without its designated attributes. The insider can then attach other attributes to it, thereby compromising the security of the system. For example, assume the exchanged key is a key-encrypting key (KEK). The attributes of a KEK should restrict its use to key management functions that are designed to prevent exposure of the keys that the KEK is used to encrypt. If that KEK is transmitted without any attributes, an attacker on the inside can turn the key into a type used for data decryption. Such a key can then be used to decipher all of the keys that were previously protected using the KEK. It is clearly very desirable to have a way of exchanging keys that prevents this modification of the attributes. TR-31 provides such a method.

The TR-31 key block has a set of defined key attributes. These attributes are securely bound to the key so that they can be transported together between any two systems that both understand the TR-31 format. This is much of the reason for

its gain in popularity. There are two supported cryptographic methods for protecting the key block. The original version of TR-31 defined a method that encrypted the key field in CBC mode and computed a TDES MAC over the header and key field. The encryption and MAC operations used different keys, created by applying predefined variants to the input key block protection key. This method is identified by a Key Block Version ID value of "A" (X'41'). An update to TR-31 adds a more modern method, identified by a Key Block Version ID value of "B" (X'42') or "C" (X'43'). The "B" method uses an authenticated encryption scheme and uses cryptographic key derivation methods to produce the encryption and MAC keys. The "C" method is exactly the same as the "A" method in terms of wrapping keys. However, the field values are expected to conform to the updated standard.

Not surprisingly, TR-31 uses some key attributes that are different from those in the CCA control vector. In some cases, there is a one-to-one correspondence between CCA and TR-31 attributes. For these cases, conversion is simple and straightforward. In other cases, the correspondence is one-to-many or many-to-one and the application program must provide information to help the CCA verbs decide how to perform the translation between CCA and TR-31 attributes. There are also CCA attributes that simply cannot be represented using TR-31. CCA keys with those attributes are not eligible for conversion to TR-31 format.

The TR-31 key block has these two important features:

1. The key is protected in such a way that it meets the "key bundling" requirements of various standards. These standards state that the individual 8-byte blocks of a double-length or triple-length TDES key must be bound in such a way that they cannot be individually manipulated. TR-31 accomplishes this mainly by computation of a MAC across the entire structure, excluding the MAC value itself.
2. Key usage attributes, defined to control how the key can be used, are securely bound to the key itself. This makes it possible for a key and its attributes to be securely transferred from one party to another while assuring that the attributes of the key cannot be modified to suit the needs of an attacker.

CCA support the management of DES keys using TR-31. Table 11 lists the verbs described in this book. See Chapter 14, “TR-31 symmetric key management verbs,” on page 681 for a detailed description of the verbs.

Table 11. TR-31 symmetric key management verbs

Verb	Page	Service	Entry point	Service location
Key Export to TR31	“Key Export to TR31 (CSNBT31X)” on page 681	Exports a CCA external or internal fixed-length symmetric key-token, converting it into an external X9 TR-31 key block format.	CSNBT31X	cryptographic engine
TR31 Key Import	“TR31 Key Import (CSNBT31I)” on page 707	Imports an external X9 TR-31 key block, converting it into a CCA external or internal fixed-length symmetric key-token.	CSNBT31I	cryptographic engine
TR31 Key Token Parse	“TR31 Key Token Parse (CSNBT31P)” on page 730	Parses the information from the standard predefined fields of the TR-31 key block header without importing the key.	CSNBT31P	security API host software

Table 11. TR-31 symmetric key management verbs (continued)

Verb	Page	Service	Entry point	Service location
TR31 Optional Data Build	"TR31 Optional Data Build (CSNBT31O)" on page 734	Constructs the optional blocks of a TR-31 key block, one block at a time.	CSNBT31O	security API host software
TR31 Optional Data Read	"TR31 Optional Data Read (CSNBT31R)" on page 737	Obtains the contents of any optional fields of a TR-31 key block header.	CSNBT31R	security API host software

Part 2. CCA verbs

CCA supports AES, DES, and PKA verbs.

It contains the following topics:

- Chapter 5, "Using the CCA nodes and resource control verbs," on page 77 describes how to use the CCA resource control verbs.
- Chapter 6, "Managing AES and DES cryptographic keys," on page 129 describes the verbs for generating and maintaining AES, DES, and HMAC cryptographic keys, the Random Number Generate verb (which generates 8-byte random numbers), the Random Number Generate Long verb (which generates up to 8192 bytes of random content), and the Secure Sockets Layer (SSL) security protocol. This chapter also describes utilities to build DES and AES tokens, generate and translate control vectors, and describes the PKA verbs that support DES and AES key distribution.
- Chapter 7, "Protecting data," on page 333 describes the verbs for enciphering and deciphering data.
- Chapter 8, "Verifying data integrity and authenticating messages," on page 369 describes the verbs for generating and verifying Message Authentication Codes (MACs), generating Modification Detection Codes (MDCs) and generating hashes (SHA-1, MD5, RIPEMD-160).
- Chapter 9, "Key storage mechanisms," on page 401 describes the use of key storage, key tokens, and associated verbs.
- Chapter 10, "Financial services," on page 443 describes the verbs for use in support of finance-industry applications. This includes several categories.
  - Verbs for generating, verifying, and translating personal identification numbers (PINS).
  - Verbs that generate and verify VISA card verification values and American Express card security codes.
  - Verbs to support smart card applications using the EMV (Europay MasterCard Visa) standards.
- Chapter 11, "Financial services for DK PIN methods," on page 557 contains information on financial services that are based on the PIN methods and requirements specified by the German Banking Industry Committee (Deutsche Kreditwirtschaft (DK)).
- Chapter 12, "Using digital signatures," on page 625 describes the verbs that support using digital signatures to authenticate messages.
- Chapter 13, "Managing PKA cryptographic keys," on page 635 describes the verbs that generate and manage PKA keys.
- Chapter 14, "TR-31 symmetric key management verbs," on page 681 describes the verbs that manage TR-31 functions.
- Chapter 15, "Utility verbs," on page 743 describes the Code Conversion (CSNBXEA) utility. Use this verb to convert text strings from ASCII to EBCDIC or from EBCDIC to ASCII.

Chapter 5. Using the CCA nodes and resource control verbs

Use these verbs to control CCA nodes and their resources.

The following verbs are described:

Cryptographic Facility Query (CSUACFQ)

The Cryptographic Facility Query verb is used to retrieve information about the coprocessor and the CCA application program in that coprocessor.

This information includes the following:

- General information about the coprocessor, its operating system, and CCA application
- The Environment Identifier (EID)
- Diagnostic information from the coprocessor
- Export-control information from the coprocessor
- Time and date information from the coprocessor
- The contents and size of the authorized PIN decimalization tables loaded onto the coprocessor

On input, you specify:

- A *rule_array_count* of 1 or 2
- Optionally, a *rule_array* keyword of **ADAPTER1** (for backward compatibility)
- The class of information queried with a *rule_array* keyword

This verb returns information elements in the *rule_array* and sets the *rule_array_count* variable to the number of returned elements.

Determining if a card is a CEX2C, CEX3C, CEX4C, or a CEX5C

Using the Cryptographic Facility Query, the output **rule_array** for option STATCCA is the most accurate way to determine the cryptographic coprocessor type.

- If first two characters of the CCA application version field are 'z' followed by '3', then this card is a CEX2C adapter.

  An updated device driver might not be available yet for all distributions where this RPM is usable. The CCA host library uses this mechanism to determine card version, and we recommend here that the application developer also use this method. Where this output and the device driver disagree about the version of a

particular card, it is the device driver that will be out of date because the Cryptographic Facility Query data is not interpreted in any way; it comes direct from the adapter.

- If first character of the CCA application version field is a number, such as '4' or greater, then this card is *not* a CEX2C. For example, a '4' in the first character indicates a CEX3C or CEX4C.
- The results of this query come directly from the card itself. If the host device driver is not up to date, it could incorrectly identify a CEX3C or CEX4C as a CEX2C. Therefore, looking at the CCA application version field for the output *rule_array* for option STATCCA resolves all questions.
- If the first character of the CCA application version field is the number 5, then this card is a CEX5C.

The commands **ivp.e** and **panel.exe -x** also tell you the cryptographic coprocessor type, by calling the Cryptographic Facility Query verb for all available adapters.

For details about panel.exe, see "The panel.exe utility" on page 1003.

Format

The format of CSUACFQ.

```
CSUACFQ(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    verb_data_length,
    verb_data)
```

Parameters

The parameter definitions for CSUACFQ.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. On input, this value must be 1 or 2.

On output, the verb sets the variable to the number of *rule_array* elements it returns to the application program.

Tip: With this verb, the number of returned *rule_array* elements can exceed the *rule_array_count* you specified on input. Be sure you allocate adequate memory to receive all the information elements according to the information class you select on input with the information-to-return keyword in the *rule_array*.

rule_array

Direction: Input/Output

Type: String array

The **rule_array** parameter is a pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters.

On input, set the **rule_array** to specify the type of information to retrieve. There are two input *rule_array* elements, as described in Table 12. This table also indicates to which parameter, **rule_array** or **verb_data**, output data is returned for each keyword.

Table 12. Keywords for Cryptographic Facility Query control information

Keyword	Description	Output in *rule_array*	Output in *verb_data*
Adapter to use (Optional)			
ADAPTER1	This keyword is ignored. It is accepted for backward compatibility.	n/a	n/a
Information to return (One required)			
GET-UDX	Obtains UDX identifiers. This keyword applies only when using Linux on z Systems.	None.	See "GET-UDX" on page 93.

MOREMKS	This keyword returns additional master key information. It is in a second rule array class and used to signify that additional data is to be returned with one or more existing keywords. This keyword is only allowed when one of STATICSFB, STATCCA, or STATCCAE is also present. Any other use results in an error return code of (8 / 0x22 [34]).
	
	When present, the reply rule array elements for the SYM and ASYM master keys have additional bytes populated. The master key status is still present in the same format in offset 0 of the element.
	
	When STATICSFB is present, an ASCII '1' is in offset 7 of the reply rule array element if ACP '0x0330' was set when the master key parts were loaded. This forces the user to enter 24-bytes of key material. This is only present for the SYM MK.
	
	Example output for **STATICSFB MOREMKS** (truncated to show only the SYM reply):
	`Rule array kw 0: Card Serial Number`
	`Rule array kw 1: "3 1" SYM MK new mk reg is full and 24-bytes entered.`
	`Rule array kw 2: "2 " SYM MK valid (contains a key) current mk.`
	`Rule array kw 3: "2 " SYM MK valid old mk.`		

NUM-DECT	Returns the number of bytes of data required for the *verb_data* variable when the **STATDECT** rule-array keyword is specified. **Note:** A TKE is used to securely load PIN decimalization tables.	None.	See "NUM-DECT" on page 94.
QPENDING	TKE uses this *rule_array* keyword to request information about pending changes previously submitted by this TKE or another TKE to this adapter. Only TKE can submit changes to be stored in the Pending Change Buffer queried with this command. The keyword is available for normal users of Cryptographic Facility Query, for informational or debugging reasons (no secrets are exposed). This keyword applies only when using Linux on z Systems.	See Table 13 on page 81.	None.

Table 12. Keywords for Cryptographic Facility Query control information (continued)

Keyword	Description	Output in *rule_array*	Output in *verb_data*
SIZEWPIN	Get the number of bytes of storage required for the output of a STATWPIN request.	None.	See Table 14 on page 94.
STATAES	Obtains status information on AES master-key registers and AES key-length enablement.	See Table 13 on page 81.	None.
STATAPKA	Obtains status information on APKA master-key registers and APKA key-length enablement. This keyword was introduced with CCA 4.1.0.	See Table 13 on page 81.	None.
STATCARD	Obtains coprocessor-related basic status information. This keyword is provided for backwards compatibility. The **STATCRD2** should be used instead of **STATCARD**.	See Table 13 on page 81.	None.
STATCCA	Obtains CCA-related status information.	See Table 13 on page 81.	None.
STATCCAE	Obtains CCA-related extended status information.	See Table 13 on page 81.	None.
STATCRD2	Obtains extended basic status information about the coprocessor.	See Table 13 on page 81.	None.
STATDECT	Obtains the information on all of the authorized PIN decimalization tables that are currently stored on the coprocessor. Output is returned in the *verb_data* variable. **Note:** A TKE is used to securely load PIN decimalization tables.	None.	See "STATDECT" on page 94.
STATDIAG	Obtains diagnostic information.	See Table 13 on page 81.	None.
STATEID	Obtains the Environment Identifier (EID).	See Table 13 on page 81.	None.
STATEXPT	Obtains function control vector-related status information.	See Table 13 on page 81.	None.
STATICSA	Obtains the indicated master key hash and verification patterns to be returned for the master keys loaded in the current domain. This keyword applies only when using Linux on z Systems.	See Table 13 on page 81.	See "STATICSA" on page 94.
STATICSB	Obtains the indicated master key hash and verification patterns to be returned for the master keys loaded in the current domain. See "STATICSB" on page 97. This keyword was introduced with CCA 4.1.0. This keyword applies only when using Linux on z Systems.	See Table 13 on page 81.	See "STATICSB" on page 97.
STATICSE	Obtains the indicated master key hash and verification patterns to be returned for the master keys loaded in the current domain. This keyword applies only when using Linux on z Systems.	See Table 13 on page 81.	See "STATICSE" on page 100.
STATICSF	This keyword returns the adapter serial number and status information about the SYM (DES) and ASYM (RSA) master-key registers, including whether a valid key is present in each of the old, current, and new registers. This keyword applies only when using Linux on z Systems.	See Table 13 on page 81.	None.
STATICSX	Obtains the indicated master key hash and verification patterns to be returned for the master keys loaded in the current domain. This keyword applies only when using Linux on z Systems.	See Table 13 on page 81.	See "STATICSX" on page 102.

Table 12. Keywords for Cryptographic Facility Query control information (continued)

Keyword	Description	Output in *rule_array*	Output in *verb_data*
STATKPR	Obtains non-secret information about an operational key part. This keyword applies only when using Linux on z Systems.	None.	See "STATKPR" on page 104.
STATKPRL	Obtains the names of the operational key parts. This keyword applies only when using Linux on z Systems.	None.	"STATKPRL" on page 105.
STATMOFN	Obtains master-key shares distribution information.	See Table 13.	None.
STATVKPL	Obtains the names of all the operational key parts for variable length key token preparation. This keyword applies only when using Linux on z Systems.	None.	See "STATVKPL" on page 105.
STATVKPR	Obtains non-secret information about an operational key part. This is different from STATKPR in that a register for creating a key in a variable length key token is described. This keyword applies only when using Linux on z Systems.	None.	See "STATVKPR" on page 105.
STATWPIN	Returns the state information on all of the weak PIN entries that are currently stored on the coprocessor.	None.	See Table 22 on page 107.
TIMEDATE	Reads the current date, time, and day of the week from the secure clock within the coprocessor.	See Table 13.	None.
TKESTATE	Indicates whether TKE access is enabled or not. This keyword applies only when using Linux on z Systems.	See Table 13.	None.
WRAPMTHD	Obtains the default key wrapping method. This keyword was introduced with CCA 4.1.0.	See Table 13.	None.

Different sets of **rule_array** elements are returned, depending on the input keyword. Table 13 describes these **rule_array** elements for keywords that result in output data in the **rule_array** parameter.

For *rule_array* elements that contain numbers, those numbers are represented by numeric characters which are left-aligned and padded on the right with space characters. For example, a **rule_array** element that contains the number 2 contains the character string **"2 "** (the number 2 followed by seven space characters).

For some keywords, there is output data in the **verb_data** variable. This output data is described in "Verb data returned for CSUACFQ keywords" on page 93.

Table 13. Cryptographic Facility Query information returned in the rule_array

Element number	Name	Description
Output rule_array for option QPENDING		

Table 13. Cryptographic Facility Query information returned in the rule_array (continued)

Element number	Name	Description
1	Change type (ASCII number)	An ASCII number that indicates the type of pending change stored in the adapter (if there is one) **Value** **Description** **none** No pending change **1** Role load **2** Profile load **3** Role delete **4** Profile delete **5** Domain zeroize **6** Enable
2	user ID (string)	A string of eight ASCII characters for the user ID of the user who initiated the pending change.
Output rule_array for option STATAES		
1	AES NMK status	State of the AES new master key register: **Value** **Description** **1** Register is clear **2** Register contains a partially complete key **3** Register contains a complete key
2	AES CMK status	State of the AES current master key register: **Value** **Description** **1** Register is clear **2** Register contains a key
3	AES OMK status	State of the AES old master key register: **Value** **Description** **1** Register is clear **2** Register contains a key
4	AES key length enablement	The maximum AES key length that is enabled by the function control vector. The value is 0 (if no AES key length is enabled in the FCV), 128, 192, or 256.
Output rule_array for option STATAPKA		
1	ECC NMK status	The state of the ECC new master key register: **Value** **Description** **1** Register is clear **2** Register contains a partially complete key **3** Register contains a complete key
2	ECC CMK status	The state of the ECC current master key register: **Value** **Description** **1** Register is clear **2** Register contains a key
3	ECC OMK status	The state of the ECC old master key register: **Value** **Description** **1** Register is clear **2** Register contains a key
4	ECC key length enablement	The maximum ECC curve size that is enabled by the function control vector. The value will be 0 (if no ECC keys are enabled in the FCV) and 521 for the maximum size.
Output rule_array for option STATCARD		

Table 13. Cryptographic Facility Query information returned in the rule_array (continued)

Element number	Name	Description
1	Number of installed adapters	A numeric character string containing the number of active coprocessors installed in the machine. This includes only coprocessors that have CCA software loaded (including those with CCA UDX software). Non-CCA coprocessors are not included in this number.
2	DES hardware level	A numeric character string containing an integer value identifying the version of DES hardware on the coprocessor.
3	RSA hardware level	A numeric character string containing an integer value identifying the version of RSA hardware on the coprocessor.
4	POST version	A character string identifying the version of the coprocessor's Power-On Self Test (POST) firmware. The first four characters define the POST0 version and the last four characters define the POST1 version.
5	Coprocessor operating system name	A character string identifying the operating system firmware on the coprocessor.
6	Coprocessor operating system version	A character string identifying the version of the coprocessor's operating system firmware.
7	Coprocessor part number	A character string containing the 8 character part number identifying the version of the coprocessor.
8	Coprocessor EC level	A character string containing the 8 character engineering change (EC) level for this version of the coprocessor.
9	Miniboot version	A character string identifying the version of the coprocessor's miniboot firmware. This firmware controls the loading of programs into the coprocessor. The first four characters define the MiniBoot0 version and the last four characters define the MiniBoot1 version.
10	CPU speed	A numeric character string containing the operating speed of the microprocessor chip, in megahertz.
11	Adapter ID (see also element number 15)	A unique identifier manufactured into the coprocessor. The coprocessor adapter ID is an 8-byte **binary** value.
12	Flash memory size	A numeric character string containing the size of the flash EPROM memory on the coprocessor, in 64 KB increments.
13	DRAM memory size	A numeric character string containing the size of the dynamic RAM (DRAM) memory on the coprocessor, in kilobytes.
14	Battery-backed memory size	A numeric character string containing the size of the battery-backed RAM on the coprocessor, in kilobytes.
15	Serial number	A character string containing the unique serial number of the coprocessor. The serial number is factory installed.
Output rule_array for option STATCCA		
1	NMK status	The state of the new master-key register: **Value** **Description** **1** The register is clear. **2** The register contains a partially complete key. **3** The register contains a key.

Table 13. Cryptographic Facility Query information returned in the rule_array (continued)

Element number	Name	Description
2	CMK status	The state of the current master-key register: **Value** **Description** **1** The register is clear. **2** The register contains a key.
3	OMK status	The state of the old master-key register: **Value** **Description** **1** The register is clear. **2** The register contains a key.
4	CCA application version	A character string that identifies the version of the CCA application program running in the coprocessor. If first two characters are 'z' followed by '3', then this card is a CEX2C adapter (no matter what device driver indicates). If first character is a number, such as '4' or greater, then this card is *not* a CEX2C. For example, a '4' in the first character indicates a CEX3C or CEX4C. A '5' in the first character indicates a CEX5C. The results of this query come directly from the card itself. If the host device driver is not up to date, it could incorrectly identify a CEX*C as a CEX2C. Therefore, looking at this field resolves all questions.
5	CCA application build date	A character string containing the build date for the CCA application program running in the coprocessor.
6	User role	A character string containing the role identifier which defines the host application user's current authority.
Output rule_array for option STATCCAE		
1	Symmetric NMK status	The state of the symmetric new master-key register: **Value** **Description** **1** The register is clear. **2** The register contains a partially complete key. **3** The register contains a key.
2	Symmetric CMK status	The state of the symmetric current master-key register: **Value** **Description** **1** The register is clear. **2** The register contains a key.
3	Symmetric OMK status	The state of the symmetric old master-key register: **Value** **Description** **1** The register is clear. **2** The register contains a key.
4	CCA application version	A character string that identifies the version of the CCA application program that is running in the coprocessor.
5	CCA application build date	A character string containing the build date for the CCA application program that is running in the coprocessor.
6	User role	A character string containing the role identifier which defines the host application user's current authority.

Table 13. Cryptographic Facility Query information returned in the rule_array (continued)

Element number	Name	Description
7	Asymmetric NMK status	The state of the asymmetric new master-key register: **Value** **Description** **1** The register is clear. **2** The register contains a partially complete key. **3** The register contains a key.
8	Asymmetric CMK status	The state of the asymmetric current master-key register: **Value** **Description** **1** The register is clear. **2** The register contains a key.
9	Asymmetric OMK status	The state of the asymmetric old master-key register: **Value** **Description** **1** The register is clear. **2** The register contains a key.
Output rule_array for option STATCRD2		
1	Number of installed adapters	A numeric character string containing the number of active coprocessors installed in the machine. This includes only coprocessors that have CCA software loaded (including those with CCA UDX software). Non-CCA coprocessors are not included in this number.
2	DES hardware level	A numeric character string containing an integer value identifying the version of DES hardware on the coprocessor.
3	RSA hardware level	A numeric character string containing an integer value identifying the version of RSA hardware on the coprocessor.
4	POST version	A character string identifying the version of the coprocessor's Power-On Self Test (POST) firmware. The first four characters define the POST0 version and the last four characters define the POST1 version.
5	Coprocessor operating system name	A character string identifying the operating system firmware on the coprocessor.
6	Coprocessor operating system version	A character string identifying the version of the coprocessor's operating system firmware.
7	Coprocessor part number	A character string containing the 8 character part number identifying the version of the coprocessor.
8	Coprocessor EC level	A character string containing the 8 character engineering change (EC) level for this version of the coprocessor.
9	Miniboot version	A character string identifying the version of the coprocessor's miniboot firmware. This firmware controls the loading of programs into the coprocessor. The first four characters define the MiniBoot0 version and the last four characters define the MiniBoot1 version.
10	CPU speed	A numeric character string containing the operating speed of the microprocessor chip, in megahertz.
11	Adapter ID (see also element number 15)	A unique identifier manufactured into the coprocessor. The coprocessor adapter ID is an 8-byte **binary** value.

Table 13. Cryptographic Facility Query information returned in the rule_array (continued)

Element number	Name	Description
12	Flash memory size	A numeric character string containing the size of the flash EPROM memory on the coprocessor, in 64 KB increments.
13	DRAM memory size	A numeric character string containing the size of the dynamic RAM (DRAM) memory on the coprocessor, in kilobytes.
14	Battery-backed memory size	A numeric character string containing the size of the battery-backed RAM on the coprocessor, in kilobytes.
15	Serial number	A character string containing the unique serial number of the coprocessor. The serial number is factory installed.
16	POST2 version	A character string identifying the version of the coprocessor's POST2 firmware. The first four characters define the POST2 version, and the last four characters are reserved and valued to space characters.
Output rule_array for option STATDIAG		
1	Battery state	A numeric character string containing a value which indicates whether the battery on the coprocessor needs to be replaced: **Value** **Description** **1** The battery is good. **2** The battery should be replaced.
2	Intrusion latch state	A numeric character string containing a value which indicates whether the intrusion latch on the coprocessor is set or cleared: **Value** **Description** **1** The latch is cleared. **2** The latch is set.
3	Error log status	A numeric character string containing a value which indicates whether there is data in the coprocessor CCA error log: **Value** **Description** **1** The error log is empty. **2** The error log contains abnormal termination data, but is not yet full. **3** The error log is full and cannot hold any more data.
4	Mesh intrusion	A numeric character string containing a value to indicate whether the coprocessor has detected tampering with the protective mesh that surrounds the secure module. This indicates a probable attempt to physically penetrate the module: **Value** **Description** **1** No intrusion has been detected. **2** An intrusion attempt has been detected.
5	Low voltage detected	A numeric character string containing a value to indicate whether a power-supply voltage was below the minimum acceptable level. This might indicate an attempt to attack the security module: **Value** **Description** **1** Only acceptable voltages have been detected. **2** A voltage has been detected below the low-voltage tamper threshold.

Table 13. Cryptographic Facility Query information returned in the rule_array (continued)

Element number	Name	Description
6	High voltage detected	A numeric character string containing a value indicates whether a power-supply voltage was greater than the maximum acceptable level. This might indicate an attempt to attack the security module: **Value** **Description** **1** Only acceptable voltages have been detected. **2** A voltage has been detected greater than the high-voltage tamper threshold.
7	Temperature range exceeded	A numeric character string containing a value to indicate whether the temperature in the secure module was outside of the acceptable limits. This might indicate an attempt to attack the security module: **Value** **Description** **1** The temperature is acceptable. **2** The temperature has been detected outside of an acceptable limit.
8	Radiation detected	A numeric character string containing a value to indicate whether radiation was detected inside the secure module. This might indicate an attempt to attack the security module: **Value** **Description** **1** No radiation has been detected. **2** Radiation has been detected.
9, 11, 13, 15, 17	Last 5 commands run	These five *rule_array* elements contain the last five commands that were run by the coprocessor CCA application. They are in chronological order, with the most recent command in element 9. Each element contains the security API command code in the first four characters and the subcommand code in the last four characters. See Table 273 on page 1009.
10, 12, 14, 16, 18	Last 5 return codes	These five *rule_array* elements contain the security API return codes and reason codes corresponding to the five commands in *rule_array* elements 9, 11, 13, 15, and 17. Each element contains the return code in the first four characters and the reason code in the last four characters.
Output rule_array for option STATEID		
1, 2	EID	The two elements, when concatenated, provide the 16-byte Environment Identifier (EID) value.
Output rule_array for option STATEXPT		
1	Base CCA services availability	A numeric character string containing a value to indicate whether base CCA services are available: **Value** **Description** **0** Base CCA services are not available. **1** Base CCA services are available.
3	56-bit DES availability	A numeric character string containing a value to indicate whether 56-bit DES encryption is available: **Value** **Description** **0** 56-bit DES encryption is not available. **1** 56-bit DES encryption is available.
4	Triple-DES availability	A numeric character string containing a value to indicate whether Triple-DES encryption is available: **Value** **Description** **0** Triple-DES encryption is not available. **1** Triple-DES encryption is available.

Table 13. Cryptographic Facility Query information returned in the rule_array (continued)

Element number	Name	Description
5	SET services availability	A numeric character string containing a value to indicate whether SET (secure electronic transaction) services are available: **Value** **Description** **0** SET services are not available. **1** SET services are available. **Note:** The SET services are not supported in the Linux on z Systems environment.
6	Maximum modulus for symmetric key encryption	A numeric character string containing the maximum modulus size enabled for the encryption of symmetric keys. This defines the longest public-key modulus that can be used for key management of symmetric-algorithm keys.
Output rule_array for option STATICSA This keyword also has verb data returned in the *verb_data* field. See "Verb data returned for CSUACFQ keywords" on page 93.		
1	Card serial number	Eight ASCII characters for the adapter serial number
2	DES new master-key register state	An ASCII number showing the state of the DES new master-key register: **Value** **Description** **1** Empty **2** Partially full **3** Full
3	DES current master-key register state	An ASCII number showing the state of the DES current master-key register: **Value** **Description** **1** Invalid **2** Valid
4	DES old master-key register state	An ASCII number showing the state of the DES old master-key register: **Value** **Description** **1** Invalid **2** Valid
5	PKA new master-key register state	An ASCII number showing the state of the PKA new master-key register: **Value** **Description** **1** Empty **2** Partially full **3** Full
6	PKA current master-key register state	An ASCII number showing the state of the PKA current master-key register: **Value** **Description** **1** Invalid **2** Valid
7	PKA old master-key register state	An ASCII number showing the state of the PKA old master-key register: **Value** **Description** **1** Invalid **2** Valid
8	AES new master-key register state	An ASCII number showing the state of the AES new master-key register: **Value** **Description** **1** Empty **2** Partially full **3** Full

Table 13. Cryptographic Facility Query information returned in the rule_array (continued)

Element number	Name	Description
9	AES current master-key register state	An ASCII number showing the state of the AES current master-key register: **Value** **Description** **1** Invalid **2** Valid
10	AES old master-key register state	An ASCII number showing the state of the AES old master-key register: **Value** **Description** **1** Invalid **2** Valid
Output rule_array for option STATICSB This keyword also has verb data returned in the *verb_data* field. See "Verb data returned for CSUACFQ keywords" on page 93.		
1	Card serial number	Eight ASCII characters for the adapter serial number
2	DES new master-key register state	An ASCII number showing the state of the DES new master-key register: **Value** **Description** **1** Empty **2** Partially full **3** Full
3	DES current master-key register state	An ASCII number showing the state of the DES current master-key register: **Value** **Description** **1** Invalid **2** Valid
4	DES old master-key register state	An ASCII number showing the state of the DES old master-key register: **Value** **Description** **1** Invalid **2** Valid
5	PKA new master-key register state	An ASCII number showing the state of the PKA new master-key register: **Value** **Description** **1** Empty **2** Partially full **3** Full
6	PKA current master-key register state	An ASCII number showing the state of the PKA current master-key register: **Value** **Description** **1** Invalid **2** Valid
7	PKA old master-key register state	An ASCII number showing the state of the PKA old master-key register: **Value** **Description** **1** Invalid **2** Valid
8	APKA new master-key register state	An ASCII number showing the state of the APKA new master-key register: **Value** **Description** **1** Empty **2** Partially full **3** Full

Table 13. Cryptographic Facility Query information returned in the rule_array (continued)

Element number	Name	Description
9	APKA current master-key register state	An ASCII number showing the state of the APKA current master-key register: **Value** **Description** **1** Invalid **2** Valid
10	APKA old master-key register state	An ASCII number showing the state of the APKA old master-key register: **Value** **Description** **1** Invalid **2** Valid
Output rule_array for option STATICSE This keyword also has verb data returned in the *verb_data* field. See "Verb data returned for CSUACFQ keywords" on page 93.		
1	Card serial number	Eight ASCII characters for the adapter serial number
2	DES new master-key register state	An ASCII number showing the state of the DES new master-key register: **Value** **Description** **1** Empty **2** Partially full **3** Full
3	DES current master-key register state	An ASCII number showing the state of the DES current master-key register: **Value** **Description** **1** Invalid **2** Valid
4	DES old master-key register state	An ASCII number showing the state of the DES old master-key register: **Value** **Description** **1** Invalid **2** Valid
5	PKA new master-key register state	An ASCII number showing the state of the PKA new master-key register: **Value** **Description** **1** Empty **2** Partially full **3** Full
6	PKA current master-key register state	An ASCII number showing the state of the PKA current master-key register: **Value** **Description** **1** Invalid **2** Valid
7	PKA old master-key register state	An ASCII number showing the state of the PKA old master-key register: **Value** **Description** **1** Invalid **2** Valid
Output rule_array for option STATICSF		
1	Card serial number	Eight ASCII characters for the adapter serial number

Table 13. Cryptographic Facility Query information returned in the rule_array (continued)

Element number	Name	Description
2	DES new master-key register state	An ASCII number showing the state of the DES new master-ky register: **Value** **Description** **1** Empty **2** Partially full **3** Full
3	DES current master-key register state	An ASCII number showing the state of the DES current master-key register: **Value** **Description** **1** Invalid **2** Valid
4	DES old master-key register state	An ASCII number showing the state of the DES old master-key register: **Value** **Description** **1** Invalid **2** Valid
5	PKA new master-key register state	An ASCII number showing the state of the PKA new master-key register: **Value** **Description** **1** Empty **2** Partially full **3** Full
6	PKA current master-key register state	An ASCII number showing the state of the PKA current master-key register: **Value** **Description** **1** Invalid **2** Valid
7	PKA old master-key register state	An ASCII number showing the state of the PKA old master-key register: **Value** **Description** **1** Invalid **2** Valid
Output rule_array for option STATICSX This keyword also has verb data returned in the *verb_data* field. See "Verb data returned for CSUACFQ keywords" on page 93.		
1	Card serial number	Eight ASCII characters for the adapter serial number
2	DES new master-key register state	An ASCII number showing the state of the DES new master-key register: **Value** **Description** **1** Empty **2** Partially full **3** Full
3	DES current master-key register state	An ASCII number showing the state of the DES current master-key register: **Value** **Description** **1** Invalid **2** Valid
4	DES old master-key register state	An ASCII number showing the state of the DES old master-key register: **Value** **Description** **1** Invalid **2** Valid

Table 13. Cryptographic Facility Query information returned in the rule_array (continued)

Element number	Name	Description
5	PKA new master-key register state	An ASCII number showing the state of the PKA new master-key register: **Value** **Description** **1** Empty **2** Partially full **3** Full
6	PKA current master-key register state	An ASCII number showing the state of the PKA current master-key register: **Value** **Description** **1** Invalid **2** Valid
7	PKA old master-key register state	An ASCII number showing the state of the PKA old master-key register: **Value** **Description** **1** Invalid **2** Valid
Output rule_array for option STATMOFN Elements 1 and 2 are treated as a 16-byte string, as are elements 3 and 4, with the high-order 15 bytes containing meaningful information and the 16th byte containing a space character. Each byte provides status information about the *i*th share, 1 ≤ *i* ≤ 15, of master-key information.		
1, 2	Master-key shares generation	The 15 individual bytes are set to one of these character values: **Value** **Description** **0** Cannot be generated **1** Can be generated **2** Has been generated but not distributed **3** Generated and distributed once **4** Generated and distributed more than once
3, 4	Master-key shares reception	The 15 individual bytes are set to one of these character values: **Value** **Description** **0** Cannot be received **1** Can be received **3** Has been received **4** Has been received more than once
5	*m*	The minimum number of shares required to instantiate a master key through the master-key-shares process. The value is returned in two characters, valued from 01 – 15, followed by six space characters.
6	*n*	The maximum number of distinct shares involved in the master-key shares process. The value is returned in two characters, valued from 01 - 15, followed by six space characters.
Output rule_array for option TIMEDATE		
1	Date	The current date is returned as a character string of the form YYYYMMDD, where: **YYYY** Represents the year. **MM** Represents the month (01 - 12). **DD** Represents the day of the month (01 - 31).
2	Time	The current UTC time of day is returned as a character string of the form HHMMSS, where: **HH** Represents the hour (0 - 23). **MM** Represents the minute (0 - 59). **SS** Represents second (0 - 59).

Table 13. Cryptographic Facility Query information returned in the rule_array (continued)

Element number	Name	Description
3	Day of the week	The day of the week is returned as a number between 1 (Sunday) and 7 (Saturday).
Output rule_array for option TKESTATE		
1	TKE access enabled	Indicates whether a TKE can be used to administer this CEX*C. Values are: **TKEPERM** Allowed **TKEDENY** Not allowed
Output rule_array for option WRAPMTHD		
1	Internal tokens	Default wrapping method for internal tokens. **Value** **Description** **0** Keys are be wrapped with the original method. **1** Keys are be wrapped with the enhanced X9.24 method.
2	External tokens	Default wrapping method for external tokens. **Value** **Description** **0** Keys are wrapped with the original method. **1** Keys are wrapped with the enhanced X9.24 method.

verb_data_length

Direction: Input/Output
Type: Integer

The *verb_data_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *verb-data* variable.

verb_data

Direction: Input/Output
Type: String

A pointer to a string variable containing output data that is returned for some of the keywords specified in the **rule_array** variable. This output is described in a separate section for each keyword in "Verb data returned for CSUACFQ keywords."

Verb data returned for CSUACFQ keywords

Some keywords return specific data in the *verb_data* parameter, and update the *verb_data_length* field with the count of bytes returned.

The *verb_data* buffer must be large enough to receive the data (see keyword-specific sizes below) and the *verb_data_length* parameter as passed in to Cryptographic Facility Query (CSUACFQ) must indicate that size (or a larger value). If either the *verb_data* or *verb_data_length* fields are not valid, no data is returned at all. In this case, a return code of 8 and a reason code of 72 is returned.

GET-UDX

This *rule_array* keyword causes a variable length list of 2-byte UDX identifiers to be returned.

The identifiers represent the authorized UDX verb IDs for the adapter. A UDX is a set of one or more custom CCA APIs added to the adapter, using the installable

code feature. Unless the programming source has also provided an updated host library, these UDX calls are not accessible from the IBM z Systems Linux host library. If an updated host library is provided, refer to the accompanying documentation for usage.

The maximum number of names to be returned is 100. Using this number, the maximum size buffer is 6400 bytes.

NUM-DECT

This *rule_array* keyword causes a 4-byte binary number to be returned.

This is the number of bytes required for the *verb_data* variable when the **STATDECT** rule-array keyword is specified.

SIZEWPIN

For the CSUACFQ verb, the SIZEWPIN and STATWPIN keywords are added to the rule array to query the size and to obtain the state information of the weak PIN table. The SIZEWPIN *rule_array* keyword returns the size of the weak PIN table.

Table 14. Output data format for the SIZEWPIN keyword

Field name	Length in bytes	Description
weak_PIN_table_length	4	Integer value of the number of bytes of data that the **verb_data** requires when the STATWPIN **rule-array** keyword is specified.

STATDECT

This *rule_array* keyword causes a table of up to 100 PIN decimalization tables to be returned.

Table 15 shows the data format for a decimalization table.

Table 15. Output data format for the STATDECT keyword

Field	Length in bytes	Description
number	3	PIN decimalization table identifier in ASCII digits 001 - 100 (X'303031' - X'313030').
state	1	Table state in ASCII: **Code Description** **A (X'41')** Active **L (X'4C')** Loaded
table	16	PIN decimalization table. Contains ASCII digits 0 - 9 (X'30' - X'39'), in the clear.
Total byte count	Depends on the number of returned decimalization tables. There are 20 byte for each decimalization table. The total byte count is returned for the NUM-DECT keyword.	

STATICSA

This *rule_array* keyword causes the indicated master key hash and verification patterns to be returned for the master keys loaded in the current domain.

The status variables for the various master key registers returned in the *rule_array* will indicate which of these verification pattern structures returned contain useful

data. An empty master key register cannot have a meaningful verification pattern. However, the data structures are returned for all registers indicated, so that interpretation is reliable.

The output data format for STATICSA operational key parts is given in Table 16.

Note:

1. The fields will be returned in the order given, however the *_ID fields should be used for verification.
2. The *verb_data_length* parameter will indicate the total size at the bottom of the table describing the *verb_data*.
3. Multiple byte fields are stored in Big-Endian format, as is typical for CEX*C communication.

Table 16. Output data format for STATICSA operational key parts

Field name	Length in bytes	Field value	Description
SYM_OMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
SYM_OMK_MDC4_ID	2	X'0F02'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
SYM_OMK_MDC4_HP	16	variable	Hash pattern over the Symmetric Key old master-key register, calculated using the MDC4 algorithm.
SYM_CMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
SYM_CMK_MDC4_ID	2	X'0F01'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
SYM_CMK_MDC4_HP	16	variable	Hash pattern over the Symmetric Key current master-key register, calculated using the MDC4 algorithm.
SYM_NMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
SYM_NMK_MDC4_ID	2	X'0F00'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
SYM_NMK_MDC4_HP	16	variable	Hash pattern over the Symmetric Key new master-key register, calculated using the MDC4 algorithm.
ASYM_OMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
ASYM_OMK_MDC4_ID	2	X'0F05'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
ASYM_OMK_MDC4_HP	16	variable	Hash pattern over the Asymmetric Key old master-key register, calculated using the MDC4 algorithm.
ASYM_CMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
ASYM_CMK_MDC4_ID	2	X'0F04'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
ASYM_CMK_MDC4_HP	16	variable	Hash pattern over the Asymmetric Key current master-key register, calculated using the MDC4 algorithm.

Table 16. Output data format for STATICSA operational key parts (continued)

Field name	Length in bytes	Field value	Description
ASYM_NMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
ASYM_NMK_MDC4_ID	2	X'0F03'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
ASYM_NMK_MDC4_HP	16	variable	Hash pattern over the Asymmetric Key new master-key register, calculated using the MDC4 algorithm.
SYM_OMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern).
SYM_OMK_VP_ID	2	X'0F08'	Hexadecimal identifier indicating the contents of the following Verification pattern field.
SYM_OMK_VP	8	variable	Verification pattern over the Symmetric Key old master-key register calculated using the default algorithm.
SYM_CMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern).
SYM_CMK_VP_ID	2	X'0F07'	Hexadecimal identifier indicating the contents of the following Verification pattern field.
SYM_CMK_VP	8	variable	Verification pattern over the Symmetric Key current master-key register, calculated using the default algorithm.
SYM_NMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern).
SYM_NMK_VP_ID	2	X'0F06'	Hexadecimal identifier indicating the contents of the following Verification pattern field.
SYM_NMK_VP	8	variable	Verification pattern over the Symmetric Key new master-key register, calculated using the default algorithm.
SYM_NMK_MKAP_LEN	2	12	Length in bytes of this Authentication pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Authentication pattern).
SYM_NMK_MKAP_ID	2	X'0F09'	Hexadecimal identifier indicating the contents of the following Authentication pattern field.
SYM_NMK_MKAP	8	variable	Authentication pattern over the Symmetric Key new master-key register, calculated using the ICSF specified algorithm.
AES_OMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern).
AES_OMK_VP_ID	2	X'0F0C'	Hexadecimal identifier indicating the contents of the following Verification pattern field.
AES_OMK_VP	8	variable	Verification pattern over the AES Key old master-key register, calculated using the SHA-256 algorithm.
AES_CMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern).
AES_CMK_VP_ID	2	X'0F0B'	Hexadecimal identifier indicating the contents of the following Verification pattern field.

Table 16. Output data format for STATICSA operational key parts (continued)

Field name	Length in bytes	Field value	Description
AES_CMK_VP	8	variable	Verification pattern over the AES Key current master-key register calculated using the SHA-256 algorithm.
AES_NMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern).
AES_NMK_VP_ID	2	X'0F0A'	Hexadecimal identifier indicating the contents of the following Verification pattern field.
AES_NMK_VP	8	variable	Verification pattern over the AES Key new master-key register, calculated using the SHA-256 algorithm.
Total byte count	204		

STATICSB

This *rule_array* keyword causes the indicated master key hash and verification patterns to be returned for the master keys loaded in the current domain.

The status variables for the various master-key registers returned in the *rule_array* will indicate which of these verification pattern structures returned contain useful data. An empty master-key register cannot have a meaningful verification pattern. However, the data structures are returned for all registers indicated, so that interpretation is reliable.

The output data format for STATICSB operational key parts is given in Table 17.

Note:

1. The fields will be returned in the order given, however the *_ID fields should be used for verification.
2. The *verb_data_length* parameter will indicate the total size at the bottom of the table describing the *verb_data*.
3. Multiple byte fields are stored in Big-Endian format, as is typical for CEX*C communication.

Table 17. Output data format for STATICSB operational key parts

Field name	Length in bytes	Field value	Description
SYM_OMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
SYM_OMK_MDC4_ID	2	X'0F02'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
SYM_OMK_MDC4_HP	16	variable	Hash pattern over the Symmetric Key old master-key register, calculated using the MDC4 algorithm.
SYM_CMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
SYM_CMK_MDC4_ID	2	X'0F01'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
SYM_CMK_MDC4_HP	16	variable	Hash pattern over the Symmetric Key current master-key register, calculated using the MDC4 algorithm.

Table 17. Output data format for STATICSB operational key parts (continued)

Field name	Length in bytes	Field value	Description
SYM_NMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
SYM_NMK_MDC4_ID	2	X'0F00'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
SYM_NMK_MDC4_HP	16	variable	Hash pattern over the Symmetric Key new master-key register, calculated using the MDC4 algorithm.
ASYM_OMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
ASYM_OMK_MDC4_ID	2	X'0F05'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
ASYM_OMK_MDC4_HP	16	variable	Hash pattern over the Asymmetric Key old master-key register, calculated using the MDC4 algorithm.
ASYM_CMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
ASYM_CMK_MDC4_ID	2	X'0F04'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
ASYM_CMK_MDC4_HP	16	variable	Hash pattern over the Asymmetric Key current master-key register, calculated using the MDC4 algorithm.
ASYM_NMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
ASYM_NMK_MDC4_ID	2	X'0F03'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
ASYM_NMK_MDC4_HP	16	variable	Hash pattern over the Asymmetric Key new master-key register, calculated using the MDC4 algorithm.
SYM_OMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern).
SYM_OMK_VP_ID	2	X'0F08'	Hexadecimal identifier indicating the contents of the following Verification pattern field.
SYM_OMK_VP	8	variable	Verification pattern over the Symmetric Key old master-key register calculated using the default algorithm.
SYM_CMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern).
SYM_CMK_VP_ID	2	X'0F07'	Hexadecimal identifier indicating the contents of the following Verification pattern field.
SYM_CMK_VP	8	variable	Verification pattern over the Symmetric Key current master-key register, calculated using the default algorithm.
SYM_NMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern).
SYM_NMK_VP_ID	2	X'0F06'	Hexadecimal identifier indicating the contents of the following Verification pattern field.

Table 17. Output data format for STATICSB operational key parts (continued)

Field name	Length in bytes	Field value	Description
SYM_NMK_VP	8	variable	Verification pattern over the Symmetric Key new master-key register, calculated using the default algorithm.
SYM_NMK_MKAP_LEN	2	12	Length in bytes of this Authentication pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Authentication pattern).
SYM_NMK_MKAP_ID	2	X'0F09'	Hexadecimal identifier indicating the contents of the following Authentication pattern field.
SYM_NMK_MKAP	8	variable	Authentication pattern over the Symmetric Key new master-key register, calculated using the ICSF specified algorithm.
AES_OMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern)
AES_OMK_VP_ID	2	X'0F0C'	Hexadecimal identifier indicating the contents of the following Verification pattern field.
AES_OMK_VP	8	variable	Verification pattern over the Symmetric Key old master-key register, calculated using the SHA-256 algorithm.
AES_CMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern).
AES_CMK_VP_ID	2	X'0F0B'	Hexadecimal identifier indicating the contents of the following Verification pattern field.
AES_CMK_VP	8	variable	Verification pattern over the Symmetric Key current master-key register, calculated using the SHA-256 algorithm.
AES_NMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Authentication pattern).
AES_NMK_VP_ID	2	X'0F0A'	Hexadecimal identifier indicating the contents of the following Verification pattern field.
AES_NMK_VP	8	variable	Verification pattern over the Symmetric Key new master-key register, calculated using the SHA-256 algorithm.
APKA_OMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern)
APKA_OMK_VP_ID	2	X'0F0F'	Hexadecimal identifier indicating the contents of the following Verification pattern field.
APKA_OMK_VP	8	variable	Verification pattern over the Symmetric Key old master-key register, calculated using the SHA-256 algorithm.
APKA_CMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern).
APKA_CMK_VP_ID	2	X'0F0E'	Hexadecimal identifier indicating the contents of the following Verification pattern field.
APKA_CMK_VP	8	variable	Verification pattern over the Symmetric Key current master-key register, calculated using the SHA-256 algorithm.
APKA_NMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Authentication pattern).

Table 17. Output data format for STATICSB operational key parts (continued)

Field name	Length in bytes	Field value	Description
APKA_NMK_VP_ID	2	X'0F0D'	Hexadecimal identifier indicating the contents of the following Verification pattern field.
APKA_NMK_VP	8	variable	Verification pattern over the Symmetric Key new master-key register, calculated using the SHA-256 algorithm.
Total byte count	240		

This keyword was introduced with CCA 4.1.0.

STATICSE

This *rule_array* keyword returns the indicated master key hash and verification patterns for the master keys loaded in the current domain.

The status variables for the various master-key registers returned in the *rule_array* will indicate which of these verification pattern structures returned contain useful data. An empty master-key register cannot have a meaningful verification pattern. However, the data structures are returned for all registers indicated, so that interpretation is reliable.

The output data format for STATICSE operational key parts is given in Table 18.

Note:

1. The fields will be returned in the order given, however the *_ID fields should be used for verification.
2. The *verb_data_length* parameter will indicate the total size at the bottom of the table describing the *verb_data*.
3. Multiple byte fields are stored in Big-Endian format, as is typical for CEX*C communication.

Table 18. Output data format for STATICSE operational key parts

Field name	Length in bytes	Field value	Description
SYM_OMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
SYM_OMK_MDC4_ID	2	X'0F02'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
SYM_OMK_MDC4_HP	16	variable	Hash pattern over the Symmetric Key old master-key register, calculated using the MDC4 algorithm.
SYM_CMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
SYM_CMK_MDC4_ID	2	X'0F01'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
SYM_CMK_MDC4_HP	16	variable	Hash pattern over the Symmetric Key current master-key register, calculated using the MDC4 algorithm.
SYM_NMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).

Table 18. Output data format for STATICSE operational key parts (continued)

Field name	Length in bytes	Field value	Description
SYM_NMK_MDC4_ID	2	X'0F00'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
SYM_NMK_MDC4_HP	16	variable	Hash pattern over the Symmetric Key new master-key register, calculated using the MDC4 algorithm.
ASYM_OMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
ASYM_OMK_MDC4_ID	2	X'0F05'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
ASYM_OMK_MDC4_HP	16	variable	Hash pattern over the Asymmetric Key old master-key register, calculated using the MDC4 algorithm.
ASYM_CMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
ASYM_CMK_MDC4_ID	2	X'0F04'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
ASYM_CMK_MDC4_HP	16	variable	Hash pattern over the Asymmetric Key current master-key register, calculated using the MDC4 algorithm.
ASYM_NMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
ASYM_NMK_MDC4_ID	2	X'0F03'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
ASYM_NMK_MDC4_HP	16	variable	Hash pattern over the Asymmetric Key new master-key register, calculated using the MDC4 algorithm.
SYM_OMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern).
SYM_OMK_VP_ID	2	X'0F08'	Hexadecimal identifier indicating the contents of the following Verification pattern field.
SYM_OMK_VP	8	variable	Verification pattern over the Symmetric Key old master-key register calculated using the default algorithm.
SYM_CMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern).
SYM_CMK_VP_ID	2	X'0F07'	Hexadecimal identifier indicating the contents of the following Verification pattern field.
SYM_CMK_VP	8	variable	Verification pattern over the Symmetric Key current master-key register, calculated using the default algorithm.
SYM_NMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern).
SYM_NMK_VP_ID	2	X'0F06'	Hexadecimal identifier indicating the contents of the following Verification pattern field.
SYM_NMK_VP	8	variable	Verification pattern over the Symmetric Key new master-key register, calculated using the default algorithm.

Table 18. Output data format for STATICSE operational key parts (continued)

Field name	Length in bytes	Field value	Description
SYM_NMK_MKAP_LEN	2	12	Length in bytes of this Authentication pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Authentication pattern).
SYM_NMK_MKAP_ID	2	X'0F09'	Hexadecimal identifier indicating the contents of the following Authentication pattern field.
SYM_NMK_MKAP	8	variable	Authentication pattern over the Symmetric Key new master-key register, calculated using the ICSF specified algorithm.
Total byte count	168		

STATICSX

This *rule_array* keyword returns the indicated master key hash and verification patterns for the master keys loaded in the current domain.

The status variables for the various master key registers returned in the *rule_array* will indicate which of these verification pattern structures returned contain useful data. An empty master key register cannot have a meaningful verification pattern. However, the data structures are returned for all registers indicated, so that interpretation is reliable.

The output data format for STATICSX operational key parts is given in Table 19.

Note:

1. The fields will be returned in the order given, however the *_ID fields should be used for verification.
2. The *verb_data_length* parameter will indicate the total size at the bottom of the table describing the *verb_data*.
3. Multiple byte fields are stored in Big-Endian format, as is typical for CEX*C communication.

Table 19. Output data format for STATICSX operational key parts

Field name	Length in bytes	Field value	Description
SYM_OMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
SYM_OMK_MDC4_ID	2	X'0F02'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
SYM_OMK_MDC4_HP	16	variable	Hash pattern over the Symmetric Key old master-key register, calculated using the MDC4 algorithm.
SYM_CMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
SYM_CMK_MDC4_ID	2	X'0F01'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
SYM_CMK_MDC4_HP	16	variable	Hash pattern over the Symmetric Key current master-key register, calculated using the MDC4 algorithm.

Table 19. Output data format for STATICSX operational key parts (continued)

Field name	Length in bytes	Field value	Description
SYM_NMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
SYM_NMK_MDC4_ID	2	X'0F00'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
SYM_NMK_MDC4_HP	16	variable	Hash pattern over the Symmetric Key new master-key register, calculated using the MDC4 algorithm.
ASYM_OMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
ASYM_OMK_MDC4_ID	2	X'0F05'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
ASYM_OMK_MDC4_HP	16	variable	Hash pattern over the Asymmetric Key old master-key register, calculated using the MDC4 algorithm.
ASYM_CMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
ASYM_CMK_MDC4_ID	2	X'0F04'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
ASYM_CMK_MDC4_HP	16	variable	Hash pattern over the Asymmetric Key current master-key register, calculated using the MDC4 algorithm.
ASYM_NMK_MDC4_LEN	2	20	Length in bytes of this Hash pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Hash pattern).
ASYM_NMK_MDC4_ID	2	X'0F03'	Hexadecimal identifier indicating the contents of the following Hash pattern field.
ASYM_NMK_MDC4_HP	16	variable	Hash pattern over the Asymmetric Key new master-key register, calculated using the MDC4 algorithm.
SYM_OMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern).
SYM_OMK_VP_ID	2	X'0F08'	Hexadecimal identifier indicating the contents of the following Verification pattern field.
SYM_OMK_VP	8	variable	Verification pattern over the Symmetric Key old master-key register calculated using the default algorithm.
SYM_CMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern).
SYM_CMK_VP_ID	2	X'0F07'	Hexadecimal identifier indicating the contents of the following Verification pattern field.
SYM_CMK_VP	8	variable	Verification pattern over the Symmetric Key current master-key register, calculated using the default algorithm.
SYM_NMK_VP_LEN	2	12	Length in bytes of this Verification pattern block in the *verb_data* (comprising this length field, the following ID field, and the field for the Verification pattern).
SYM_NMK_VP_ID	2	X'0F06'	Hexadecimal identifier indicating the contents of the following Verification pattern field.

Table 19. Output data format for STATICSX operational key parts (continued)

Field name	Length in bytes	Field value	Description
SYM_NMK_VP	8	variable	Verification pattern over the Symmetric Key new master-key register, calculated using the default algorithm.
Total byte count	156		

STATKPR

This keyword returns non-secret information about a particular named operational key part loaded by the TKE to the user.

The structures for various key types are given under section *STATKPR output data*. An appropriate name for an existing operational key part is expected to be provided as described in section *STATKPR input data*. If not, the error return code of 8 and a reason code of 1026 is be returned, meaning key name not found.

STATKPRL input data:

A 64-byte key name must be provided in the *verb_data* field, while the *verb_data_length* must be set to a value of 64.

The operational key name must exactly match the name returned by a call to STATKPRL.

STATKPR output data:

Note:

1. The fields are returned in the order given.
2. Output data overwrites the input data in the *verb_data* field, and set the *verb_data_length* field to the output value.
3. The *verb_data_length* parameter indicates the total size, at the bottom of the table describing the *verb_data*.

   Notice that the output data is smaller than the input data.
4. Multiple byte fields are stored in Big-Endian format, as is typical for CEX*C communication.

Table 20. Output data format for STATKPR operational key parts

Field name	Length in bytes	Description
state	1	State of the key part register: **Value** **Description** **X'00'** The register is empty. **X'01'** The first DES key part was entered for the named key into this register. **X'02'** An intermediate DES key part (part after first) has been entered. **X'03'** The register contains a completed DES key. **X'11'** The first AES key part was entered for the named key into this register. **X'12'** An intermediate AES key part (part after first) has been entered. **X'13'** The register contains a completed AES key.
reserved	1	Will have a value of X'00'.
key_length	1	Length of key in bytes. For DES keys, values are: 8, 16, 24. For AES keys, values are: 16, 24, 32.

Table 20. Output data format for STATKPR operational key parts (continued)

Field name	Length in bytes	Description
cv_length	1	Length of Control Vector (CV) for key part, in bytes. The value is 8 or 16, indicating how much of the CV field to use. Note that CV is NOT a variable length field.
cv	16	Control Vector for the operational key part.
reserved_2	8	Has a value of X'00' for the entire length.
key_part_hash	20	Hash over the key stored in the key part register. For DES keys, the hash algorithm is SHA-1. For AES keys, the hash algorithm is SHA-256.
ver_pattern	4	Verification pattern over the key calculated using the default algorithm.
Total byte count	52	

STATKPRL

This keyword returns a list of the names of all operational key parts that have been loaded by the TKE into the CEX*C.

Each name has a length of 64 bytes. The maximum number of key slots (and thus the count of labels returned) depends on the firmware loaded to the adapter. This count is currently set to 100. If not enough space has been provided (using the *verb_data_length* field passed in by the application) to return the available list, a return code of 8 and a reason code of 72 is returned.

STATVKPL

This keyword returns a list of the names of all operational key parts that have been loaded by the TKE into the CEX*C for preparation of variable length key tokens.

This function is different from STATKPRL, which describes the list of 64-byte legacy style tokens in preparation.

Each name has a length of 64 bytes. The maximum number of key slots (and thus the count of labels returned) depends on the firmware loaded to the adapter. This count is currently set to 100. If not enough space has been provided (using the *verb_data_length* field passed in by the application) to return the available list, a return code of 8 and a reason code of 72 is returned.

STATVKPR

This keyword cause non-secret information about a particular named operational key part loaded by the TKE to returned to the user.

This is different from STATKPR in that a register for creating a key in a variable length key token is described. The structures for various key types are given in section *STATVKPR output data*. An appropriate name for an existing operational key part is expected to be provided as described in *STATVKPL input data*. If not, the error return code of 8 and a reason code of 1026 is returned, meaning that the key name is not found.

STATVKPL input data:

A 64 byte key name must be provided in the *verb_data* field, while the *verb_data_length* must be set to 64.

The operational key name must match exactly the name returned by a call to STATVKPL.

STATVKPR output data:

See Table 21 for the output data format.

Note:

1. The fields are returned in the order given.
2. Output data overwrites the input data in the *verb_data* field, and set the *verb_data_length* field to the output value.
3. The *verb_data_length* parameter indicates the total size, at the bottom of the table describing the *verb_data*.

   Note that the output data is smaller than the input data.
4. Multiple byte fields are stored in Big-Endian format, as is typical for CEX*C communication.

Table 21. Output data format for STATVKPR operational key parts

Field name	Length in bytes	Description
version	1	Version of the structure
state	1	State of the key part register: **Value** **Description** **X'00'** The register is empty. **X'01'** The first DES key part was entered for the named key into this register. **X'02'** An intermediate DES key part (part after first) has been entered. **X'03'** The register contains a completed DES key. **X'11'** The first AES key part was entered for the named key into this register. **X'12'** An intermediate AES key part (part after first) has been entered. **X'13'** The register contains a completed AES key. **X'21'** The first AES key part for variable length token was entered for the named key into this register. **X'22'** An intermediate AES key part for variable length token (part after first) has been entered. **X'23'** The register contains a completed AES key part for variable length token.
key_length	1	Length of key in bytes. For DES keys, values are: 8, 16, 24. For AES keys, values are: 16, 24, 32.
key_completeness	1	Number of parts needed to complete key **X'C0'** Two parts are needed. **X'80'** One part is needed. **X'40'** No parts are needed. Key is complete.
ver_pattern	4	ENC-ZERO method calculated verification pattern of the key.
key_part_hash	8	Hash using the SHA-256 algorithm over the key that is currently stored, at the current level of completeness.
skel_length	2	Skeleton token length.
pad	2	Pad structure to 4-byte boundary.
skel	384	Stored key token skeleton, which will hold completed key when operation is complete. No key material is stored or returned here.
reserved2	108	Extra bytes.
Total byte count	512	

STATWPIN

For the CSUACFQ verb, the SIZEWPIN and STATWPIN keywords are added to the rule array to query the size and to obtain the state information of the weak PIN table. The STATWPIN *rule_array* keyword obtains the state information of the weak PIN table.

Table 22. Output data format for the STATWPIN keyword

Field name	Length in bytes	Description
weak_PIN_table_state	11 - 460	Returns the WPIN entry structure as described in Table 23.

Table 23. CSUACFQ weak PIN entry structure (type X'30')

Offset	Length in bytes	Description
0	01	Weak PIN structure type (in ASCII): X'30' - weak PIN entry structure
1	03	Weak PIN entry identifier in ASCII digits 001 - 020 (X'303031' - X'303230')
4	01	State of entry (in ASCII): **Value Meaning** **A (X'41')** Active **L (X'4C')** Loaded
5	02	Clear PIN length (in ASCII digits). For DK verbs, valid values are 04 - 12 (X'3034' - X'3132'). Otherwise, valid values are 04 - 16 (X'3034' - X3136').
7	04 — 16	Clear weak PIN (in ASCII digits). Valid values are 0 - 9 (X'30' - X'39').

Restrictions

You cannot limit the number of returned *rule_array* elements.

Table 13 on page 81 describes the number and meaning of the information in output *rule_array* elements.

Tip: Allocate a minimum of 30 *rule_array* elements to allow for extensions of the returned information.

Required commands

The CSUACFQ required commands.

None.

Usage notes

Usage notes for CSUACFQ.

This verb is not impacted by the AUTOSELECT option. See “Verbs that ignore AUTOSELECT” on page 10 for more information.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSUACFQJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSUACFQJ(
           hikmNativeInteger   return_code,
           hikmNativeInteger   reason_code,
           hikmNativeInteger   exit_data_length,
           byte[]              exit_data,
           hikmNativeInteger   rule_array_count,
           byte[]              rule_array,
           hikmNativeInteger   verb_data_length,
           byte[]              verb_data
);
```

Cryptographic Facility Version (CSUACFV)

The Cryptographic Facility Version verb is used to retrieve information about the Security Application Program Interface (SAPI) Version and the Security Application Program Interface build date.

In the same format as the Cryptographic Facility Query (CSUACFQ) verb returns for the CCA application with the STATCCA *rule_array* option.

This verb returns information elements in the *version_data* variable.

Format

The format of CSUACFV.

```
CSUACFV(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    version_data_length,
    version_data)
```

Parameters

The parameter definitions for CSUACFV.

Note that there is no **rule_array** keyword.

For the definitions of the **return_code**, **reason_code**, **exit_data_length**, and **exit_data** parameters, see "Parameters common to all verbs" on page 20.

version_data_length

Direction: Input/Output
Type: Integer

The **version_data_length** parameter is a pointer to an integer variable containing the number of bytes in the version data variable. This value must be a minimum of 17 bytes. On input, the **version_data_length** variable must be set to the total size of the variable pointed to by the **version_data** parameter. On output, this variable contains the number of bytes of data returned by the verb in the **version_data** variable.

version_data

Direction: Output
Type: String

The **version_data** parameter is a pointer to a string variable containing data returned by the verb. An 8-byte character string identifies the version of the Security Application Program Interface (SAPI) library, followed by an 8-byte character string containing the build date for the SAPI library, followed by a null terminating character. The build date is in the format: *yyyymmdd*, where *yyyy* is the year, *mm* is the month, and *dd* is the day of the month.

Restrictions

The restrictions for CSUACFV.

None.

Required commands

The CSUACFV required commands.

None.

Usage notes

Usage notes for CSUACFV.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSUACFVJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSUACFVJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger version_data_length,
           byte[]            version_data
);
```

Cryptographic Resource Allocate (CSUACRA)

The Cryptographic Resource Allocate verb is used to allocate a specific CCA coprocessor for use by the thread or process, depending on the scope of the verb.

This verb is scoped to a thread. When a thread or process, depending on the scope, allocates a cryptographic resource, requests are routed to that resource. When a cryptographic resource is not allocated, requests are routed to the default cryptographic resource.

You can set the default cryptographic resource. If you take no action, the default assignment is CRP01.

You cannot allocate a cryptographic resource while one is already allocated. Use the Cryptographic Resource Deallocate verb (see "Cryptographic Resource Deallocate (CSUACRD)" on page 112) to deallocate an allocated cryptographic resource.

Be sure to review "Multi-coprocessor selection capabilities" on page 9.

Format

The format of CSUACRA.

```
CSUACRA(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    resource_name_length,
    resource_name)
```

Parameters

The parameter definitions for CSUACRA.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1.

rule_array

Direction: Input
Type: String array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* keyword is described in Table 24.

Table 24. Keywords for Cryptographic Resource Allocate control information

Keyword	Description
Cryptographic resource (Required)	
DEVICE	Specifies a CEX*C coprocessor.
DEV-ANY	Specifies to enable the AUTOSELECT option, such that the operating system may select the CCA coprocessor to be used from the available resources according to its policy. This selection applies to most verbs, but not all. See "Multi-coprocessor selection capabilities" on page 9 for more information.
HCPUACLR	Specifies the use of host CPU assist for clear keys. This keyword enables clear key use of the CPACF, for clear key AES encryption and decryption with hash algorithms: SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512. This is the default state at the time of the first use of the CCA library by a PID or TID.
HCPUAPRT	Specifies the use of host CPU assist for protected keys. This keyword enables protected key use of the CPACF for protected key AES and DES, TDES, and MAC. This is *not* the default state at the time of the first use of the CCA library by a PID or TID.

There is an environment variable that also impacts the default card: CSU_DEFAULT_ADAPTER (see "Multi-coprocessor selection capabilities" on page 9). There are also environment variables that influence CPACF support (see "Environment variables that affect CPACF usage" on page 12).

The actual hardware configuration determines what features are available, and CCA uses what exists if the user sets these values as desired, with respect to appropriate defaults.

resource_name_length

Direction: Input
Type: Integer

The *resource_name_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *resource-name* variable. The length must be 1 - 64.

resource_name

Direction: Input
Type: String

The *resource_name* parameter is a pointer to a string variable containing the name of the coprocessor to be allocated.

Restrictions

Restrictions for CSUACRA.

None.

Required commands

The CSUACRA required commands.

None.

Usage notes

Usage notes for CSUACRA.

For optimal performance, ensure that you have enabled CPACF in the thread doing the processing, by making a quick call on the host side at thread startup time, to Cryptographic Resource Allocate, specifying the correct HCPUACLR and HCPUAPRT keyword values for your operation. See the Cryptographic Resource Allocate **rule_array** keyword definitions, and see "Access control points that affect CPACF protected key operations" on page 13 for more affected verbs. Note that it is the user's responsibility to ensure that all adapters accessible to the operating system use the DEV-ANY state. Further use of DEV-ANY as a target adapter for verbs that modify the state of an adapter (such as the generation of retained keys) can lead to unexpected results. See "Multi-coprocessor selection capabilities" on page 9 for more information.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSUACRAJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSUACRAJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger resource_name_length,
           byte[]            resource_name);
```

Cryptographic Resource Deallocate (CSUACRD)

The Cryptographic Resource Deallocate verb is used to deallocate a specific CCA coprocessor that is allocated by the thread or process, depending on the scope of the verb.

This verb is scoped to a thread. When a thread or process, depending on the scope, de-allocates a cryptographic resource, requests are routed to the default cryptographic resource.

You can set the default cryptographic resource. If you take no action, the default assignment is CRP01.

If a thread with an allocated coprocessor ends without first de-allocating the coprocessor, excess memory consumption results. It is not necessary to deallocate a cryptographic resource if the process itself is ending, only if individual threads end while the process continues to run.

Be sure to review “Multi-coprocessor selection capabilities” on page 9.

Format

The format of CSUACRD.

```
CSUACRD(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count
    rule_array,
    resource_name_length,
    resource_name)
```

Parameters

The parameter definitions for CSUACRD.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters common to all verbs” on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1.

rule_array

Direction: Input
Type: String array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* keyword is described in Table 25.

Table 25. Keywords for Cryptographic Resource Deallocate control information

Keyword	Description
Cryptographic resource (Required)	
DEVICE	Specifies a CEX*C coprocessor.
DEV-ANY	Specifies to disable the AUTOSELECT option. Verbs will now all use the default adapter or a previously configured "selected" adapter as chosen via CSUACRA. See “Multi-coprocessor selection capabilities” on page 9 for more information.
HCPUACLR	Specifies the use of host CPU assist for clear keys. This keyword enables clear key use of the CPACF, for clear key AES encryption and decryption with hash algorithms: SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512. This is the default state at the time of the first use of the CCA library by a PID or TID.
HCPUAPRT	Specifies the use of host CPU assist for protected keys. This keyword disables protected key use of the CPACF for protected key AES and DES, TDES, and MAC. This is the default state at the time of the first use of the CCA library by a PID or TID.

There is an environment variable that also impacts the default card: CSU_DEFAULT_ADAPTER (see “Multi-coprocessor selection capabilities” on page 9). There are also environment variables that influence CPACF support (see “Environment variables that affect CPACF usage” on page 12).

The actual hardware configuration determines what features are available, and CCA uses what exists if the user sets these values as desired, with respect to appropriate defaults.

resource_name_length

Direction: Input
Type: Integer

The *resource_name_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *resource_name* variable. The length must be 1 - 64.

resource_name

Direction: Input
Type: String

The *resource_name* parameter is a pointer to a string variable containing the name of the coprocessor to be deallocated.

Restrictions

The restrictions for CSUACRD.

None.

Required commands

The CSUACRD required commands.

None.

Usage notes

Usage notes for CSUACRD.

To disable CPACF usage in your processing thread, make a call to Cryptographic Resource Deallocate, specifying the correct HCPUACLR and HCPUAPRT keyword as appropriate. See the Cryptographic Resource Deallocate *rule_array* keyword definitions, and see "Access control points that affect CPACF protected key operations" on page 13 for more affected verbs.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSUACRDJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSUACRDJ(
          hikmNativeInteger return_code,
          hikmNativeInteger reason_code,
          hikmNativeInteger exit_data_length,
          byte[]            exit_data,
          hikmNativeInteger rule_array_count,
          byte[]            rule_array,
          hikmNativeInteger resource_name_length,
          byte[]            resource_name);
```

Key Storage Initialization (CSNBKSI)

The Key Storage Initialization verb initializes a key-storage file using the current symmetric or asymmetric master-key.

The initialized key storage file does not contain any preexisting key records. The key storage data and index files are in the `/opt/IBM/CCA/keys` directory.

The key storage functions do not work with HMAC keys.

Format

The format of CSNBKSI.

```
CSNBKSI(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_storage_file_name_length,
    key_storage_file_name,
    key_storage_description_length,
    key_storage_description,
    clear_master_key)
```

Parameters

The parameter definitions for CSNBKSI.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 2.

rule_array

Direction: Input
Type: String array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* keywords are described in Table 26.

Table 26. Keywords for Key Storage Initialization control information

Keyword	Description
Master-key source (Required)	
CURRENT	Specifies the current symmetric master-key of the default cryptographic facility is to be used for the initialization.
Key-storage selection (One required)	
AES	Initialize AES key storage.
DES	Initialize DES key storage.
PKA	Initialize PKA key storage (PKA and, beginning with Release 4.1.0, ECC key tokens).

key_storage_file_name_length

Direction: Input
Type: Integer

The *key_storage_file_name_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *key_storage_file_name* variable. The length must be within the range of 1 - 64.

key_storage_file_name

Direction: Input
Type: String

The *key_storage_file_name* parameter is a pointer to a string variable containing the fully qualified file name of the key-storage file to be initialized. If the file does not exist, it is created. If the file does exist, it is overwritten and all existing keys are lost.

key_storage_description_length

Direction: Input
Type: Integer

The *key_storage_description_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *key_storage_description* variable.

key_storage_description

Direction: Input
Type: String

The *key_storage_description* parameter is a pointer to a string variable containing the description string stored in the key-storage file when it is initialized.

clear_master_key

Direction: Input
Type: String

The *clear_master_key* parameter is unused, but it must be declared and point to 24 data bytes in application storage.

Restrictions

The restrictions for CSNBKSI.

ECC and variable-length symmetric key tokens are not supported in releases before Release 4.1.0.

Required commands

The CSNBKSI required commands.

In order to access key storage, the Key Storage Initialization verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

Usage notes for CSNBKSI.

Using the Key Storage Initialization verb to initialize DES (symmetric) key storage currently requires both the symmetric master key and the asymmetric master key to be loaded, completed and set using the Master Key Process (CSNBMKP) verb or suitable utility, or an interface tool that uses the Master Key Process (CSNBMKP) verb.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKSIJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBKSIJ(
          hikmNativeInteger return_code,
          hikmNativeInteger reason_code,
          hikmNativeInteger exit_data_length,
          byte[]            exit_data,
          hikmNativeInteger rule_array_count,
          byte[]            rule_array,
          hikmNativeInteger filename_length,
          byte[]            filename,
```

```
hikmNativeInteger key_storage_description_length,
byte[]            key_storage_description,
byte[]            clear_master_key);
```

Log Query (CSUALGQ)

Use the Log Query verb to retrieve system log (syslog) message data and CCA log message data from the coprocessor. Syslog data is available for one of the five latest boot cycles (current boot cycle and up to four previous boot cycles). CCA log message data is optionally available during the current boot cycle. The verb supports service personnel and developers in testing and debugging issues.

The syslog for a boot cycle is initially populated with operating system messages and other startup messages, but more messages are written as CCA encounters errors at log level 12 or above. A log level of 12 means, that if an error at return code value 12 or higher occurs during the processing of an operation or during general system processing, then a descriptive message is sent to the syslog. The log level is configurable, with a default value of 12. If configured to be logged, messages for return code 4 and return code 8 are saved in volatile memory (CCA log) separately from messages in the syslog. This keeps these messages from overwhelming more important errors and system messages. Each new boot cycle wipes out any previous CCA log, while the current syslog and up to four previous boot cycle syslogs are maintained. The CCA log is a circular log. When the log is full, the oldest messages are replaced as needed with any new messages.

Format

The format of CSUALGQ.

```
CSUALGQ(
                  return_code,
                  reason_code,
                  exit_data_length,
                  exit_data,
                  rule_array_count,
                  rule_array,
                  log_number_or_level,
                  reserved0,
                  log_data_length,
                  log_data,
                  reserved1_length,
                  reserved1,
                  reserved2_length,
                  reserved2)
```

Parameters

The parameter definitions for CSUALGQ.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. The value must be 1.

rule_array

Direction: Input
Type: String array

A pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The **rule_array** keywords are described in Table 27.

Table 27. Keywords for Log Query control information.

Keyword	Description
Service requested (One required)	
CCALGLEN	Specifies to return the length of the CCA log data buffer in the **log_data_length** variable (the **log_number_or_level** parameter is ignored). No data is returned in the **log_data** variable.
GETCCALG	Specifies to get up to **log_data_length** bytes of data from the current CCA log data buffer (the **log_number_or_level** parameter is ignored). This data is returned in the **log_data** variable. The returned log only contains all available CCA messages that had a return code of 4 or 8. All other messages are filtered out. **Note:** If the log level (see **SETLGLVL** keyword) has never been set to 4 or 8 in the current boot cycle for the coprocessor, then the CCA log is empty and no data is available to return.
GETLGLVL	Specifies to return the current log level in the **log_number_or_level** variable. The log level is used to determine which return codes the coprocessor uses to generate a log message. A return code greater than or equal to this level causes a log message to be generated. No data is returned in the **log_data** variable.
GETSYSLG	Specifies to get up to **log_data_length** bytes of data from the syslog (for the given log number as specified by the **log_number_or_level** parameter). This data is returned in the **log_data** variable. The returned log contains all available syslog messages, including boot, major error, and CCA messages.
SETLGLVL	Specifies to set the current log level inside the coprocessor to the value specified by the **log_number_or_level** parameter (4, 8, or 12). A CCA return code greater than or equal to the log level causes a log message to be generated. No data is returned in the **log_data** variable. If the log level for the coprocessor has never been set in the current boot cycle, it defaults to 12. Where the coprocessor stores a message, depends on the message return code: • Return code 4 and return code 8 messages are stored in the CCA log. The CCA log is stored in a volatile message data buffer and is only available for the current boot cycle. • Return code 12 and return code 16 messages are stored in the current boot cycle syslog along with non-CCA messages. In addition to the current boot cycle syslog, there are up to four syslogs maintained from previous boot cycles.
SYSLGLEN	Specifies to return the length of the syslog data buffer in the **log_data_length** variable (for the given log number specified by the **log_number_or_level** parameter). No data is returned in the **log_data** variable.

log_number_or_level

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the log number or level for the service requested. This parameter may be unused, but must always be declared. Table 28 describes the meaning of this parameter for the different requested services.

Table 28. Meaning of **log_number_or_level** *for the requested service (keyword).*

Service requested	Direction	Meaning
CCALGLEN	Input	The value is ignored.

Table 28. Meaning of **log_number_or_level** *for the requested service (keyword) (continued).*

Service requested	Direction	Meaning
GETCCALG	Input	The value is ignored.
GETLGLVL	Output	This is the current minimum level of CCA return code for which a coprocessor generates a log message. Valid values are 4, 8, or 12.
GETSYSLG	Input	Specifies which of five possible groups of system log message data to retrieve from the coprocessor. See the note at end of the table.
SETLGLVL	Input	Specifies the minimum level of the CCA return code for which a coprocessor is to generate a log message. Valid values are 4, 8, or 12. If the log level is not set, the default is 12. **Value** **CCA return code that generates a log message** **4** 4, 8, 12, or 16 **8** 8, 12, or 16 **12** 12 or 16 (default if log level not set)
SYSLGLEN	Input	Specifies which length of five possible groups of syslog message data to retrieve from the coprocessor. See the note at end of the table.
Note: To process the system log message data from the current boot cycle, specify 0 in the **log_number_or_level** parameter. To process the data from the oldest available boot cycle possible, use a log number of 4. If the log number specifies an empty slot (that is, the coprocessor has not been power-cycled enough times for the given log number to have data yet), no data is returned.		

reserved0

Direction: Input/Output
Type: Integer

A pointer to an integer variable. This parameter is reserved for future use. It must be a null pointer or point to a value of 0.

log_data_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes in the **log_data** variable, or the maximum number of data bytes available to retrieve for a particular log inside the coprocessor. This parameter may be unused, but must always be declared. Table 29 describes the meaning of this parameter for the different requested services.

Table 29. Meaning of **log_data_length** *for the requested service (keyword).*

Service requested	Direction	Meaning
CCALGLEN	Output	Specifies the maximum number of CCA log data bytes available to retrieve inside the Coprocessor (the **log_number_or_level** parameter is ignored). No data is returned in the log_data variable.

Table 29. Meaning of **log_data_length** *for the requested service (keyword) (continued).*

Service requested	Direction	Meaning
GETCCALG	Input/Output	On input, this variable specifies the number of bytes available in the **log_data** variable. On output, the variable is updated to the number of bytes of CCA log data returned in the **log_data** variable. **Note:** 1. To determine the maximum number of bytes that the current CCA log data buffer contains, use the CCALGLEN keyword. The maximum is less than or equal to 1024. 2. The actual amount of data returned is variable and can be less than the maximum, depending on the amount of log messages issued so far. 3. On input, it is acceptable to specify a value less than the total size of the available CCA log data buffer. On output, the end of the data buffer is truncated as needed. 4. On input, it is acceptable to specify a value greater than the total size of the available CCA log data buffer. On output, the length is adjusted to the total size. 5. Interpretation of the data is defined internally by IBM. The data may or may not be human-readable. The data may contain truncated messages or partial messages that in some way conflict with a given editor.
GETLGLVL	Input	The value is ignored.
GETSYSLG	Input/Output	On input, this variable specifies the number of bytes available in the **log_data** variable. On output, the variable is updated to the number of bytes of syslog data returned in the **log_data** variable (for the given log number as specified by the **log_number_or_level** parameter). **Note:** 1. To determine the maximum number of bytes that the specified syslog data buffer contains, use the SYSLGLEN keyword. The maximum that can be returned is 16384. 2. The actual amount of data returned is variable and can be less than the maximum, depending on the amount of log messages issued so far. 3. On input, it is acceptable to specify a value less than the total size of the available syslog data buffer. On output, the end of the data buffer is truncated as needed. 4. On input, it is acceptable to specify a value greater than the total size of the available syslog data buffer. On output, the length is adjusted to the total size. 5. Interpretation of the data is defined internally by IBM. The data may or may not be human-readable. The data may contain truncated messages or partial messages that in some way conflict with a given editor.
SETLGLVL	Input	The value is ignored.
SYSLGLEN	Output	Specifies the maximum number of syslog data bytes available to retrieve inside the coprocessor (for the given log number as specified by the **log_number_or_level** parameter). No data is returned in the **log_data** variable.

log_data

Direction: Input/Output
Type: String

A pointer to a string variable containing the returned log data when keyword **GETCCALG** or **GETSYSLG** is specified in the rule array. This parameter may be unused, but must always be declared.

reserved1_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the **reserved1** variable. This parameter must be a null pointer or point to a value of 0.

reserved1

Direction: Input/Output
Type: String

This parameter is a pointer to a string variable. It is reserved for future use.

reserved2_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the **reserved2** variable. This parameter must be a null pointer or point to a value of 0.

reserved2

Direction: Input/Output
Type: String

This parameter is a pointer to a string variable. It is reserved for future use.

Restrictions

The restrictions for CSUALGQ.

None.

Required commands

The CSUALGQ required commands.

The Log Query verb requires the following commands to be enabled in the active role:

Table 30. Required commands for the Log Query verb.

Rule-array keyword	Log number value	Offset	Command
CCALGLEN, GETCCALG	Ignored	X'0035'	**Log Query: CCA**
GETSYSLG, SYSLGLEN	Ignored	X'0034'	**Log Query: System**
SETLGLVL	4	X'0036'	**Log Query: Set Log Level -4-**
	8	X'0037'	**Log Query: Set Log Level -8-**

Usage notes

Usage notes for CSUALGQ.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSUALGQJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSUALGQ(
          hikmNativeInteger   return_code,
          hikmNativeInteger   reason_code,
          hikmNativeInteger   exit_data_length,
          byte[]              exit_data,
          hikmNativeInteger   rule_array_count,
          byte[]              rule_array,
          hikmNativeInteger   log_number_or_level,
          hikmNativeInteger   reserved0,
          hikmNativeInteger   log_data_length,
          byte[]              log_data,
          hikmNativeInteger   reserved1_length,
          byte[]              reserved1,
          hikmNativeInteger   reserved2_length,
          byte[]              reserved2);
```

Master Key Process (CSNBMKP)

The Master Key Process verb operates on the three master-key registers: new, current, and old.

Use the verb to perform the following services:

- Clear the new and clear the old master-key registers.
- Generate a random master-key value in the new master-key register.
- XOR a clear value as a key part into the new master-key register.
- Set the master key, which transfers the current master-key to the old master-key register, and the new master-key to the current master-key register. It then clears the new master-key register.

You can choose to process either the symmetric or asymmetric registers by specifying the **SYM-MK** and the **ASYM-MK** *rule_array* keywords.

Tip: Before starting to load new master-key information, ensure the new master-key register is cleared. Do this by using the **CLEAR** keyword in the *rule_array*.

To form a master key from key parts in the new master-key register, use the verb several times to complete the following tasks:

- Clear the register, if it is not already clear.
- Load the first key part.
- Load any middle key parts, calling the verb once for each middle key part.
- Load the last key part.
- SET or confirm a master key for which the last key part has been loaded into the new master-key register.

For the SYM-MK, the low-order bit in each byte of the key is used as parity for the remaining bits in the byte. Each byte of the key part must contain an odd number of one bits. If this is not the case, a warning is issued. The product maintains odd parity on the accumulated symmetric master-key value.

When the last master key part is entered, this additional processing is performed:

- If any two of the 8-byte parts of the *new* master-key have the same value, a warning is issued. *Do not ignore this warning.* Do not use a key with this property.
- If any of the 8-byte parts of the *new* master-key compares equal to one of the weak DES-keys, the verb fails with return code 8, reason code 703. See "Questionable DES keys" on page 125 for a list of these weak keys. A parity-adjusted version of the asymmetric master-key is used to look for weak keys.

If an AES, DES or PKA key storage exists, the header record of each key storage is updated with the verification pattern of the new, current master key.

Format

The format of CSNBMKP.

```
CSNBMKP(
   return_code,
   reason_code,
   exit_data_length,
   exit_data,
   rule_array_count,
   rule_array,
   key_part)
```

Parameters

The parameter definitions for CSNBMKP.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1, 2, or 3.

rule_array

Direction: Input
Type: String array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* keywords are described in Table 31.

Table 31. Keywords for Master Key Process control information

Keyword	Description
Cryptographic component (Optional)	

Table 31. Keywords for Master Key Process control information (continued)

Keyword	Description
ADAPTER	Specifies the coprocessor. This is the default.
Master key register class (One, required) See Note at the end of this table.	
AES-MK	Specifies operation with the AES master-key registers.
APKA-MK	Specifies operation with the APKA master-key registers. This keyword was introduced with CCA 4.1.0.
ASYM-MK	Specifies operation with the asymmetric master-key registers.
SYM-MK	Specifies operation with the symmetric master-key registers.
Master-key process (One, required)	
CLEAR	Specifies to clear the NMK register.
FIRST	Specifies to load the first *key_part*.
MIDDLE	Specifies to XOR the second, third, or other intermediate *key_part* into the NMK register.
LAST	Specifies to XOR the last *key_part* into the NMK register.
SET	Specifies to advance the CMK to the OMK register, to advance the NMK to the CMK register, and to clear the NMK register.
Note: The *master-key register class* is not optional for Linux on z Systems. There is no default for this environment. If a suitable keyword is not specified, return code 8 with reason code 33 will be returned.	

key_part

Direction: Input
Type: String

A pointer to a string variable containing a 168-bit or 192-bit clear key-part used when you specify one of the keywords **FIRST**, **MIDDLE**, or **LAST**. If you use the **CLEAR** or **SET** keywords, the information in the variable is ignored, but you must declare the variable.

Restrictions

The restrictions for CSNBMKP.

General restrictions:

- You must set up the groups for the users who will be loading the master keys to the cards. Each part of the load process is owned by a different Linux group created by the RPM install procedure, and verified in the host library implementing the API allowing master key processing. To complete a specific step, the user must have membership in the proper group. See **Master key load** (Step 7 on page 995).
- The **AES-MK** rule-array keyword is not supported in releases before Release 3.30.
- The **APKA-MK** rule-array keyword is not supported in releases before Release 4.1.0.

For applications that use this verb:

- When writing your own application, you must link it with the `/usr/lib64/libcsulccamk.so` library.

Required commands

The CSNBMKP required commands.

This verb requires the following commands to be enabled in the active role based on the master-key class and master-key operation:

Master-key operation	Master-key class	Offset	Command
CLEAR	AES-MK	X'0124'	**AES Master Key - Clear new master key register**
	SYM-MK	X'0032'	**DES Master Key - Clear new master key register**
	ASYM-MK	X'0060'	**RSA Master Key - Clear new master key register**
	APKA-MK	X'031F'	**ECC Master Key - Clear new master key register**
FIRST	AES-MK	X'0125'	**AES Master Key - Load first key part**
	SYM-MK	X'0018'	**DES Master Key - Load first key part**
	ASYM-MK	X'0053'	**RSA Master Key - Load first key part**
	APKA-MK	X'0320'	**ECC Master Key - Load first key part**
MIDDLE or LAST	AES-MK	X'0126'	**AES Master Key - Combine key parts**
	SYM-MK	X'0019'	**DES Master Key - Combine key parts**
	ASYM-MK	X'0054'	**RSA Master Key - Combine key parts**
	APKA-MK	X'0321'	**ECC Master Key - Combine key parts**
SET	AES-MK	X'0128'	**AES Master Key - Set master key**
	SYM-MK	X'001A'	**DES Master Key - Set master key**
	ASYM-MK	X'0057'	**RSA Master Key - Set master key**
	APKA-MK	X'0322'	**ECC Master Key - Set master key**

Usage notes

Usage notes for CSNBMKP.

This verb is not impacted by the AUTOSELECT option. See "Verbs that ignore AUTOSELECT" on page 10 for more information.

Questionable DES keys

These keys are considered questionable DES keys, and so should probably not be used when entering SYM-MK or ASYM-MK master keys.

```
01 01 01 01 01 01 01 01  /* weak */
FE FE FE FE FE FE FE FE  /* weak */
1F 1F 1F 1F 0E 0E 0E 0E  /* weak */
E0 E0 E0 E0 F1 F1 F1 F1  /* weak */
01 FE 01 FE 01 FE 01 FE  /* semi-weak */
FE 01 FE 01 FE 01 FE 01  /* semi-weak */
1F E0 1F E0 0E F1 0E F1  /* semi-weak */
E0 1F E0 1F F1 0E F1 0E  /* semi-weak */
01 E0 01 E0 01 F1 01 F1  /* semi-weak */
E0 01 E0 01 F1 01 F1 01  /* semi-weak */
1F FE 1F FE 0E FE 0E FE  /* semi-weak */
FE 1F FE 1F FE 0E FE 0E  /* semi-weak */
01 1F 01 1F 01 0E 01 0E  /* semi-weak */
1F 01 1F 01 0E 01 0E 01  /* semi-weak */
E0 FE E0 FE F1 FE F1 FE  /* semi-weak */
FE E0 FE E0 FE F1 FE F1  /* semi-weak */
1F 1F 01 01 0E 0E 01 01  /* possibly semi-weak  */
01 1F 1F 01 01 0E 0E 01  /* possibly semi-weak  */
1F 01 01 1F 0E 01 01 0E  /* possibly semi-weak  */
01 01 1F 1F 01 01 0E 0E  /* possibly semi-weak  */
E0 E0 01 01 F1 F1 01 01  /* possibly semi-weak  */
FE FE 01 01 FE FE 01 01  /* possibly semi-weak  */
FE E0 1F 01 FE F1 0E 01  /* possibly semi-weak  */
```

```
E0 FE 1F 01 F1 FE 0E 01  /* possibly semi-weak  */
FE E0 01 1F FE F1 01 0E  /* possibly semi-weak  */
E0 FE 01 1F F1 FE 01 0E  /* possibly semi-weak  */
E0 E0 1F 1F F1 F1 0E 0E  /* possibly semi-weak  */
FE FE 1F 1F FE FE 0E 0E  /* possibly semi-weak  */
FE 1F E0 01 FE 0E F1 01  /* possibly semi-weak  */
E0 1F FE 01 F1 0E FE 01  /* possibly semi-weak  */
FE 01 E0 1F FE 01 F1 0E  /* possibly semi-weak  */
E0 01 FE 1F F1 01 FE 0E  /* possibly semi-weak  */
01 E0 E0 01 01 F1 F1 01  /* possibly semi-weak  */
1F FE E0 01 0E FE F1 01  /* possibly semi-weak  */
1F E0 FE 01 0E F1 FE 01  /* possibly semi-weak  */
01 FE FE 01 01 FE FE 01  /* possibly semi-weak  */
1F E0 E0 1F 0E F1 F1 0E  /* possibly semi-weak  */
01 FE E0 1F 01 FE F1 0E  /* possibly semi-weak  */
01 E0 FE 1F 01 F1 FE 0E  /* possibly semi-weak  */
1F FE FE 1F 0E FE FE 0E  /* possibly semi-weak  */
E0 01 01 E0 F1 01 01 F1  /* possibly semi-weak  */
FE 1F 01 E0 FE 0E 01 F1  /* possibly semi-weak  */
FE 01 1F E0 FE 01 0E F1  /* possibly semi-weak  */
E0 1F 1F E0 F1 0E 0E F1  /* possibly semi-weak  */
FE 01 01 FE FE 01 01 FE  /* possibly semi-weak  */
E0 1F 01 FE F1 0E 01 FE  /* possibly semi-weak  */
E0 01 1F FE F1 01 0E FE  /* possibly semi-weak  */
FE 1F 1F FE FE 0E 0E FE  /* possibly semi-weak  */
1F FE 01 E0 E0 FE 01 F1  /* possibly semi-weak  */
01 FE 1F E0 01 FE 0E F1  /* possibly semi-weak  */
1F E0 01 FE 0E F1 01 FE  /* possibly semi-weak  */
01 E0 1F FE 01 F1 0E FE  /* possibly semi-weak  */
01 01 E0 E0 01 01 F1 F1  /* possibly semi-weak  */
1F 1F E0 E0 0E 0E F1 F1  /* possibly semi-weak  */
1F 01 FE E0 0E 01 FE F1  /* possibly semi-weak  */
01 1F FE E0 01 0E FE F1  /* possibly semi-weak  */
1F 01 E0 FE 0E 01 F1 FE  /* possibly semi-weak  */
01 1F E0 FE 01 E0 F1 FE  /* possibly semi-weak  */
01 01 FE FE 01 01 FE FE  /* possibly semi-weak  */
1F 1F FE FE 0E 0E FE FE  /* possibly semi-weak  */
FE FE E0 E0 FE FE F1 F1  /* possibly semi-weak  */
E0 FE FE E0 F1 FE FE F1  /* possibly semi-weak  */
FE E0 E0 FE FE F1 F1 FE  /* possibly semi-weak  */
E0 E0 FE FE F1 F1 FE FE  /* possibly semi-weak  */
```

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBMKPJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBMKPJ(
          hikmNativeInteger return_code,
          hikmNativeInteger reason_code,
          hikmNativeInteger exit_data_length,
          byte[]            exit_data,
          hikmNativeInteger rule_array_count,
          byte[]            rule_array,
          byte[]            key_part);
```

Random Number Tests (CSUARNT)

The Random Number Tests verb invokes the USA NIST FIPS PUB 140-1 specified cryptographic operational tests.

These tests, selected by a *rule_array* keyword, consist of:
- For random numbers: a monobit test, poker test, runs test, and long-run test
- Known-answer tests of DES, RSA, and SHA-1 processes

The tests are performed three times. If there is any test failure, the verb returns return code 4 and reason code 1.

Format

The format of CSUARNT.

```
CSUARNT(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array)
```

Parameters

The parameter definitions for CSUARNT.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1.

rule_array

Direction: Input
Type: String array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* keywords are described in Table 32.

Table 32. Keywords for Random Number Tests control information

Keyword	Description
Test selection (One required)	
FIPS-RNT	Perform the FIPS 140-1 specified test on the random number generation output.
KAT	Perform the FIPS 140-1 specified known-answer tests on DES, RSA, and SHA-1.

Restrictions

The restrictions for CSUARNT.

None.

Required commands

The CSUARNT required commands.

None.

Usage notes

Usage notes for CSUARNT.

This verb is not impacted by the AUTOSELECT option. See "Verbs that ignore AUTOSELECT" on page 10 for more information.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSUARNTJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSUARNTJ(
          hikmNativeInteger   return_code,
          hikmNativeInteger   reason_code,
          hikmNativeInteger   exit_data_length,
          byte[]              exit_data,
          hikmNativeInteger   rule_array_count,
          byte[]              rule_array);
```

Chapter 6. Managing AES and DES cryptographic keys

A group of verbs that generate and maintain AES and DES cryptographic keys.

Using CCA, you can generate keys using the Key Generate verb. CCA provides a number of verbs to assist you in managing and distributing AES and DES keys, generating random numbers, and maintaining the key storage files.

The following verbs are described:

Clear Key Import (CSNBCKI)

Use the Clear Key Import verb to import a clear DATA key that is to be used to encipher or decipher data.

This verb can import only DATA keys. The Clear Key Import verb accepts an 8-byte clear DATA key, enciphers it under the master key, and returns the encrypted DATA key in operational form in an internal key token.

If the clear key value does not have odd parity in the low-order bit of each byte, the verb returns a warning value in the *reason_code* parameter. This verb does not adjust the parity of the key.

Note: To import 16-byte or 24-byte DATA keys, use the Multiple Clear Key Import verb that is described in "Multiple Clear Key Import (CSNBCKM)" on page 131.

Format

The format of CSNBCKI.

```
CSNBCKI(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             clear_key,
             key_identifier)
```

Parameters

The parameter definitions for CSNBCKI.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

clear_key

Direction: Input
Type: String

The *clear_key* specifies the 8-byte clear key value to import.

key_identifier

Direction: Input/Output
Type: String

A 64-byte string that is to receive the internal key token. "Key tokens, key labels, and key identifiers" on page 21 describes the internal key token.

Restrictions

The restrictions for CSNBCKI.

None.

Required commands

The CSNBCKI required commands.

This verb requires the **Clear Key Import/Multiple Clear Key Import - DES** command (offset X'00C3') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Note: A role with offset X'00C3' enabled can also use the Multiple Clear Key Import verb with the DES algorithm.

Usage notes

Usage notes for CSNBCKI.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBCKIJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBCKIJ(
  hikmNativeInteger   return_code,
  hikmNativeInteger   reason_code,
  hikmNativeInteger   exit_data_length,
  byte[]              exit_data,
  byte[]              clear_key,
  byte[]              target_key_identifier);
```

Multiple Clear Key Import (CSNBCKM)

Use the Multiple Clear Key Import verb to import a clear single, double, or triple-length DATA key that is to be used to encipher or decipher data.

This verb can import only DATA keys. Multiple Clear Key Import accepts a clear DATA key, enciphers it under the master key, and returns the encrypted DATA key in operational form in an internal key token.

Format

The format of CSNBCKM.

```
CSNBCKM(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             clear_key_length,
             clear_key,
             target_key_identifier)
```

Parameters

The parameter definitions for CSNBCKM.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0, 1, 2, or 3.

rule_array

Direction: Input
Type: String array

Zero or one keyword that supplies control information to the verb. The keyword must be in eight bytes of contiguous storage, left-aligned and padded on the right with blanks. The *rule_array* keywords are described in Table 33.

Table 33. Keywords for Multiple Clear Key Import control information

Keyword	Description
Algorithm (On, optional)	
AES	The key should be enciphered under the master key as an AES key.
DES	The key should be enciphered under the master key as a DES key. This is the default.
Key-wrapping method (One, optional)	
USECONFG	This is the default. Specifies to wrap the key using the configuration setting for the default wrapping method. The default wrapping method configuration setting may be changed using the TKE. This keyword is ignored for AES keys.
WRAP-ENH	Specifies to wrap the key using the legacy wrapping method. This keyword is ignored for AES keys. This keyword was introduced with CCA 4.1.0.
WRAP-ECB	Specifies to wrap the key using the enhanced wrapping method. Valid only for DES keys. This keyword was introduced with CCA 4.1.0.

Table 33. Keywords for Multiple Clear Key Import control information (continued)

Keyword	Description
Translation control (Optional). This is valid only with key-wrapping method **WRAP-ENH** or with **USECONFG** when the default wrapping method is **WRAP-ENH**. This option cannot be used on a key with a control vector valued to binary zeros. This keyword was introduced with CCA 4.1.0.	
ENH-ONLY	Specifies to restrict the key from being wrapped with the legacy wrapping method after it has been wrapped with the enhanced wrapping method. Sets bit 56 (**ENH-ONLY**) of the control vector to B'1'.

clear_key_length

Direction: Input
Type: Integer

The *clear_key_length* specifies the length of the clear key value to import. This length must be 8, 16, or 24.

clear_key

Direction: Input
Type: String

The *clear_key* specifies the clear key value to import.

target_key_identifier

Direction: Output
Type: String

A 64-byte string that is to receive the internal key token. Chapter 17, “Key token formats,” on page 769 describes the key tokens.

Restrictions

The restrictions for CSNBCKM.

None.

Required commands

The CSNBCKM required commands.

This verb requires the following commands to be enabled in the active role based on the algorithm or key-wrapping method:

Algorithm or method	Offset	Command
AES	X'0129'	**Multiple Clear Key Import/Multiple Secure Key Import - AES**
DES	X'00C3'	**Clear Key Import/Multiple Clear Key Import - DES**
WRAP-ECB or WRAP-ENH used, and default key-wrapping method setting does not match keyword	X'0141'	**Multiple Clear Key Import - Allow wrapping override keywords**

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Note: Note: A role with offset X'00C3' can also use the Clear Key Import verb.

Usage notes

Usage notes for CSNBCKM.

This verb produces an internal DATA token with a control vector which is usable on the Cryptographic Coprocessor Feature. If a valid internal token is supplied as input to the verb in the *target_key_identifier* field, that token's control vector will not be used in the encryption of the clear key value.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBCKMJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBCKMJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger clear_key_length,
           byte[]            clear_key,
           byte[]            target_key_identifier);
```

Control Vector Generate (CSNBCVG)

The Control Vector Generate verb builds a control vector from keywords specified by the *key_type* and *rule_array* parameters.

Format

The format of CSNBCVG.

```
CSNBCVG(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             key_type,
             rule_array_count,
             rule_array,
             reserved,
             control_vector)
```

Parameters

The parameter definitions for CSNBCVG.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

key_type

Direction: Input
Type: String

A string variable containing a keyword for the key type. The keyword is eight bytes in length, left-aligned, and padded on the right with space characters. It is taken from the following list:

CIPHER	CVARXCVL	DKYGENKY	MAC
CIPHERXI	CVARXCVR	ENCIPHER	MACVER
CIPHERXL	DATA	EXPORTER	OKEYXLAT
CIPHERXO	DATAC	IKEYXLAT	OPINENC
CVARDEC	DATAM	IMPORTER	PINGEN
CVARENC	DATAMV	IPINENC	PINVER
CVARPINE	DECIPHER	KEYGENKY	SECMSG

For information on the meaning of the key types, see Table 4 on page 40.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. Each keyword is left-aligned in 8-byte fields, and padded on the right with blanks. All keywords must be in contiguous storage. "Key Token Build (CSNBKTB)" on page 213 illustrates the key type and key usage keywords that can be combined in the Control Vector Generate and Key Token Build verbs to create a control vector.

See Figure 3 on page 43 for the key usage keywords that can be specified for a given key type. The *rule_array* keywords are shown here:

AMEX-CSC	DKYL0	ENH-ONLY	KEYLN8	SMPIN
ANSIX9.9	DKYL1	EPINGEN	KEYLN16	TRANSLAT
ANY	DKYL2	EPINGENA	LMTD-KEK	T31XPTOK
ANY-MAC	DKYL3	EPINVER	MIXED	UKPT
CLR8-ENC	DKYL4	EXEX	NO-SPEC	VISA-PVV
CPINENC	DKYL5	EXPORT	NO-XPORT	XLATE
CPINGEN	DKYL6	GBP-PIN	NOOFFSET	XPORT-OK
CPINGENA	DKYL7	GBP-PINO	NOT31XPT	
CVVKEY-A	DMAC	IBM-PIN	NOT-KEK	
CVVKEY-B	DMKEY	IBM-PINO	OPEX	
DALL	DMPIN	IMEX	OPIM	
DATA	DMV	IMIM	PIN	
DDATA	DOUBLE	IMPORT	REFORMAT	
DEXP	DOUBLE-O	INBK-PIN	SINGLE	
DIMP	DPVR	KEY-PART	SMKEY	

Note:

1. When the **KEYGENKY** key type is coded, either **CLR8-ENC** or **UKPT** must be specified in *rule_array*.
2. When the **SECMSG** *key_type* is coded, either **SMKEY** or **SMPIN** must be specified in the *rule_array*.
3. Keyword **ENH-ONLY** was introduced with CCA 4.1.0.

reserved

Direction: Input
Type: String

The *reserved* parameter must be a variable of eight bytes of X'00'.

control_vector

Direction: Output
Type: String

A 16-byte string variable in application storage where the verb returns the generated control vector.

Restrictions

The restrictions for CSNBCVG.

None.

Required commands

The CSNBCVG required commands.

None.

Usage notes

Usage notes for CSNBCVG.

See the *key_type* parameter on page "key_type " on page 214 for an illustration of key type and key usage keywords that can be combined in the Control Vector Generate and Key Token Build verbs to create a control vector.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBCVGJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBCVGJ(
          hikmNativeInteger return_code,
          hikmNativeInteger reason_code,
          hikmNativeInteger exit_data_length,
          byte[]            exit_data,
          byte[]            key_type,
          hikmNativeInteger rule_array_count,
          byte[]            rule_array,
          byte[]            reserved_field_1,
          byte[]            control_vector);
```

Control Vector Translate (CSNBCVT)

The Control Vector Translate verb changes the control vector used to encipher an external DES key.

Detailed information about control vectors and how to use this verb can be found in Chapter 19, "Control vectors and changing control vectors with the Control Vector Translate verb," on page 897.

Format

The format of CSNBCVT.

```
CSNBCVT(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              KEK_key_identifier,
              source_key_token,
              array_key_left_identifier,
              mask_array_left,
              array_key_right_identifier,
              mask_array_right,
              rule_array_count,
              rule_array,
              target_key_token)
```

Parameters

The parameter definitions for CSNBCVT.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

KEK_key_identifier

Direction: Input
Type: String

A pointer to a string variable containing an operational key-token or the key label of an operational key-token record containing the key-encrypting key. The control vector in the key token must specify the key type **IMPORTER**, **EXPORTER**, **IKEYXLAT**, or **OKEYXLAT**.

source_key_token

Direction: Input
Type: String

A pointer to a string variable containing the external DES key-token with the key and control vector to be processed.

array_key_left_identifier

Direction: Input
Type: String

A pointer to a string variable containing an operational DES key-token or a key label of an operational DES key-token record that deciphers the left mask-array. The key token must contain a control vector specifying a **CVARXCVL** key-type. The **CVARXCVL** key must be single length.

mask_array_left

Direction: Input
Type: String

A pointer to a string variable containing the mask array enciphered under the left-array key.

array_key_right_identifier

Direction: Input
Type: String

A pointer to a string variable containing an operational DES key-token or the key label of an operational DES key-token record that deciphers the right mask-array. The key token must contain a control vector specifying a **CVARXCVR** key-type. The **CVARXCVR** key must be single length.

mask_array_right

Direction: Input
Type: String

A pointer to a string variable containing the mask array enciphered under the right-array key.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0, 1, or 2.

rule_array

Direction: Input
Type: String array

A pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* keywords are described in Table 34.

Table 34. Keywords for Control Vector Translate control information

Keyword	Description
Parity adjustment (One, optional)	
ADJUST	Ensures that all target-key bytes have odd parity. This is the default.
NOADJUST	Prevents the parity of the target key from being altered.
Key half processing mode (One, optional)	
LEFT	Causes an 8-byte source key, or the left half of a 16-byte source key, to be processed with the result placed into both halves of the target key. This is the default.
RIGHT	Causes the right half of a 16-byte source key to be processed with the result placed into only the right half of the target key. The left half of the target key is unchanged.
BOTH	Causes both halves of a 16-byte source key to be processed with the result placed into corresponding halves of the target key. When you use the **BOTH** keyword, the mask array must be able to validate the translation of both halves.
SINGLE	Causes the left half of the source key to be processed with the result placed into only the left half of the target. The right half of the target key is unchanged.

target_key_token

Direction: Input/Output
Type: String

A pointer to a string variable containing an external DES key-token with the new control vector. This key token contains the key halves with the new control vector.

Restrictions

The restrictions for CSNBCVT.

None.

Required commands

The CSNBCVT required commands.

This verb requires the **Control Vector Translate** command (offset X'00D6') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

Usage notes for CSNBCVT.

Consider that Control Vector Translate represents the capability to translate, by definition, the limitations on the operations that a key can be used for, into a different set of limitations. The control vector is the heart of security against the misuse of keys that were defined for a specific purpose. The masks that control what the key can be translated into being able to do (the right and left masks) are themselves single-length (8-byte), and are encrypted with DES. Therefore, the protection against translating the key to have more power (or less power) than it did before are protected with single-DES. This reduces the security (somewhat) of a double-length DES key. You cannot decrypt the double-length key with this approach, or gain access to a key that you did not otherwise have the rights to use. But you can make a key which you already have access to, on a system you already have access to, more powerful than it was before if you can break single-DES.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBCVTJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBCVTJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           byte[]            kek_key_identifier,
           byte[]            source_key_token,
           byte[]            array_key_left,
           byte[]            mask_array_left,
           byte[]            array_key_right,
           byte[]            mask_array_right,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            target_key_token);
```

Cryptographic Variable Encipher (CSNBCVE)

This verb is used to encrypt plain text using a CVARENC key to produce ciphertext using the Cipher Block Chaining (CBC) method.

The plaintext must be a multiple of eight bytes in length.

Specify the following parameters to encrypt plaintext:

- An operational DES key-token or a key label of an operational DES key-token record that contains the key to be used to encrypt the plaintext with the *c-variable_encrypting_key_identifier* parameter. The control vector in the key token must specify the **CVARENC** key-type.
- The length of the plaintext, which is the same as the length of the returned ciphertext, with the *text_length* parameter. The plaintext must be a multiple of eight bytes in length.
- The plaintext with the *plaintext* parameter.
- The initialization vector with the *initialization_vector* parameter.
- A variable for the returned ciphertext with the *ciphertext* parameter. The length of this field is specified with the *text_length* variable.

This verb does the following:

- Uses the **CVARENC** key and the initialization value with the CBC method to encrypt the plaintext.
- Returns the encrypted plaintext in the variable pointed to by the *ciphertext* parameter.

Format

The format of CSNBCVE.

```
CSNBCVE(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             c-variable_encrypting_key_identifier,
             text_length,
             plaintext,
             initialization_vector,
             ciphertext)
```

Parameters

The parameter definitions for CSNBCVE.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

c-variable_encrypting_key_identifier

Direction: Input
Type: String

A pointer to a string variable containing an operational DES key-token or a key label of an operational DES key-token record. The key token must contain a control vector that specifies a CVARENC key-type.

text_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the length of the *plaintext* variable and the *ciphertext* variable.

plaintext

Direction: Input
Type: String

A pointer to is a string variable containing the plaintext to be encrypted.

initialization_vector

Direction: Input
Type: String

A pointer to a string variable containing the 8-byte initialization vector that the verb uses in encrypting the plain text.

ciphertext

Direction: Output
Type: String

A pointer to a string variable containing the ciphertext returned by the verb.

Restrictions

The restrictions for CSNBCVE.

The text length must be a multiple of eight bytes.

The minimum length of text that the security server can process is eight bytes and the maximum is 256 bytes.

Required commands

The required commands for CSNBCVE.

This verb requires the **Cryptographic Variable Encipher** command (offset X'00DA') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

Usage notes for CSNBCVE.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBCVEJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBCVEJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           byte[]            cvarenc_key_id,
           hikmNativeInteger text_length,
           byte[]            plain_text,
           byte[]            init_vector,
           byte[]            cipher_text);
```

Data Key Export (CSNBDKX)

Use the Data Key Export verb to re-encipher a data-encrypting key (key type of DATA only) from encryption under the master key to encryption under an exporter key-encrypting key.

The re-enciphered key is in a form suitable for export to another system.

The Data Key Export verb generates a key token with the same key length as the input token's key.

Format

The format of CSNBDKX.

```
CSNBDKX(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             source_key_identifier,
             exporter_key_identifier,
             target_key_identifier)
```

Parameters

The parameter definitions for CSNBDKX.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

source_key_identifier

Direction: Input/Output
Type: String

A 64-byte string for an internal key token or label that contains a *data-encrypting* key to be re-enciphered. The data-encrypting key is encrypted under the master key.

exporter_key_identifier

Direction: Input/Output
Type: String

A 64-byte string for an internal key token or key label that contains the exporter *key_encrypting* key. The data-encrypting key above will be encrypted under this exporter *key_encrypting* key.

target_key_identifier

Direction: Input/Output
Type: String

A 64-byte field that is to receive the external key token, which contains the re-enciphered key that has been exported. The re-enciphered key can now be exchanged with another cryptographic system.

Restrictions

The restrictions for CSNBDKX.

For security reasons, requests will fail by default if they use an equal key halves exporter to export a key with unequal key halves. You must have access control point 'Data Key Export - Unrestricted' explicitly enabled if you want to export keys in this manner.

Required commands

The CSNBDKX required commands.

This verb requires the **Data Key Export** command (offset X'010A') to be enabled in the active role.

By also specifying the **Data Key Export - Unrestricted** command (offset X'0277'), you can permit a less secure mode of operation that enables an equal key-halves EXPORTER key-encrypting-key to export a key having unequal key-halves (key parity bits are ignored).

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

Usage notes for.CSNBDKX.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBDKXJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBDKXJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  byte[]            source_key_identifier,
  byte[]            exporter_key_identifier,
  byte[]            target_key_token);
```

Data Key Import (CSNBDKM)

Use the Data Key Import verb to import an encrypted source DES single-length, double-length or triple-length DATA key and create or update a target internal key token with the master key enciphered source key.

Format

The format of CSNBDKM.

```
CSNBDKM(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             source_key_token,
             importer_key_identifier,
             target_key_identifier)
```

Parameters

The parameter definitions for CSNBDKM.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

source_key_token

Direction: Input
Type: String

A pointer to a 64-byte string variable containing the source key to be imported. The source key must be an external token or null token. The external key token must indicate that a control vector is present; however, the control vector is usually valued at zero. A double-length key that should result in a default **DATA** control vector must be specified in a version X'01' external key token. Otherwise, both single and double-length keys are presented in a version X'00' key token. For the null token, the verb will process this token format as a **DATA** key encrypted by the importer key and a null (all zero) control vector.

importer_key_identifier

Direction: Input/Output
Type: String

A pointer to a 64-byte string variable containing the (IMPORTER) transport key or key label of the transport key used to decipher the source key.

target_key_identifier

Direction: Output
Type: String

A pointer to a 64-byte string variable containing a null key token or an internal key token. The key token receives the imported key.

Restrictions

The restrictions for CSNBDKM.

For security reasons, requests will fail by default if they use an equal key halves importer to import a key with unequal key halves. You must have access control point 'Data Key Import - Unrestricted' explicitly enabled if you want to import keys in this manner.

Required commands

The CSNBDKM required commands.

This verb requires the **Data Key Import** command (offset X'0109') to be enabled in the active role.

By also specifying the **Data Key Import - Unrestricted** command (offset X'027C'), you can permit a less secure mode of operation that enables an equal key-halves **IMPORTER** key-encrypting key to import a key having unequal key-halves (key parity bits are ignored).

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

Usage notes for CSNBDKM.

This verb does not adjust the key parity of the source key.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBDKMJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBDKMJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  byte[]            source_key_token,
  byte[]            importer_key_identifier,
  byte[]            target_key_identifier);
```

Diversified Key Generate (CSNBDKG)

Use the Diversified Key Generate verb to generate a key based on the key-generating key, the processing method, and the parameter supplied.

The control vector of the key-generating key also determines the type of target key that can be generated.

To use this verb, specify the following:

- The *rule_array* keyword to select the diversification process.
- The operational key-generating key from which the diversified keys are generated. The control vector associated with this key restricts the use of this key to the key generation process. This control vector also restricts the type of key that can be generated.
- The data and length of data used in the diversification process.
- The generated-key could be an internal token or a skeleton token containing the desired CV of the generated-key. The generated key CV must be one that is permitted by the processing method and the key-generating key. The generated key will be returned in this parameter.
- A key generation method keyword.

This verb generates diversified keys as follows:

- Determines if it can support the process specified in the *rule_array*.

- Recovers the key-generating key and checks the key-generating key class and the specified usage of the key-generating key.
- Determines that the control vector in the generated-key token is permissible for the specified processing method.
- Determines that the control vector in the generated-key token is permissible by the control vector of the key-generating key.
- Determines the required data length from the processing method and the generated-key CV. Validates the *data_length*.
- Generates the key appropriate to the specific processing method. Adjusts parity of the key to odd. Creates the internal token and returns the generated diversified key.

Format

The format of CSNBDKG.

```
CSNBDKG(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             generating_key_identifier,
             data_length,
             data,
             data_decrypting_key_identifier,
             generated_key_identifier)
```

Parameters

The parameter definitions for CSNBDKG.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1, 2, or 3.

rule_array

Direction: Input
Type: String array

A pointer to a string variable containing an array of keywords that provide control information to the verb. The processing method is the algorithm used to create the generated key. The keyword is left-aligned and padded on the right with blanks. The *rule_array* keywords are described in Table 35.

Table 35. Keywords for Diversified Key Generate control information

Keyword	Description
Processing Method for generating or updating diversified keys (One required)	

Table 35. Keywords for Diversified Key Generate control information (continued)

Keyword	Description
CLR8-ENC	Specifies that eight bytes of clear (not encrypted) data shall be Triple-DES encrypted with the generating key (**generating_key_identifier**) to create a generated key. The key selected by the generating_key_identifier must specify a **KEYGENKY** key-type also with control vector bit 19 set to B'1'. The key identified by the **data_decrypting_key_identifier** must identify a null DES key-token. The key token identified by the generated_key_identifier variable must contain a control vector that specifies a single-length key of one of these types: **DATA**, **CIPHER**, **DECIPHER**, **MAC**, or **MACVER**.
TDES-CBC	Specifies that 16 bytes of clear (not encrypted) data shall be Triple-DES encrypted with the generating key to create the generated key. If the generated_key_identifier variable specifies a double-length key, then 16 bytes of clear data are Triple-DES encrypted in CBC mode with an initial value of binary zeros. **Note:** The EMV Card Personalization specification states that CBC encryption mode should be used in the diversification process. The key selected by the **generating_key_identifier** parameter must specify a DKYGENKY key-type that has the appropriate control vector usage bits (bits 19 – 22) set for the desired generated key. Control vector bits 12 – 14 binary encode the key-derivation sequence level (**DKYL7** down to **DKYL0**). The final key is derived when bits 12 – 14 are B'000'. The verb verifies the incremental relationship between the value in the control vectors of the key tokens identified by the generated_key_identifier parameter and the generating_key_identifier parameter control vector. In the case when the generated_key_identifier is a null DES token, the appropriate counter value is placed into the output key-token. The **data_decrypting_key_identifier** parameter must identify a null DES key-token. A key token identified by the **generated_key_identifier** parameter that is not a null key-token must contain a control vector that specifies a double-length key having a key type that is consistent with the specification in bits 19 – 22 of the generating key.

Table 35. Keywords for Diversified Key Generate control information (continued)

Keyword	Description
TDES-DEC	Specifies that 8 or 16 bytes of clear (not encrypted) data shall be Triple-DES decrypted with the generating key to create the generated key. If the **generated_key_identifier** variable specifies a single-length key, then 8 bytes of clear data are Triple-DES decrypted. If the **generated_key_identifier** variable specifies a double-length key, then 16 bytes of clear data are Triple-DES decrypted in ECB mode. The key selected by the generating_key_identifier must specify a **DKYGENKY** key-type that has the appropriate control vector usage bits (bits 19 – 22) set for the desired generated key. Control vector bits 12 – 14 binary encode the key-derivation sequence level (**DKYL7** down to **DKYL0**). The final key is derived when bits 12 – 14 are B'000'. The verb verifies the incremental relationship between the value in the generated_key_identifier variable control-vector and the generating_key_identifier variable control-vector. Or in the case when the generated_key_identifier variable is a null DES key-token, the appropriate counter value is placed into the output key-token. The **data_decrypting_key_identifier** parameter must identify a null DES key-token. A key token identified by the generated_key_identifier variable that is not a null DES key-token must contain a control vector that specifies a single-length or double-length key having a key type consistent with the specification in bits 19 – 22 of the generating key.
TDES-ENC	Specifies that 16 bytes of clear (not encrypted) data shall be Triple-DES decrypted with the generating key to create the generated key. If the **generated_key_identifier** variable specifies a double-length key, then 16 bytes of clear data are Triple-DES decrypted in CBC mode with an initial value of binary zeros. **Note:** The EMV Card Personalization specification states that CBC encryption mode should be used in the diversification process. The key selected by the **generated_key_identifier** must specify a **DKYGENKY** key-type that has the appropriate control vector usage bits (bits 19 – 22) set for the desired generated key. Control vector bits 12 – 14 binary encode the key-derivation sequence level (**DKYL7** down to **DKYL0**). The final key is derived when bits 12 – 14 are B'000'. The verb verifies the incremental relationship between the value in the generated_key_identifier variable control-vector and the generating_key_identifier variable control-vector. Or in the case when the generated_key_identifier variable is a null DES key-token, the appropriate counter value is placed into the output key-token. The **data_decrypting_key_identifier** parameter must identify a null DES key-token. A key token identified by the generated_key_identifier variable that is not a null DES key-token must contain a control vector that specifies a single-length or double-length key having a key type consistent with the specification in bits 19 – 22 of the generating key.

Table 35. Keywords for Diversified Key Generate control information (continued)

Keyword	Description
TDES-XOR	Specifies that 10 bytes or 18 bytes of clear (not encrypted) data shall be processed as described in "Working with Europay-Mastercard-Visa Smart cards" on page 444 to create the generated key. The data variable contains either 8 bytes or 16 bytes of data to be triple-encrypted to which you append a 2-byte Application Transaction Counter value (previously received from the smart card). Place the counter value in a string construct with the high-order counter bit first in the string. The key selected by the generating_key_identifier parameter must specify a DKYGENKY key-type at level-0 (bits 12 – 14 B'000') and indicate permission to create one of several key types in bits 19 – 22: **B'0001'** DDATA, to generate a DATA key **B'0010'** DMAC, to generate a MAC key **B'0011'** DMV, to generate a MACVER key **B'1000'** DMKEY, to generate a SECMSG SMKEY (used in secure messaging, key encryption, see the Secure Messaging for PINs verb) **B'1001'** DMPIN, to generate a SECMSG SMPIN (used in secure messaging, PIN encryption, see the Secure Messaging for PINs verb). The **data_decrypting_key_identifier** parameter must identify a null DES key-token. A key token or key-token record identified by the generated_key_identifier parameter that is not a null DES key-token. The token must contain a control vector that specifies a key type conforming to that specified in control-vector bits 19 – 22 for the key-generating key. The control vector must specify a double-length key.
TDESEMV2	This option supports generation of a session key by the EMV 2000 algorithm (This EMV2000 algorithm uses a branch factor of 2). The generating key must be a level 0 **DKYGENKY** and cannot have replicated halves. The session key generated must be double length and the allowed key types are **DATA**, **DATAC**, **DATAM**, **DATAMV**, **MAC**, **MACVER**, **SMPIN**, and **SMKEY**. Key type must be allowed by the generating key control vector.
TDESEMV4	This option supports generation of a session key by the EMV 2000 algorithm (This EMV2000 algorithm uses a branch factor of 4). The generating key must be a level 0 **DKYGENKY** and cannot have replicated halves. The session key generated must be double length and the allowed key types are **DATA**, **DATAC**, **DATAM**, **DATAMV**, **MAC**, **MACVER**, **SMPIN**, and **SMKEY**. Key type must be allowed by the generating key control vector.
Processing Method for updating a diversified key (optional)	

Table 35. Keywords for Diversified Key Generate control information (continued)

Keyword	Description
SESS-XOR	Specifies the VISA method for session key generation, namely that 8 bytes or 16 bytes of data shall be exclusive-ORed with the clear value of the session key contained in the key token identified by the **generating_key_identifier** parameter. If the **generating_key_identifier** parameter identifies a single-length key, then 8 bytes of data are exclusive-ORed. If the **generating_key_identifier** parameter identifies a double-length key, then 16 bytes of data are exclusive-ORed. The key token specified by the generating_key_identifier parameter must be of key type **DATA**, **DATAC**, **DATAM**, **DATAMV**, **MAC**, **MACVER**. The **data_decrypting_key_identifier** parameter must identify a null DES key-token. On input, the token identified by the **generated_key_identifier** parameter must identify a null DES key-token. The control vector contained in the output key token identified by the **generated_key_identifier** is the same as the control vector contained in the key token identified by the **generating_key_identifier**.
Key-wrapping method (One, optional)	
USECONFG	This is the default. Specifies to wrap the key using the configuration setting for the default wrapping method. The default wrapping method configuration setting may be changed using the TKE. This keyword is ignored for AES keys.
WRAP-ECB	Specifies to wrap the key using the legacy wrapping method.
WRAP-ENH	Specifies to wrap the key using the enhanced wrapping method.
Translation control (Optional). This is valid only with key-wrapping method **WRAP-ENH** or with **USECONFG** when the default wrapping method is **WRAP-ENH**. This option cannot be used on a key with a control vector valued to binary zeros.	
ENH-ONLY	Specifies to restrict the key from being wrapped with the legacy wrapping method once it has been wrapped with the enhanced wrapping method. Sets bit 56 (**ENH-ONLY**) of the control vector to B'1'.

generating_key_identifier

Direction: Input/Output
Type: String

The label or internal token of a key generating key. The type of key-generating key depends on the processing method.

data_length

Direction: Input
Type: Integer

The length of the *data* parameter that follows. Length depends on the processing method and the generated key.

data

Direction: Input
Type: String

Data input to the diversified key or session key generation process. Data depends on the processing method and the **generated_key_identifier**.

key_identifier

Direction: Input/Output
Type: String

This parameter is currently not used. It must be a 64-byte null token.

generated_key_identifier

Direction: Input/Output
Type: String

The internal token of an operational key, a skeleton token containing the control vector of the key to be generated, or a null token. A null token can be supplied if the **generated_key_identifier** is a **DKYGENKY** with a CV derived from the **generating_key_identifier**. A skeleton token or internal token is required when **generated_key_identifier** will not be a **DKYGENKY** key type or the processing method is not **SESS-XOR**. For **SESS-XOR**, this must be a null token. On output, this parameter contains the generated key.

Restrictions

The restrictions for CSNBDKG.

None.

Required commands

The CSNBDKG required commands.

This verb requires the following commands to be enabled in the active role based on the keyword specified for the process rule:

Rule-array keyword	Offset	Command
CLR8-ENC	X'0040'	**Diversified Key Generate - CLR8-ENC**
SESS-XOR	X'0043'	**Diversified Key Generate - SESS-XOR**
TDES-DEC	X'0042'	**Diversified Key Generate - TDES-DEC**
TDES-ENC	X'0041'	**Diversified Key Generate - TDES-ENC**
TDES-XOR	X'0045'	**Diversified Key Generate - TDES-XOR**
TDESEMV2 or **TDESEMV4**	X'0046'	**Diversified Key Generate - TDESEMV2/TDESEMV4**
WRAP-ECB or **WRAP-ENH** and default key-wrapping method setting does not match keyword	X'013D'	**Diversified Key Generate - Allow wrapping override keywords**

When a key-generating key of key type **DKYGENKY** is specified with control vector bits (19 - 22) of B'1111', the **Diversified Key Generate - DKYGENKY - DALL** command (offset X'0290') must also be enabled in the active role.

Note: A role with offset X'0290' enabled can also use the PIN Change/Unblock verb with a **DALL** key.

When using the **TDES-ENC** or **TDES-DEC** modes, you can specifically enable generation of a single-length key or a double-length key with equal key-halves (an effective single-length key) by enabling the **Diversified Key Generate - Single length or same halves** command (offset X'0044').

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

Usage notes for CSNBDKG.

Refer to Chapter 19, "Control vectors and changing control vectors with the Control Vector Translate verb," on page 897 for information on the control vector bits for the DKG key generating key.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBDKGJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBDKGJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  hikmNativeInteger rule_array_count,
  byte[]            rule_array,
  byte[]            generating_key_identifier,
  hikmNativeInteger data_length,
  byte[]            data,
  byte[]            data_decrypting_key_identifier,
  byte[]            generated_key_identifier);
```

Diversified Key Generate2 (CSNBDKG2)

The Diversified Key Generate2 service generates an AES key based on a function of a key-generating key, the process rule, and data that you supply.

To use this service, specify:

- the rule array keyword to select the diversification process
- the operational AES key-generating key from which the diversified keys are generated:
  - Key usage field 1 determines the type of key that is generated and restricts the use of this key to the key-diversification process.
  - Key usage field 2 contains a flag to determine how key usage fields 3 through 6 control the key usage fields of the generated key.
    - When the flag is on, the key usage fields of the DKYGENKY must be equal to the key usage fields of the generated key (KUF-MBE, meaning: *key usage fields must be equal*).
    - When the flag is off, the key usage fields of the DKYGENKY limit the values of the key usage fields of the generated key (KUF-MBP, meaning: *key usage fields must be permitted*).

    For the service to be valid, the generated key cannot have a usage that is not enabled in the DKYGENKY key. The UDX-ONLY bit is always treated as *must be equal*.
  - Key usage fields 3 through 6 in the key generating key indicate the key usage attributes for the key to be generated.

Note: The only exception to this rule is when the type of key to diversify is D-ALL.

- the data and length of data used in the diversification process
- the AES key token with a suitable key usage field for receiving the diversified key.

Format

The format of CSNBDKG2.

```
CSNBDKG2(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             generating_key_identifier_length,
             generating_key_identifier,
             derivation_data_length,
             derivation_data,
             reserved1_length,
             reserved1,
             reserved2_length,
             reserved2,
             generated_key_identifier1_length,
             generated_key_identifier1,
             generated_key_identifier2_length,
             generated_key_identifier2)
```

Parameters

The parameter definitions for CSNBDKG2.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. This value must be 1 or 2.

rule_array

Direction: Input
Type: String array

The keyword that provides control information to the verb. The only **rule_array** keyword is described in Table 36.

Table 36. Keyword for Diversified Key Generate2 control information

Keyword	Description
Diversification Process (One required)	

Table 36. Keyword for Diversified Key Generate2 control information (continued)

Keyword	Description
KDFFM-DK (Release 4.4 or later)	Specifies to use the DK version of Key Derivation Function (KDF) in Feedback Mode (NIST SP 800-108), as specified in *DK Kryptographie Teil 1: Empfohlene kryptographische Algorithmen*, to generate a bank specific Issuer Master Key. The generated Issuer Master Key (keying material) can be used to derive an ICC master key. This method uses AES CMAC to encipher 16 - 40 bytes of derivation data with the k-bit diversified key generating key (banking association specific master key) to produce a k-bit generated Bank specific Issuer Master Key, where k = 128, 192, or 256.
MK-OPTC (Release 4.4 or later)	Specifies to use the EMV Master Key Derivation Option C, as specified in *EMV Integrated Circuit Card Specifications for Payments Systems*, to generate an ICC master key. The generated ICC master key (keying material) can be used for Application Cryptogram generation or verification, issuer authentication, and secure messaging. This method uses AES in ECB mode to encipher the 16 bytes of derivation data with the k-bit diversified key generating key (Issuer Master Key) to produce a k-bit generated ICC master key, where k = 128, 192, or 256.
SESS-ENC	A session key is created by enciphering a 16-byte diversification value with the k-bit AES key-generating key to produce a k-bit AES session key using the AES algorithm in ECB mode, where k is 128, 192 or 256 bits.
Bit length of generated key (One, optional). Release 4.4 or later. Valid only with the KDFFM-DK keyword. Default is to use the bit length of the generating key as the bit length of the generated key.	
KLEN128	Specifies the bit length of the generated key to be 128.
KLEN192	Specifies the bit length of the generated key to be 192, allowed if and only if the bit length of the generating key is greater than or equal to 192. See "Required commands" on page 157.
KLEN256	Specifies the bit length of the generated key to be 256, allowed if and only if the bit length of the generating key is 256. See "Required commands" on page 157.

generating_key_identifier_length

Direction: Input
Type: Integer

Length of the **generating_key_identifier** parameter in bytes. If the **generating_key_identifier** contains a label, the value must be 64. Otherwise, the value must be at least the actual token length, up to 725.

generating_key_identifier

Direction: Input/Output
Type: String

The identifier of the key-generating key. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES and the key type must be DKYGENKY. The key usage field indicates the key type of the generated key.

If SESS-ENC is specified, the clear length of the generated key is equal to the clear length of the generating key. Also,beginning with Release 4.4, the sequence level can be set to DKYL0, DKYL1, or DKYL2 in the key usage field 2.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

If the **rule_array** parameter specifies a diversification process of KDFFM-DK, the key-derivation sequence level of the generating key must be DKYL2. Otherwise, if KDFFM-DK is not specified, any sequence level is allowed for the generating key.

derivation_data_length

Direction: Input
Type: Integer

Length of the **derivation_data** parameter in bytes. The value must be in the range 16 - 40 for the diversification process keyword KDFFM-DK, otherwise the value must be 16.

derivation_data

Direction: Input
Type: String

The derivation data to be used in the key generation process. This data is often referred to as the diversification data. For SESS-ENC, the derivation data is 16 bytes long. Note that if SESS-ENC is specified and the length of the key generating key is 192 bits or 256 bits, the data is manipulated in conformance with the **EMV Common Session Key Derivation Option**.

reserved1_length

Direction: Input
Type: Integer

Length of the **reserved1** parameter in bytes. The value must be 0.

reserved1

Direction: Input
Type: String

This parameter is ignored.

reserved2_length

Direction: Input
Type: Integer

Length of the **reserved2** parameter in bytes. The value must be 0.

reserved2

Direction: Input
Type: String

This parameter is ignored.

generated_key_identifier1_length

Direction: Input/Output
Type: Integer

On input, this parameter specifies the length in bytes of the buffer for the **generated_key_identifier1** parameter. The maximum value is 725 bytes.

On output, the parameter holds the actual length in bytes of the **generated_key_identifier1** parameter.

generated_key_identifier1

Direction: Input/Output
Type: String

A pointer to a string variable containing an internal variable-length symmetric key-token or the key label of such a record in AES key-storage.

On input, identify a null key token or a skeleton key token that specifies the desired attributes of the key on output. The key token identified by **generating_key_identifier1** determines whether on input the **generating_key_identifier1** can identify a null key token or a skeleton key token. See Table 37:

Table 37. Generating and generated key tokens

Iunpt generating key token	Iunpt generated key token	Output generated key token
DKYGENKY, DKYL0, type of key to diversify D-ALL	Null AES key token not allowed; AES skeleton key token required.	Key type same as skeleton; diversified key final.
DKYGENKY, DKYL0, type of key to diversify not D-ALL	Either null AES key token or AES skeleton key token required.	Key type determined by input generated key token type of key to diversify; if null key token on input, output key token will have attributes based on the related generated key usage fields of the input generating key token, otherwise the output key token has attributes of the input skeleton key token.
DKYGENKY, DKYL1, any type of key to diversify	Null AES key token required; AES skeleton key token not allowed.	Same as input generating key token except DKYL0 and with new level of diversified key.
DKYGENKY, DKYL2, any type of key to diversify	Null AES key token required; AES skeleton key token not allowed.	Same as input generating key token except DKYL1 and with new level of diversified key.

Note:

1. If the supplied generated key-token contains a key, the key value and length are ignored and overwritten.
2. The key type must match what the generating key indicates can be created in the key generating key usage field at offset 45.
3. The key usage fields in the generated key must meet the requirements (`KUF must be equal (KUF-MBE)` or `KUF must be permissible (KUF-MBP)`) of the corresponding key usage fields in the generating key unless D-ALL is specified in the generating key. A flag bit in the DKYGENKY key-usage field 2 determines whether the key-usage field level of control is KUF-MBE or KUF-MBP.
4. If authorized by access control, D-ALL permits the derivation of several different keys. See “Required commands” on page 157.

generated_key_identifier2_length

Direction: Input
Type: Integer

Length of the **generated_key_identifier2** parameter in bytes. The value must be 0.

generated_key_identifier2

Direction: Input/Output
Type: String

This parameter is ignored.

Restrictions

The restrictions for CSNBDKG2.

None.

Required commands

The CSNBDKG2 required commands.

The Diversified Key Generate2 verb requires the following commands to be enabled in the active role:

Rule-array keyword	Offset	Command
KDFFM-DK (Release 4.4 or later)	X'02D3'	**Diversified Key Generate2 - KDFFM-DK**
KLEN192 and **KLEN256** (Release 4.4 or later)	X'02D4'	**Allow DKG2 Generated Key Length Option with KDFFM-DK Keyword**
MK-OPTC (Release 4.4 or later)	X'02D2'	**Diversified Key Generate2 - MK-OPTC**
SESS-ENC	X'02CC'	**Diversified Key Generate2 - AES EMV1 SESS**

An additional command is required when the key usage fields of the key token identified by the **generating_key_identifier** parameter specify a type of key to diversify of D-ALL (any type of key allowed). In this case, the verb requires the **Diversified Key Generate2 - DALL** command (offset X'02CD') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

Usage notes for CSNBDKG2.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBDKG2J.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBDKG2J(
  hikmNativeInteger  return_code,
  hikmNativeInteger  reason_code,
  hikmNativeInteger  exit_data_length,
  byte[]             exit_data,
```

```
| hikmNativeInteger  rule_array_count,
| byte[]             rule_array,
| hikmNativeInteger  generating_key_identifier_length,
| byte[]             generating_key_identifier,
| hikmNativeInteger  derivation_data_length,
| byte[]             derivation_data,
| hikmNativeInteger  reserved1_length,
| byte[]             reserved1,
| hikmNativeInteger  reserved2_length,
| byte[]             reserved2,
| hikmNativeInteger  generated_key_identifier1_length,
| byte[]             generated_key_identifier1,
| hikmNativeInteger  generated_key_identifier2_length,
| byte[]             generated_key_identifier2);
```

EC Diffie-Hellman (CSNDEDH)

Use the EC Diffie-Hellman verb to create symmetric key material from a pair of Elliptic Curve Cryptography (ECC) keys using the Elliptic Curve Diffie-Hellman (ECDH) protocol and "Z" - The "secret" material output from EC Diffie-Hellman process.

For more information, see “EC Diffie-Hellman key agreement models” on page 44.

ECDH is a key-agreement protocol that allows two parties, each having an elliptic curve public-private key pair, to establish a shared secret over an insecure channel. This shared secret is used to derive another symmetric key. The ECDH protocol is a variant of the Diffie-Hellman protocol using elliptic curve cryptography. ECDH derives a shared secret value from a secret key owned by an Entity A and a public key owned by an Entity B, when the keys share the same elliptic curve domain parameters. Entity A can be either the initiator of a key-agreement transaction, or the responder in a scheme. If two entities both correctly perform the same operations with corresponding keys as inputs, the same shared secret value is produced.

Both parties must create an ECC public-private key pair. See “PKA Key Token Build (CSNDPKB)” on page 644 and “PKA Key Generate (CSNDPKG)” on page 635. A key can be internal or external, as well as encrypted or in the clear. Both keys must have the same elliptic curve domain parameters (curve type and key size):

- Brainpool (key size 160, 192, 224, 256, 320, 384, or 512)
- Prime (key size 192, 224, 256, 384, or 521)

In addition to having the same elliptic curve domain parameters, the keys must have their key-usage field set to permit key establishment (either KEY-MGMT or KM-ONLY). See “ECC key token” on page 796.

To use this verb, specify the following:

- One to five rule-array keywords:
  - A required key-agreement keyword
  - An optional transport key-type (required if *output_KEK_key_identifier* is a label) that identifies which key-storage dataset contains the output KEK key-token
  - An optional output key-type (required if *output_key_identifier* is a label) that identifies which key-storage dataset contains the output key-token

  - When the output is a DES key-token, an optional key-wrapping method and an optional translation control keyword
- The internal or external ECC key-token containing the private key (public-private key pair).

  If the private key is in an external key-token and is not in the clear, specify the internal KEK that was used to wrap the private key. Otherwise, specify a private KEK key-length of 0.
- An internal or external ECC key-token containing the public key (public key only or public-private key pair) of the other party.

  If the public key is in a key token that contains a public-private key pair, only the public-key portion is used. No attempt is made to decrypt the private key.
- From 8 - 64 bytes of party information data of the initiator and the responder entities, according to NIST SP800-56A Section 5.8
- The number of bits of key material, from 64 - 256, to derive and place in the provided output key-token
- The length in bytes of the buffer for the output key-identifier
- An internal or external skeleton key-token to be used as input for the output key-token

  The skeleton key-token must be an AES key or a DES key, as shown in the following table:

Table 38. Skeleton key-tokens

Algorithm	Token version number	Key type (see note)
AES	X'04' (legacy fixed-length symmetric key-token)	DATA
	X'05' (variable-length symmetric key-token)	• CIPHER Both parties can provide any combination of encryption or decryption for key-usage field. • EXPORTER or IMPORTER Both parties can provide any combination of EXPORTER or IMPORTER.
DES	X'00', X'01', X'03' (legacy fixed-length symmetric key-token)	• CIPHER, DECIPHER, ENCIPHER Both parties can provide any combination of Encipher or Decipher key-usage bits in the control vector. • EXPORTER or IMPORTER Both parties can provide any combination of EXPORT or IMPORT key-usage bits in the control vector.
Note: Except as otherwise noted, both parties must provide identical skeleton key tokens for the output key in order to derive identical keys. For legacy skeletons, control vector parity bits are not used in the key derivation process.		

  If the skeleton key-token is an external key-token, specify the internal KEK to be used to wrap the output key-token. Otherwise, specify an output KEK length of 0.

  If the *output_key_identifier* specifies a DES key-token, then the *output_KEK_key_token* must identify a legacy DES KEK. Otherwise it must identify a variable-length symmetric AES KEK key-token.

The output from this verb can be one of the following formats:

- Internal CCA Token (DES or AES): AES keys are in the variable-length symmetric key token format. DES keys are in the DES external key token format.
- External CCA key token (DES or AES): AES keys are in the variable-length symmetric key token format. DES keys are in the DES external key token format.
- "Z"– The "secret" material output from EC Diffie-Hellman process.

The **PASSTHRU** service is provided as a convenience to users who wish to implement their own key completion process in host application software. While the "Z" derivation process is not reversible (the ECC keys cannot be discovered by obtaining "Z") there is a level of security compromise associated with returning the clear "Z" to the application. Future derivations for CCA key tokens using ECC keys previously used in **PASSTHRU** must be considered to have lower security, and using the same ECC keys for **PASSTHRU** as for **DERIV01** is strongly discouraged. It should also be noted that since "Z" is the secret material, returning it in the clear to the host application reduces security level of the "Z" from the HSM level to the host application level, and keys made from this material should not be regarded as having any HSM protection.

For more information, see “EC Diffie-Hellman key agreement models” on page 44.

Format

The format of CSNDEDH.

```
CSNDEDH(
           return_code,
           reason_code,
           exit_data_length,
           exit_data,
           rule_array_count,
           rule_array,
           private_key_identifier_length,
           private_key_identifier,
           private_KEK_key_identifier_length,
           private_KEK_key_identifier,
           public_key_identifier_length,
           public_key_identifier,
           chaining_vector_length,
           chaining_vector,
           party_info_length,
           party_info,
           key_bit_length,
           reserved_1_length,
           reserved_1,
           reserved_2_length,
           reserved_2,
           reserved_3_length,
           reserved_3,
           reserved_4_length,
           reserved_4,
           reserved_5_length,
           reserved_5,
           output_KEK_key_identifier_length,
           output_KEK_key_identifier,
           output_key_identifier_length,
           output_key_identifier)
```

Parameters

The parameter definitions for CSNDEDH.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. Valid values are 1 - 5.

rule_array

Direction: Input
Type: String array

A pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* keywords for this verb are shown below:

Table 39. Keywords for EC Diffie-Hellman control information

Keyword	Meaning
Key agreement (one required)	
DERIV01	Use input skeleton key-token and derive one element of any key pair. Denotes ANSI X9.63 protocol static unified model key-agreement scheme (see NIST SP800-56A). Initiator and responder must have a sufficient level of trust such that they each derive only one element of any key pair.
PASSTHRU	Skip key derivation step and return raw "Z" material. **Note:** This keyword is available only for Linux on z Systems.
Transport key-type (one, optional; one required if *output_KEK_key_identifier* is a label)	
OKEK-AES	The *output_KEK_key_identifier* represents an AES key-token.
OKEK-DES	The *output_KEK_key_identifier* represents a DES key-token.
Output key-type (one, optional; required if *output_key_identifier* is a label)	
KEY-AES	The outbound key-encrypting key represents an AES skeleton key-token.
KEY-DES	The outbound key-encrypting key represents a DES skeleton key-token.
Key-wrapping method (one, optional). DES only.	
USECONFG	This is the default. Specifies to wrap the key using the configuration setting for the default wrapping method. The default wrapping method configuration setting may be changed using the TKE. This keyword is ignored for AES keys.
WRAP-ECB	Specifies to wrap the key using the legacy wrapping method. This keyword is ignored for AES keys.
WRAP-ENH	Specifies to wrap the key using the enhanced wrapping method. Valid only for DES keys.
Translation control (optional). This is valid only with key-wrapping method **WRAP-ENH** or with **USECONFG** when the default wrapping method is **WRAP-ENH**. This option cannot be used on a key with a control vector valued to binary zeros.	
ENH-ONLY	Specifies to restrict the key from being wrapped with the legacy wrapping method after it has been wrapped with the enhanced wrapping method. Sets bit 56 (ENH-ONLY) of the control vector to B'1'.

private_key_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes in the *private_key_identifier* variable.

private_key_identifier

Direction: Input
Type: String

A pointer to a string variable containing an internal or external ECC key-token, or a key label identifying a key-storage record for such a token. The ECC key-token will contain a public-private key pair. Clear keys are allowed.

The ECC curve type and size must be the same as the type (Prime or Brainpool) and size of the ECC key-token specified by the *public_key_identifier* parameter. The key-usage flag of the ECC key-token identified by the *private_key_identifier* parameter must permit key establishment (either **KEY-MGMT** or **KM-ONLY**).

private_KEK_key_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes in the *private_KEK_key_identifier* variable. The maximum value is 900. If the *private_key_identifier* contains an internal ECC token, this value must be a zero.

private_KEK_key_identifier

Direction: Input
Type: String

A pointer to a string variable containing an internal KEK key-token, a key label identifying a key-storage record for such a key token, or a null token. The KEK key-token must be present if the key token specified by the *private_key_identifier* contains an external encrypted ECC key-token.

public_key_identifier_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes in the *public_key_identifier* variable.

Note that even though this variable is not currently updated on output, it is reserved as an output field for future use.

public_key_identifier

Direction: Input/Output
Type: String

A pointer to a string variable containing an ECC key-token, or a key label identifying a key-storage record for such a key token. The ECC curve type and size must be the same as the type and size of the ECC key-token specified by the *private_key_identifier* parameter. The *public_key_identifier* specifies the other party's ECC public key which is enabled for key management functions. If the *public_key_identifier* parameter identifies a key token containing a public-private

key pair, no attempt to decrypt the private part is made.

Note that even though this variable is not currently updated on output, it is reserved as an output field for future use.

chaining_vector_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes in the *chaining_vector* variable. This field is currently not used. The value must be 0.

chaining_vector

Direction: Input/Output
Type: String

A pointer to a string variable containing a buffer that is currently reserved.

party_info_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes in the *party_info* variable. Valid values are 0, or 8 - 64. The *party_info_length* must be 0 when the **PASSTHRU** rule-array keyword is specified.

party_info

Direction: Input/Output
Type: String

A pointer to a string variable containing combined entity identifier information, including nonce. This information must contain data of both entities according to NIST SP800-56A Section 5.8, when the **DERIV01** rule-array keyword is specified.

key_bit_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bits of key material to derive and place in the provided output key-token. The value must be 0 if the **PASSTHRU** rule-array keyword is specified. The value must be 64 - 256.

reserved_1_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes in the *reserved_1* variable. The value must be 0.

reserved_1

Direction: Input/Output
Type: String

A pointer to a string variable that is currently not used.

reserved_2_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes in the *reserved_2* variable. The value must be 0.

reserved_2

Direction: Input/Output
Type: String

A pointer to a string variable that is currently not used.

reserved_3_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes in the *reserved_3* variable. The value must be 0.

reserved_3

Direction: Input/Output
Type: String

A pointer to a string variable that is currently not used.

reserved_4_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes in the *reserved_4* variable. The value must be 0.

reserved_4

Direction: Input/Output
Type: String

A pointer to a string variable that is currently not used.

reserved_5_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes in the *reserved_5* variable. The value must be 0.

reserved_5

Direction: Input/Output
Type: String

A pointer to a string variable that is currently not used.

output_KEK_key_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes in the

output_KEK_key_identifier variable. The maximum value is 900. The *output_KEK_key_identifier_length* must be zero if *output_key_identifier* will contain an internal token or if the **PASSTHRU** rule-array keyword was specified.

output_KEK_key_identifier

Direction: Input
Type: String

A pointer to a string variable containing an internal KEK key-token, or a key label identifying a key-storage record for such a token. This parameter must identify a KEK key-token whenever the *output_key_identifier* specifies an external key-token. If the *output_key_identifier* specifies a DES key-token, then the *output_KEK_key_identifier* must identify a legacy DES KEK, otherwise it must identify a variable-length symmetric AES KEK key-token.

If this variable contains a key label, specify a transport key-type rule-array keyword (**OKEK-DES** or **OKEK-AES**) to identify which key-storage dataset contains the key token. If a transport key-type keyword is specified, it must match the type of key identified by this parameter, whether the key is in key storage or not.

If the *output_KEK_key_identifier* specifies a legacy DES KEK, then the key token must contain either an EXPORTER control vector with bit 21 on (EXPORT) or an IMPORTER control vector with bit 21 set to B'1' (IMPORT). The XLATE bit (bit 22) is not checked. Similarly, if the *output_KEK_key_identifier* identifies a variable-length symmetric AES KEK, then the KEK must be have a key type of EXPORTER or IMPORTER. Key-usage field 1 of the KEK must be set so that the key can be used for EXPORT or IMPORT. In addition, key-usage field 4 must be set so that the key can wrap DERIVATION class keys.

output_key_identifier_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes in the *output_key_identifier* variable. On input, the *output_key_identifier* length variable must be set to the total size of the buffer pointed to by the *output key identifier* parameter. On output, this variable contains the number of bytes of data returned by the verb in the *output_key_identifier* variable. The maximum allowed value is 900 bytes.

output_key_identifier

Direction: Input/Output
Type: String

A pointer to a string variable. On input, it must contain an internal or an external skeleton key-token, or a key label identifying a key-storage record for such a token. The skeleton key-token must be one of the following:

DES (legacy DES key-token)

- **CIPHER**, **DECIPHER**, or **ENCIPHER**
- **EXPORTER** or **IMPORTER**

AES

- **DATA** (legacy AES key-token)
- **CIPHER** (variable-length symmetric key-token) with key-usage field set so that the key can be used for decryption, encryption, or both)

- **EXPORTER** or **IMPORTER** (variable-length symmetric key-token)

On successful completion, this variable contains either:

- An updated key-token that contains the generated symmetric key material, or the key label of the key-token that has been updated in key storage.
- "Z" data (in the clear) if the **PASSTHRU** rule-array keyword was specified.

If this variable contains an external key-token on input, then the *output_KEK_key_identifier* is used to securely encrypt it for output. If this variable contains a key label, specify an output key-type rule-array keyword (**KEY-DES** or **KEY-AES**) to identify which key-storage dataset contains the key token. If an output key-type keyword is specified, it must match the type of key identified by this parameter, whether the key is in key storage or not.

If this variable identifies an external DES key-token, then the *output_KEK_key_identifier* must identify a DES KEK key-token. If this variable is present and identifies an external key-token other than a DES key-token, then the *output_KEK_key_identifier* must identify an AES KEK key-token.

Restrictions

The restrictions for CSNDEDH.

The NIST security strength requirements are enforced, with respect to ECC curve type (input) and derived key-length. See “Required commands” about how you can override this enforcement.

This table lists the valid key bit lengths and the minimum curve size required for each of the supported output key types:

Output key ID type	Valid key bit lengths	Minimum curve required
DES	64	P160
	128	P160
	192	P224
AES	128	P256
	192	P384
	256	P512

Required commands

The CSNDEDH required commands.

This table describes access control points that the EC Diffie-Hellman verb must have enabled in the active role under certain circumstances.

Command	Offset	When required
ECC Diffie-Hellmann	X'0360'	When using the EC Diffie-Hellman verb
ECC Diffie-Hellman - Allow key wrap override	X'0362'	If the *output_key_identifier* parameter identifies a DES key-token, and the wrapping method specified is not the default method

Command	Offset	When required
Prohibit weak wrapping - Transport keys This command affects multiple verbs. See Chapter 22, "Access control points and verbs," on page 953.	X'0328'	To disable the wrapping of a stronger key with a weaker key
Warn when weak wrap - Transport keys The command **Prohibit weak wrapping - Transport keys** (offset X'0328') overrides this command.	X'032C'	To receive a warning against the wrapping of a stronger key with a weaker key
ECC Diffie-Hellman - Prohibit weak key generate	X'036F'	To disable a weaker key from being used to generate a stronger key
ECC Diffie-Hellman - Allow PASSTHRU	X'0361'	When specifying the **PASSTHRU** rule-array keyword.

Depending on curve type, each length of *p* in bits contained in the ECC private-key section and the ECC public-key section must have the following command enabled in the active role:

Curve type	Length of prime *p* in bits	Offset	Command
Brainpool	160 (X'00A0')	X'0368'	**ECC Diffie-Hellmann - Allow BP Curve 160**
	192 (X'00C0')	X'0369'	**ECC Diffie-Hellmann - Allow BP Curve 192**
	224 (X'00E0')	X'036A'	**ECC Diffie-Hellmann - Allow BP Curve 224**
	256 (X'0100')	X'036B'	**ECC Diffie-Hellmann - Allow BP Curve 256**
	320 (X'0140')	X'036C'	**ECC Diffie-Hellmann - Allow BP Curve 320**
	384 (X'0180')	X'036D'	**ECC Diffie-Hellmann - Allow BP Curve 384**
	512 (X'0200')	X'036E'	**ECC Diffie-Hellmann - Allow BP Curve 512**
Prime	192 (X'00C0')	X'0363'	**ECC Diffie-Hellmann - Allow Prime Curve 192**
	224 (X'00E0')	X'0364'	**ECC Diffie-Hellmann - Allow Prime Curve 224**
	256 (X'0100')	X'0365'	**ECC Diffie-Hellmann - Allow Prime Curve 256**
	384 (X'0180')	X'0366'	**ECC Diffie-Hellmann - Allow Prime Curve 384**
	521 (X'0209')	X'0367'	**ECC Diffie-Hellmann - Allow Prime Curve 521**

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

Usage notes for CSNDEDH.

The **PASSTHRU** service is provided as a convenience to users who wish to implement their own key completion process in host application software. While the "Z" derivation process is not reversible (the ECC keys cannot be discovered by obtaining "Z") there is a level of security compromise associated with returning the clear "Z" to the application. Future derivations for CCA key tokens using ECC keys previously used in **PASSTHRU** must be considered to have lower security, and using the same ECC keys for **PASSTHRU** as for **DERIV01** is strongly discouraged. It should also be noted that since "Z" is the secret material, returning it in the clear to the host application reduces security level of the "Z" from the HSM level to the host application level, and keys made from this material should not be regarded as having any HSM protection.

For more information, see “EC Diffie-Hellman key agreement models” on page 44.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDEDHJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void  CSNDEDHJ(
          hikmNativeInteger   return_code,
          hikmNativeInteger   reason_code,
          hikmNativeInteger   exit_data_length,
          byte[]              exit_data,
          hikmNativeInteger   rule_array_count,
          byte[]              rule_array,
          hikmNativeInteger   private_key_identifier_length,
          byte[]              private_key_identifier,
          hikmNativeInteger   private_KEK_key_identifier_length,
          byte[]              private_KEK_key_identifier,
          hikmNativeInteger   public_key_identifier_length,
          byte[]              public_key_identifier,
          hikmNativeInteger   chaining_vector_length,
          byte[]              chaining_vector,
          hikmNativeInteger   party_info_length,
          byte[]              party_info,
          hikmNativeInteger   key_bit_length,
          hikmNativeInteger   reserved_1_length,
          byte[]              reserved_1,
          hikmNativeInteger   reserved_2_length,
          byte[]              reserved_2,
          hikmNativeInteger   reserved_3_length,
          byte[]              reserved_3,
          hikmNativeInteger   reserved_4_length,
          byte[]              reserved_4,
          hikmNativeInteger   reserved_5_length,
          byte[]              reserved_5,
          hikmNativeInteger   output_KEK_key_identifier_length,
          byte[]              output_KEK_key_identifier,
          hikmNativeInteger   output_key_identifier_length,
          byte[]              output_key_identifier);
```

Key Export (CSNBKEX)

Use the Key Export verb to re-encipher any type of key (except an IMP-PKA) from encryption under a master key variant to encryption under the same variant of an exporter key-encrypting key. The format of .

The re-enciphered key can be exported to another system.

If the key to be exported is a **DATA** key, the Key Export verb generates a key token with the same key length as the input token's key.

This verb supports the no-export bit that the Prohibit Export verb sets in the internal token.

Format

The format of CSNBKEX.

```
CSNBKEX(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             key_type,
             source_key_identifier,
             exporter_key_identifier,
             target_key_identifier)
```

Parameters

The parameter definitions for CSNBKEX.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

key_type

Direction: Input
Type: String

The parameter is an 8-byte field that contains either a key type value or the keyword **TOKEN**. The keyword is left-aligned and padded on the right with blanks.

If the key type is **TOKEN**, CCA determines the key type from the control vector (CV) field in the internal key token provided in the *source_key_identifier* parameter.

Key type values for the Key Export verb are:

CIPHER	DATAMV	MAC
CIPHERXI	DECIPHER	MACVER
CIPHERXL	ENCIPHER	OKEYXLAT
CIPHERXO	EXPORTER	OPINENC
DATA	IKEYXLAT	PINGEN
DATAC	IMPORTER	PINVER
DATAM	IPINENC	TOKEN

For information about the meaning of the key types, see Table 4 on page 40.

source_key_identifier

Direction: Input
Type: String

A 64-byte string of the internal key token that contains the key to be re-enciphered. This parameter must identify an internal key token in application storage, or a label of an existing key in the DES key storage file.

If you supply **TOKEN** for the *key_type* parameter, CCA looks at the control vector in the internal key token and determines the key type from this information. If you supply **TOKEN** for the *key_type* parameter and supply a label for this parameter, the label must be unique in the DES key storage file.

exporter_key_identifier

Direction: Input/Output
Type: String

A 64-byte string of the internal key token or key label that contains the exporter key-encrypting key. This parameter must identify an internal key token in application storage, or a label of an existing key in the key storage file.

If the NOCV bit is on in the internal key token containing the key-encrypting key, the key-encrypting key itself (not the key-encrypting key variant) is used to encipher the generated key.

Control vectors are explained in "Control vector" on page 36 and the NOCV bit is shown in Table 209 on page 772.

target_key_identifier

Direction: Input/Output
Type: String

The 64-byte field external key token that contains the re-enciphered key. The re-enciphered key can be exchanged with another cryptographic system.

Restrictions

The restrictions for CSNBKEX.

For security reasons, requests will fail by default if they use an equal key halves exporter to export a key with unequal key halves. You must have access control point 'Key Export - Unrestricted' explicitly enabled if you want to export keys in this manner.

Required commands

The CSNBKEX required commands.

This verb requires the **Key Export** command (offset X'0013') to be enabled in the active role.

By also specifying the **Key Export - Unrestricted** command (offset X'0276'), you can permit a less secure mode of operation that enables an equal key-halves EXPORTER key-encrypting-key to export a key having unequal key-halves (key parity bits are ignored).

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

Usage notes for CSNBKEX.

For Key Export, you can use the following combinations of parameters:

- A valid key type in the *key_type* parameter and an internal key token in the *source_key_identifier* parameter. The key type must be equivalent to the control vector specified in the internal key token.
- A *key_type* parameter of **TOKEN** and an internal key token in the *source_key_identifier* parameter. The *source_key_identifier* can be a label with **TOKEN** only if the label name is unique in the key storage. The key type is extracted from the control vector contained in the internal key token.
- A valid key type in the *key_type* parameter, and a label in the *source_key_identifier* parameter.

If internal key tokens are supplied in the *source_key_identifier* or *exporter_key_identifier* parameters, the key in one or both tokens can be re-enciphered. This occurs if the master key was changed since the internal key token was last used. The return and reason codes that indicate this do *not* indicate which key was re-enciphered. Therefore, assume both keys have been re-enciphered.

Existing internal tokens created with key type **MACD** must be exported with either a **TOKEN** or **DATAM** key type. The external CV will be **DATAM** CV. The **MACD** key type is not supported.

To export a double-length MAC generation or MAC verification key, it is recommended that a key type of **TOKEN** be used.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKEXJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBKEXJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  byte[]            key_type,
  byte[]            source_key_identifier,
  byte[]            exporter_key_identifier,
  byte[]            target_key_token);
```

Key Generate (CSNBKGN)

Use the Key Generate verb to generate an AES key of type DATA, or either one or two odd parity DES keys of *any* type.

The DES keys can be single-length (8-byte), double-length (16-byte), or, in the case of **DATA** keys, triple-length (24-byte). The AES keys can be 16, 24 or 32 bytes in length. The Key Generate verb does not produce keys in clear form; all keys are returned in encrypted form. When two keys are generated (DES only), each key has the same clear value, although this clear value is not exposed outside the secure cryptographic feature.

For AES, the verb returns only one copy of the key, enciphered under the AES master key. For DES, the verb selectively returns one copy of the key or two, with each copy enciphered under a user-specified DES key-encrypting key.

This verb returns the key to the application program that called it and the application program can then use the CCA key storage verbs to store the key in the key storage file.

Format

The format of CSNBKGN.

```
CSNBKGN(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             key_form,
             key_length,
             key_type_1,
             key_type_2,
             kek_key_identifier_1,
             kek_key_identifier_2,
             generated_key_identifier_1,
             generated_key_identifier_2)
```

Parameters

The parameter definitions for CSNBKGN.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

key_form

Direction: Input
Type: String

A 4-byte keyword that defines the type of key you want generated. This parameter also specifies if each key should be returned for either operational, importable, or exportable use. The keyword must be in a 4-byte field, left-aligned, and padded with blanks.

The possible key forms are:

Operational (OP)
The key is used for cryptographic operations on the local system. Operational keys are protected by master key variants and can be stored in the CCA key storage file or held by applications in internal key tokens.

Importable (IM)
The key is stored with a file or sent to another system. Importable keys are protected by importer key-encrypting keys.

Exportable (EX)
The key is transported or exported to another system and imported there for use. Exportable keys are protected by exporter key-encrypting keys and cannot be used by CCA verb.

Importable and exportable keys are contained in external key tokens. For more information on key tokens, refer to "Key token" on page 31.

The first two characters refer to *key_type_1*. The next two characters refer to *key_type_2*.

The following keywords are allowed: **OP**, **IM**, **EX**, **OPIM**, **OPEX**, **IMEX**, **EXEX**, **OPOP**, and **IMIM**. See Table 40 for their meanings.

Table 40. Keywords for the Key Generate verb key_form parameter

Keyword	Description
EX	One key that can be sent to another system.
EXEX	A key pair; both keys to be sent elsewhere, possibly for exporting to two different systems. The key pair has the same clear value.
IM	One key that can be locally imported. The key can later be imported onto this system to make it operational.
IMEX	A key pair to be imported; one key to be imported locally and one key to be sent elsewhere. Both keys have the same clear value.
IMIM	A key pair to be imported; both keys to be imported locally at a later time.
OP	One operational key. The key is returned to the caller in the key token format.
OPEX	A key pair; one key that is operational and one key to be sent from this system. Both keys have the same clear value.
OPIM	A key pair; one key that is operational and one key to be imported to the local system. Both keys have the same clear value. On the other system, the external key token can be imported to make it operational.
OPOP	A key pair; normally with different control vector values.

The key forms are defined as follows:

Operational (OP)
The key value is enciphered under a master key. The result is placed into an internal key token. The key is then operational at the local system.

Importable (IM)
The key value is enciphered under an importer key-encrypting key. The result is placed into an external key token.

Exportable (EX)
The key value is enciphered under an exporter key-encrypting key. The result is placed into an external key token. The key can then be transported or exported to another system and imported there for use. This key form cannot be used by any CCA verb.

The keys are placed into tokens that the *generated_key_identifier_1* and *generated_key_identifier_2* parameters identify.

Valid key type combinations depend on the key form. See Table 44 on page 179 for valid key combinations.

key_length

Direction: Input
Type: String

An 8-byte value that defines the length of the key as being 8, 16, 24 or 32 bytes. The keyword must be left-aligned and padded on the right with blanks. You must supply one of the key length values in the *key_length* parameter.

Table 41 on page 174 lists the key lengths used for various key types.

Table 41. Key length values for the Key Generate verb

Value	Description	Algorithm
SINGLE, **SINGLE-R**, or **KEYLN8**	Single length (8-byte or 64-bit) key	DES
DOUBLE or **KEYLN16**	Double length (16-byte or 128-bit) key	AES or DES
DOUBLE-O	Double length (16-byte or 128-bit) key	DES
KEYLN24	Triple length (24-byte or 192-bit) key	AES or DES
KEYLN32	32-byte (256-bit) key	AES

AES keys allow only **KEYLN16**, **KEYLN24**, and **KEYLN32**. To generate a 128-bit AES key, specify *key_length* as **KEYLN16**. For 192-bit AES keys specify *key_length* as **KEYLN24**. A 256-bit AES key requires a *key_length* of **KEYLN32**. All AES keys are **DATA** keys.

Keys with a length of 32 bytes have four 8-byte key parts. This key length is valid only for AES keys. To generate a 32-byte AES key with four different values to be the basis of each key part, specify *key_length* as **KEYLN32**.

To generate a single-length key, specify *key_length* as **SINGLE** or **KEYLN8**.

Double-length (16-byte) keys have an 8-byte left half and an 8-byte right half. Both halves can have identical clear values or not. If you want the same value to be used in both key halves (called *replicated key values*), specify a *key_length* of **SINGLE**, **SINGLE-R**, or **KEYLN8**. If you want different values to be the basis of each key half, specify a *key_length* of **DOUBLE** or **KEYLN16**.

Triple-length (24-byte) keys have three 8-byte key parts. This key length is valid for DATA keys only. To generate a triple-length **DATA** key with three different values to be the basis of each key part, specify a *key_length* of **KEYLN24**.

Use **SINGLE** or **SINGLE-R** if you want to create a DES transport key that you would use to exchange **DATA** keys with a PCF system. Because PCF does not use double-length transport keys, specify **SINGLE** so that the effects of multiple encipherment are nullified.

When generating an AKEK, the *key_length* parameter is ignored. The AKEK key length (8-byte or 16-byte) is determined by the skeleton token created by the Key Token Build verb and provided in the *generated_key_identifier_1* parameter.

The key length specified must be consistent with the key length indicated by the token you supply. For DES keys, this length is a field in the control vector. For AES keys, the length is an explicit field in the token. Table 42 shows the valid key lengths for each key type. An **X** indicates that a key length is permitted for a key type. A **Y** indicates that the key generated will be a double-length key with replicated key values. It is preferred that **SINGLE-R** be used for this result.

Table 42. Key Generate - key lengths for each key type

Key Type	SINGLE (KEYLN8)	SINGLE-R	DOUBLE (KEYLN16)	DOUBLE-O	Triple (KEYLN24)	(KEYLN32)
AESDATA			X		X	X
MAC	X	X	X	X		
MACVER	X	X	X	X		

Table 42. Key Generate - key lengths for each key type (continued)

Key Type	SINGLE (KEYLN8)	SINGLE-R	DOUBLE (KEYLN16)	DOUBLE-O	Triple (KEYLN24)	(KEYLN32)
DATA	X	X	X	X	X	
DATAM **DATAMV**			X X	X X		
EXPORTER **IMPORTER**	Y Y	X X	X X	X X		
IKEYXLAT **OKEYXLAT**	Y Y	X X	X X	X X		
CIPHER **DECIPHER** **ENCIPHER**	X X X		X X X	X X X		
IPINENC **OPINENC** **PINGEN** **PINVER**	Y Y Y Y	X X X X	X X X X	X X X X		
CVARDEC* **CVARENC*** **CVARPINE*** **CVARXCVL*** **CVARXCVR***	X X X X X		X X X X X	X X X X X		
DKYGENKY* **KEYGENKY***		X X	X X	X X		
CIPHERXI **CIPHERXL** **CIPHERXO**				X X X		

Note: Key types marked with an asterisk (*) are requested through the use of the **TOKEN** keyword and specifying a proper control vector in a key token.

key_type_1

Direction: Input
Type: String

An 8-byte keyword from the following group:

```
AESDATA     DATA        ENCIPHER    MACVER
AESTOKEN    DATAC       EXPORTER    OKEYXLAT
CIPHER      DATAM       IKEYXLAT    OPINENC
CIPHERXI    DATAMV      IMPORTER    PINGEN
CIPHERXL    DATAXLAT    IPINENC     PINVER
CIPHERXO    DECIPHER    MAC
```

or the keyword **TOKEN**.

For information on the meaning of the key types, see Table 4 on page 40.

Use the **key_type_1** parameter for the first, or only key, that you want generated. The keyword must be left-aligned and padded with blanks. Valid type combinations depend on the key form.

If **key_type_1** is **TOKEN**, CCA examines the control vector (CV) field in the **generated_key_identifier_1** parameter to derive the key type. When

key_type_1 is **TOKEN**, CCA does not check for the length of the key for DATA keys. Instead, it uses the **key_length** parameter to determine the length of the key.

Use the **AESTOKEN** keyword for AES keys, or the **TOKEN** keyword for DES keys to indicate that the verb should determine the key type from the key token that you supply. For AES, all keys are type **AESDATA**. For DES, the key type is determined from the control vector in the key tokens. Alternatively, you can specify the key type using keywords shown in Table 43 on page 179 and Table 44 on page 179.

Key types can have mandatory key forms. For example, CVARENC keys must be generated in pairs with **CVARDEC** keys. The reason is that a **CVARENC** key can only be used for encryption, and without a **CVARDEC** key you cannot decrypt the data. See Table 43 on page 179 and Table 44 on page 179 for valid key type and key form combinations.

key_type_2

Direction: Input
Type: String

An 8-byte keyword from the following group:

AESDATA	DATA	ENCIPHER	MACVER
AESTOKEN	DATAC	EXPORTER	OKEYXLAT
CIPHER	DATAM	IKEYXLAT	OPINENC
CIPHERXI	DATAMV	IMPORTER	PINGEN
CIPHERXL	DATAXLAT	IPINENC	PINVER
CIPHERXO	DECIPHER	MAC	

or the keyword **TOKEN**.

For information on the meaning of the key types, see Table 4 on page 40.

Use the *key_type_2* parameter for a key pair, which is shown in Table 44 on page 179. The keyword must be left-aligned and padded with blanks. Valid type combinations depend on the key form.

If *key_type_2* is **TOKEN**, CCA examines the control vector (CV) field in the *generated_key_identifier_2* parameter to derive the key type. When *key_type_2* is **TOKEN**, CCA does not check for the length of the key for **DATA** keys. Instead, it uses the *key_length* parameter to determine the length of the key.

If you want only one key to be generated, specify the *key_type_2* and *KEK_key_identifier_2* as binary zeros.

See Table 43 on page 179 and Table 44 on page 179 for valid key type and key form combinations.

KEK_key_identifier_1

Direction: Input/Output
Type: String

A 64-byte string of an internal key token containing the importer or exporter key-encrypting key, or a key label. If you supply a key label that is less than 64-bytes, it must be left-aligned and padded with blanks. *KEK_key_identifier_1* is required for a *key_form* of **IM**, **EX**, **IMEX**, **EXEX**, or **IMIM**.

If the *key_form* is **OP**, **OPEX**, **OPIM**, or **OPOP**, the *KEK_key_identifier_1* is null.

If the NOCV bit is on in the internal key token containing the key-encrypting key, the key-encrypting key itself (not the key-encrypting key variant) is used to encipher the generated key.

Control vectors are explained in "Control vector" on page 36 and the NOCV bit is shown in Table 209 on page 772.

This parameter is not used when generating AES keys, and should point to null key-tokens.

KEK_key_identifier_2

Direction: Input/Output
Type: String

A 64-byte string of an internal key token containing the importer or exporter key-encrypting key, or a key label of an internal token. If you supply a key label that is less than 64-bytes, it must be left-aligned and padded with blanks. *KEK_key_identifier_2* is required for a *key_form* of **OPIM**, **OPEX**, **IMEX**, **IMIM**, or **EXEX**. This field is ignored for *key_form* keywords **OP**, **IM** and **EX**.

If the NOCV bit is on in the internal key token containing the key-encrypting key, the key-encrypting key itself (not the key-encrypting key variant) is used to encipher the generated key.

Control vectors are explained in "Control vector" on page 36 and the NOCV bit is shown in Table 209 on page 772.

This parameter is not used when generating AES keys, and should point to null key-tokens.

generated_key_identifier_1

Direction: Input/Output
Type: String

This parameter specifies either a generated:

- Internal key token for an operational key form, or
- External key token containing a key enciphered under the *kek_key_identifier_1* parameter.

When *key_type_1* parameter is **AESDATA**, the *generated_key_identifier_1* parameter is ignored. In this case, it is recommended that the parameter be initialized to 64-bytes of X'00'.

If you specify a *key_type_1* of **TOKEN**, then this field contains a valid token of the key type you want to generate. Otherwise, on input, this parameter must be binary zeros. See *key_type_1* for a list of valid key types.

If you specify a *key_type_1* of **IMPORTER** or **EXPORTER** and a *key_form* of **OPEX**, and if the *generated_key_identifier_1* parameter contains a valid internal token of the same type, the NOCV bit, if on, is propagated to the generated key token.

Using the **AESTOKEN** or **TOKEN** keyword in the key type parameters requires that the key tokens already exist when the verb is called, so the information in those tokens can be used to determine the key type:

- The *key_type_1* parameter overrides the type in the token.
- The *key_length* parameter overrides the length value in the generated key token.

In general, unless you are using the **AESTOKEN** or **TOKEN** keyword, you must identify a null key token in the generated key identifier parameters on input.

generated_key_identifier_2

Direction: Input/Output
Type: String

This parameter specifies a generated external key token containing a key enciphered under the *kek_key_identifier_2* parameter.

If you specify a *key_type_2* of **TOKEN**, then this field contains a valid token of the key type you want to generate. Otherwise, on input, this parameter must be binary zeros. See *key_type_1* for a list of valid key types.

The token can be an internal or external token.

Using the **AESTOKEN** or **TOKEN** keyword in the key type parameters requires that the key tokens already exist when the verb is called, so the information in those tokens can be used to determine the key type. In general, unless you are using the **AESTOKEN** or **TOKEN** keyword, you must identify a null key token in the generated key identifier parameters on input.

Restrictions

The restrictions for CSNBKGN.

None.

Required commands

The CSNBKGN required commands.

Depending on the **key_type** and **key_form** parameters selected, the verb could require one or more of these commands to be enabled in the active role:

Offset	Command
X'008C'	**Key Generate - Key set**
X'008E'	**Key Generate - OP**
X'00D7'	**Key Generate - Key set extended**
X'00DB'	**Key Generate - SINGLE-R**

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Note: A role with offset X'00DB' enabled can also use the Remote Key Export verb.

Usage notes

The usage notes for CSNBKGN.

Table 43 on page 179 shows the valid key type and key form combinations for a single key. Key types marked with an '*' must be requested through the specification of a proper control vector in a key token and through the use of the **TOKEN** keyword. See also Chapter 18, “Key forms and types used in the Key Generate verb,” on page 893.

Note: Not all key types are valid on all hardware. See Table 4 on page 40.

For AES keys, only key form **OP** is supported. AES keys cannot be generated in pairs.

Table 43. Keywords for Key Generate, valid key types and key forms for a single key

Key Type 1	Key Type 2	OP	IM	EX
AESDATA	Not applicable	X		
AESTOKEN	Not applicable	X		
DATA	Not applicable	X	X	X
DATAC*	Not applicable	X	X	X
DATAM	Not applicable	X	X	X
DKYGENKY*	Not applicable	X	X	X
KEYGENKY*	Not applicable	X	X	X
MAC	Not applicable	X	X	X
PINGEN	Not applicable	X	X	X

Table 44 shows the valid key type and key form combinations for a key pair.

Table 44. Keywords for Key Generate, valid key types and key forms for a key pair

Key Type 1	Key Type 2	OPEX	EXEX	OPIM, OPOP, IMIM	IMEX
CIPHER	**CIPHER**	X	X	X	X
CIPHER	**DECIPHER**	X	X	X	X
CIPHER	**ENCIPHER**	X	X	X	X
CVARDEC*	**CVARENC***	E			E
CVARDECC*	**CVARPINE***	E			E
CVARENC*	**CVARDEC***	E			E
CVARENC*	**CVARXCVL***	E			E
CVARENC*	**CVARXCVR***	E			E
CVARXCVL*	**CVARENC***	E			E
CVARXCVR*	**CVARENC***	E			E
CVARPINE*	**CVARDEC***	E			E
DATA	**DATA**	X	X	X	X
DATA	**DATAXLAT**	X	X		X
DATAC*	**DATAC***	X	X	X	X
DATAM	**DATAM**	X	X	X	X
DATAM	**DATAMV**	X	X	X	X
DATAXLAT	**DATAXLAT**	X	X		X
DECIPHER	**CIPHER**	X	X	X	X
DECIPHER	**ENCIPHER**	X	X	X	X
DKYGENKY*	**DKYGENKY***	X	X	X	X
ENCIPHER	**CIPHER**	X	X	X	X

Table 44. Keywords for Key Generate, valid key types and key forms for a key pair (continued)

Key Type 1	Key Type 2	OPEX	EXEX	OPIM, OPOP, IMIM	IMEX
ENCIPHER	**DECIPHER**	X	X	X	X
EXPORTER	**IKEYXLAT**	X	X		X
EXPORTER	**IMPORTER**	X	X		X
IKEYXLAT	**EXPORTER**	X	X		X
IKEYXLAT	**OKEYXLAT**	X	X		X
IMPORTER	**EXPORTER**	X	X		X
IMPORTER	**OKEYXLAT**	X	X		X
IPINENC	**OPINENC**	X	X	E	X
KEYGENKY*	**KEYGENKY***	X	X	X	X
MAC	**MAC**	X	X	X	X
MAC	**MACVER**	X	X	X	X
OKEYXLAT	**IKEYXLAT**	X	X		X
OKEYXLAT	**IMPORTER**	X	X		X
OPINENC	**IPINENC**	X	X	E	X
OPINENC	**OPINENC**			X	
PINVER	**PINGEN**	X	X		X
PINGEN	**PINVER**	X	X		X

Note:

1. AES keys cannot be generated in pairs.
2. An 'X' indicates a permissible key type combination for a given key form. An 'E' indicates that a special (Extended) command is required as those keys require special handling.
3. The key types marked with an '*' must be requested through the specification of a proper control-vector in a key token and the use of the **TOKEN** keyword.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKGNJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBKGNJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           byte[]            key_form,
           byte[]            key_length,
           byte[]            key_type_1,
           byte[]            key_type_2,
           byte[]            KEK_key_identifier_1,
           byte[]            KEK_key_identifier_2,
           byte[]            generated_key_identifier_1,
           byte[]            generated_key_identifier_2);
```

Key Generate2 (CSNBKGN2)

Use the Key Generate2 verb to generate either one or two keys of any type.

This verb does not produce keys in clear form and all keys are returned in encrypted form. When two keys are generated, each key has the same clear value, although this clear value is not exposed outside the secure cryptographic feature.

This verb returns variable-length CCA key tokens and uses the **AESKW** wrapping method.

This verb supports AES and HMAC keys. Operational keys will be encrypted under the AES master key.

Format

The format of CSNBKGN2.

```
CSNBKGN2(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             clear_key_bit_length,
             key_type_1,
             key_type_2,
             key_name_1_length,
             key_name_1,
             key_name_2_length,
             key_name_2,
             user_associated_data_1_length,
             user_associated_data_1,
             user_associated_data_2_length,
             user_associated_data_2,
             key_encrypting_key_identifier_1_length,
             key_encrypting_key_identifier_1,
             key_encrypting_key_identifier_2_length,
             key_encrypting_key_identifier_2,
             generated_key_identifier_1_length,
             generated_key_identifier_1,
             generated_key_identifier_2_length,
             generated_key_identifier_2)
```

Parameters

The parameter definitions for CSNBKGN2J.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

The number of keywords you supplied in the *rule_array* parameter. This value must be 2.

rule_array

Direction: Input
Type: String array

The *rule_array* contains keywords that provide control information to the verb. The keywords must be in contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. The *rule_array* keywords are described in Table 45.

Table 45. Keywords for Key Generate2 control information

Keyword	Description
Token algorithm (Required)	
AES	Specifies to generate an AES key token.
HMAC	Specifies to generate an HMAC key token.
Key form (One, required) The first two characters refer to *key_type_1*. The next two characters refer to *key_type_2*. See "Usage notes" on page 187 for details.	
EX	One key that can be exported to another system.
EXEX	A key pair. Both keys are to be exported, possibly to two different systems. Both keys have the same clear value.
IM	One key that can be locally imported. The key can be imported onto this system to make it operational at another time.
IMEX	A key pair to be imported. One key is to be imported locally, and one key to be exported. Both keys have the same clear value.
IMIM	A key pair to be imported. Both keys are to be imported locally at another time. Both keys have the same clear value.
OP	One operational key. The key is returned to the caller in operational form to be used locally.
OPEX	A key pair. One key that is operational, and one key to be exported. Both keys have the same clear value.
OPIM	A key pair. One key that is operational, and one key to be imported locally at another time. Both keys have the same clear value.
OPOP	A key pair. Either with the same key type with different associated data, or complementary key types. Both keys have the same clear value.
Payload Version for generated_key_identifier_1 (one, optional) **Note:** This keyword overrides the payload format version of any corresponding skeleton token.	
V0PYLDK1	Build a token with the old variable-length payload format for the **generated_key_identifier_1** parameter. This is the default for AES CIPHER, EXPORTER, and IMPORTER key types and is only valid with those key types.
V1PYLDK1	Build a token with the new fixed-length payload format for the **generated_key_identifier_1** parameter. This is the default for AES MAC, PINPROT, PINCALC, PINPRW, and DKYGENKY key types. Not valid with the HMAC MAC key type.
Payload Version for generated_key_identifier_2 (one, optional) **Note:** This keyword overrides the payload format version of any corresponding skeleton token.	
V0PYLDK2	Build a token with the old variable-length payload format for the **generated_key_identifier_2** parameter. This is the default for AES CIPHER, EXPORTER, and IMPORTER key types and is only valid with those key types.
V1PYLDK2	Build a token with the new fixed-length payload format for the **generated_key_identifier_2** parameter. This is the default for AES MAC, PINPROT, PINCALC, PINPRW, and DKYGENKY key types. Not valid with the HMAC MAC key type.

clear_key_bit_length

Direction: Input

Type: Integer

A pointer to an integer variable containing the number of clear-key bits to randomly generate and return encrypted in the generated key or keys. If a generated key token has a key type of **TOKEN**, this value overrides any key length contained in the key token. The value can be 128, 192, and 256 for AES keys, and 80 - 2048 for HMAC keys.

key_type_1

Direction: Input
Type: String

Use the *key_type_1* parameter for the first, or only, key that you want generated. The keyword must be left-aligned and padded with blanks. Valid type combinations depend on the key form, and are documented in Table 48 on page 187 and Table 49 on page 188.

The 8-byte keyword for the *key_type_1* parameter can be one of the following:

Table 46. Keywords and associated algorithms for key_type_1 parameter

Keyword	Algorithm
CIPHER	**AES**
EXPORTER	**AES**
IMPORTER	**AES**
MAC	**HMAC**
MACVER	**HMAC**
Specify the keyword **TOKEN** when supplying a key token in the *generated_key_identifier_1* parameter.	

If *key_type_1* is **TOKEN**, the associated data in the *generated_key_identifier_1* parameter is used to derive the key type.

key_type_2

Direction: Input
Type: String

Use the *key_type_2* parameter for a key pair, which is shown in Table 49 on page 188. The keyword must be left-aligned and padded with blanks. Valid type combinations depend on the key form.

The 8-byte keyword for the *key_type_2* parameter can be one of the following:

Table 47. Keywords and associated algorithms for key_type_2 parameter

Keyword	Algorithm
CIPHER	**AES**
EXPORTER	**AES**
IMPORTER	**AES**
MAC	**HMAC**
MACVER	**HMAC**
Specify the keyword **TOKEN** when supplying a key token in the *generated_key_identifier_2* parameter.	

If *key_type_2* is **TOKEN**, the associated data in the *generated_key_identifier_2* parameter is used to derive the key type.

When only one key is being generated, this parameter is ignored.

key_name_1_length

Direction: Input
Type: Integer

The length of the *key_name* parameter for *generated_key_identifier_1*. Valid values are 0 and 64.

key_name_1

Direction: Input
Type: String

A 64-byte key store label to be stored in the associated data structure of *generated_key_identifier_1*.

key_name_2_length

Direction: Input
Type: Integer

The length of the *key_name* parameter for *generated_key_identifier_2*. Valid values are 0 and 64. When only one key is being generated, this parameter is ignored.

key_name_2

Direction: Input
Type: String

A 64-byte key store label to be stored in the associated data structure of *generated_key_identifier_2*.

When only one key is being generated, this parameter is ignored.

user_associated_data_1_length

Direction: Input
Type: Integer

The length of the user-associated data parameter for *generated_key_identifier_1*. The valid values are 0 - 255 bytes.

user_associated_data_1

Direction: Input
Type: String

User-associated data to be stored in the associated data structure for *generated_key_identifier_1*.

user_associated_data_2_length

Direction: Input
Type: Integer

The length of the user-associated data parameter for *generated_key_identifier_2*. The valid values are 0 - 255 bytes. When only one key is being generated, this parameter is ignored.

user_associated_data_2

Direction: Input
Type: String

User associated data to be stored in the associated data structure for *generated_key_identifier_2*.

When only one key is being generated, this parameter is ignored.

key_encrypting_key_identifier_1_length

Direction: Input
Type: Integer

The length of the buffer for *key_encrypting_key_identifier_1* in bytes. When the key form rule is **OP**, **OPOP**, **OPIM**, or **OPEX**, this length must be zero. When the key form rule is **EX**, **EXEX**, **IM**, **IMEX**, or **IMIM**, the value must be between the actual length of the token and 725 bytes when *key_encrypting_key_identifier_1* is a token.

The value must be 64 bytes when *key_encrypting_key_identifier_1* is a label.

key_encrypting_key_identifier_1

Direction: Input
Type: String

When *key_encrypting_key_identifier_1_length* is zero, this parameter is ignored. Otherwise, *key_encrypting_key_identifier_1* contains an internal key token containing the AES importer or exporter key-encrypting key, or a key label.

If the token supplied was encrypted under the old master key, the token will be returned encrypted under the current master key.

key_encrypting_key_identifier_2_length

Direction: Input
Type: Integer

The length of the buffer for *key_encrypting_key_identifier_2* in bytes. When the key form rule is **OPOP**, this length must be zero. When the key form rule is **EXEX**, **IMEX**, **IMIM**, **OPIM**, or **OPEX**, the value must be between the actual length of the token and 725 when *key_encrypting_key_identifier_2* is a token. The value must be 64 when *key_encrypting_key_identifier_2* is a label.

When only one key is being generated, this parameter is ignored.

key_encrypting_key_identifier_2

Direction: Input/Output
Type: String

When *key_encrypting_key_identifier_2_length* is zero, this parameter is ignored. Otherwise, *key_encrypting_key_identifier_2* contains an internal key token containing the AES importer or exporter key-encrypting key, or a key label.

If the token supplied was encrypted under the old master key, the token will be returned encrypted under the current master key.

When only one key is being generated, this parameter is ignored.

generated_key_identifier_1_length

Direction: Input/Output
Type: Integer

On input, the length of the buffer for the *generated_key_identifier_1* parameter in bytes. The maximum value is 900 bytes.

On output, the parameter will hold the actual length of the *generated_key_identifier_1*.

generated_key_identifier_1

Direction: Input/Output
Type: String

The buffer for the first generated key token.

On input, if you specify a *key_type_1* of **TOKEN**, then the buffer contains a valid key token of the key type you want to generate. The key token must be left-aligned in the buffer. Otherwise, this parameter must be binary zeros. See *key_type_1* on page "key_type_1" on page 183 for a list of valid key types.

On output, the buffer contains the generated key token.

generated_key_identifier_2_length

Direction: Input/Output
Type: Integer

On input, the length of the buffer for the *generated_key_identifier_2* in bytes. The minimum value is 120 bytes and the maximum value is 725 bytes. The maximum value is 900 bytes.

On output, the parameter will hold the actual length of the *generated_key_identifier_2*.

When only one key is being generated, this parameter is ignored.

generated_key_identifier_2

Direction: Input/Output
Type: String

The buffer for the second generated key token.

On input, if you specify a *key_type_2* of **TOKEN**, then the buffer contains a valid key token of the key type you want to generate. The key token must be left-aligned in the buffer. Otherwise, this parameter must be binary zeros. See *key_type_2* on page "key_type_2" on page 183 for a list of valid key types.

On output, the buffer contains the generated key token.

When only one key is being generated, this parameter is ignored

Restrictions

The restrictions for CSNBKGN2.

This verb was introduced with CCA 4.1.0.

Required commands

The CSNBKGN2 required commands.

Depending on the *key_type* and *key_form* parameters selected, the verb could require one or more of these commands to be enabled in the active role:

Offset	Command
X'00EA'	**Key Generate2 - OP**
X'00EB'	**Key Generate2 - Key set**

To disallow the wrapping of a key with a weaker key-encrypting key, enable the **Prohibit weak wrapping - Transport keys** command (offset X'0328') in the active role. This command affects multiple verbs. See Chapter 22, "Access control points and verbs," on page 953.

To receive a warning when wrapping a key with a weaker key-encrypting key, enable the **Warn when weak wrap - Transport keys** command (offset X'032C') in the active role. The **Prohibit weak wrapping - Transport keys** command (offset X'0328') overrides this command.

To disable the wrapping of a key with a weaker master key, the **Prohibit weak wrapping - Master keys** command (offset X'0333') must be enabled in the active role.

To receive a warning when wrapping a key with a weaker master key, enable the **Warn when weak wrap - Master keys** command (offset X'0332') in the active role. The **Prohibit weak wrapping - Master keys** command overrides this command.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

Usage notes for CSNBKGN2.

The key forms are defined as follows:

Operational (OP)
The key value is enciphered under a master key. The result is placed into an internal key token. The key is then operational at the local system.

Importable (IM)
The key value is enciphered under an importer key-encrypting key. The result is placed into an external key token. The corresponding *key_encrypting_key_identifier_n* parameter must contain an AES **IMPORTER** key token or label.

Exportable (EX)
The key value is enciphered under an exporter key-encrypting key. The result is placed into an external key token. The corresponding *key_encrypting_key_identifier_n* parameter must contain an AES **EXPORTER** key token or label.

These tables list the valid key type and key form combinations.

Table 48. Key Generate2 valid key type and key form for one key

key_type_1	Key Form OP, IM, EX
CIPHER	X
MAC	X

Table 49. Key Generate2 valid key type and key forms for two keys

key_type_1	key_type_2	Key form OPOP, OPIM, or IMIM	Key Form OPEX, EXEX, or IMEX
CIPHER	**CIPHER**	X	X
MAC	**MAC**	X	X
MAC	**MACVER**	X	X
MACVER	**MAC**	X	X
EXPORTER	**IMPORTER**		X
IMPORTER	**EXPORTER**		X

For AES keys, the AES KEK must be at least as strong as the key being generated to be considered sufficient strength.

For HMAC keys, the AES KEK must be sufficient strength as described in the following table:

Table 50. AES KEK strength required for generating an HMAC key under an AES KEK

Key-usage field 2 in the HMAC key contains	Minimum strength of AES KEK to adequately protect the HMAC key
SHA-256, **SHA-384**, or **SHA-512**	256 bits
SHA-224	192 bits
SHA-1	128 bits

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKGN2J.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBKGN2J(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            clear_key_bit_length,
           byte[]            key_type_1,
           byte[]            key_type_2,
           byte[]            key_name_1_length,
           byte[]            key_name_1,
           byte[]            key_name_2_length,
           byte[]            key_name_2,
           byte[]            user_associated_data_1_length,
           byte[]            user_associated_data_1,
           byte[]            user_associated_data_2_length,
           byte[]            user_associated_data_2,
           byte[]            key_encrypting_key_identifier_1_length,
           byte[]            key_encrypting_key_identifier_1,
           byte[]            key_encrypting_key_identifier_2_length,
           byte[]            key_encrypting_key_identifier_2,
           byte[]            generated_key_identifier_1_length,
           byte[]            generated_key_identifier_1,
           byte[]            generated_key_identifier_2_length,
           byte[]            generated_key_identifier_2);
```

Key Import (CSNBKIM)

Use the Key Import verb to re-encipher a key from encryption under an importer key-encrypting key to encryption under the master key.

The re-enciphered key is in operational form.

Choose one of the following options:

- Specify the *key_type* parameter as **TOKEN** and specify the external key token in the *source_key_identifier* parameter. The key type information is determined from the control vector in the external key token.
- Specify a key type in the *key_type* parameter and specify an external key token in the *source_key_identifier* parameter. The specified key type must be compatible with the control vector in the external key token.
- Specify a valid key type in the *key_type* parameter and a null key token in the *source_key_identifier* parameter. The default control vector for the *key_type* specified will be used to process the key.

For **DATA** keys, this verb generates a key of the same length as that contained in the input token.

Format

The format of CSNBKIM.

```
CSNBKIM(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             key_type,
             source_key_identifier,
             importer_key_identifier,
             target_key_identifier)
```

Parameters

The parameter definitions for CSNBKIM.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

key_type

Direction: Input
Type: String

The type of key you want to re-encipher under the master key. Specify an 8-byte keyword or the keyword TOKEN. The keyword must be left-aligned and padded on the right with blanks.

If the key type is **TOKEN**, CCA determines the key type from the control vector (CV) field in the external key token provided in the *source_key_identifier* parameter.

The key type of **TOKEN** is not allowed when the *importer_key_identifier* parameter is NOCV.

Key type values for the Key Import verb are:

CIPHER	DATAC	EXPORTER	MACVER	PINVER
CIPHERXL	DATAM	IMPORTER	MACD	TOKEN
CIPHERXI	DATAMV	IKEYXLAT	OKEYXLAT	
CIPHERXO	DECIPHER	IPINENC	OPINENC	
DATA	ENCIPHER	MAC	PINGEN	

For information on the meaning of the key types, see Table 4 on page 40.

We recommend using key type of **TOKEN** when importing double-length **MAC** and **MACVER** keys.

source_key_identifier

Direction: Input
Type: String

The key you want to re-encipher under the master key. The parameter is a 64-byte field for the enciphered key to be imported containing either an external key token or a null key token. If you specify a null token, the token is all binary zeros, except for a key in bytes 16-23 or 16-31, or in bytes 16-31 and 48-55 for triple-length **DATA** keys. Refer to Table 212 on page 775.

If key type is **TOKEN**, this field might not specify a null token.

This verb supports the no-export function in the CV.

importer_key_identifier

Direction: Input/Output
Type: String

The importer key-encrypting key that the key is currently encrypted under. The parameter is a 64-byte area containing either the key label of the key in the cryptographic key data set or the internal key token for the key. If you supply a key label that is less than 64-bytes, it must be left-aligned and padded with blanks.

Note: If you specify a NOCV importer in the *importer_key_identifier* parameter, the key to be imported must be enciphered under the importer key itself.

target_key_identifier

Direction: Input/Output
Type: String

This parameter is the generated re-enciphered key. The parameter is a 64-byte area that receives the internal key token for the imported key.

If the imported key type is **IMPORTER** or **EXPORTER** and the token key type is the same, the *target_key_identifier* parameter changes direction to both input and output. If the application passes a valid internal key token for an **IMPORTER** or **EXPORTER** key in this parameter, the NOCV bit is propagated to the imported key token.

Restrictions

The restrictions for CSNBKIM.

For security reasons, requests will fail by default if they use an equal key halves importer to import a key with unequal key halves. You must have access control point 'Key Import - Unrestricted' explicitly enabled if you want to import keys in this manner.

Required commands

The required commands for CSNBKIM.

This verb requires the **Key Import** command (offset X'0012') to be enabled in the active role.

By also enabling the **Key Import - Unrestricted** command (offset X'027B'), you can permit a less secure mode of operation that enables an equal key-halves **IMPORTER** key-encrypting key to import a key having unequal key-halves (key parity bits are ignored).

To disable the wrapping of a key with a weaker master key, the **Prohibit weak wrapping - Master keys** command (offset X'0333') must be enabled in the active role.

To receive a warning when wrapping a key with a weaker master key, enable the **Warn when weak wrap - Master keys** command (offset X'0332') in the active role. The **Prohibit weak wrapping - Master keys** command overrides this command.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

Usage notes for CSNBKIM.

Use of NOCV keys are controlled by an access control point in the CEX*C. Creation of NOCV key-encrypting keys is available only for standard IMPORTERs and EXPORTERs.

This verb will mark an imported KEK as a NOCV-KEK KEK:

- If a token is supplied in the target token field, it must be a valid importer or exporter token. If the token fails token validation, processing continues, but the NOCV flag will not be copied
- The source token (key to be imported) must be a importer or exporter with the default control vector.
- If the target token is valid and the NOCV flag is on and the source token is valid and the control vector of the target token is exactly the same as the source token, the imported token will have the NOCV flag set on.
- If the target token is valid and the NOCV flag is on and the source token is valid and the control vector of the target token is NOT exactly the same as the source token, a return code will be given.
- All other scenarios will complete successfully, but the NOCV flag will not be copied

The software bit used to mark the imported token with export prohibited is not supported on a CEX*C. The internal token for an export prohibited key will have the appropriate control vector that prohibits export.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKIMJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBKIMJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           byte[]            key_type,
           byte[]            source_key_token,
           byte[]            importer_key_identifier,
           byte[]            target_key_identifier);
```

Key Part Import (CSNBKPI)

Use the Key Part Import verb to combine, by XORing, the clear key parts of any key type and return the combined key value either in an internal token or as an update to the key storage file.

Before you use the Key Part Import verb for the first key part, you must use the Key Token Build or Key Token Build2 verb to create the internal key token into which the key will be imported. Subsequent key parts are combined with the first part in internal token form or as a label from the key storage file.

The preferred way to specify key parts is **FIRST**, **ADD-PART**, and **COMPLETE** in the *rule_array*. Only when the combined key parts have been marked as complete can the key token be used in any cryptographic operation. The partial key can be passed to the Key Token Change or Key Token Change2 verb for re-encipherment, in case building the key was started during a master key change operation. The partial key can be passed to the Key Token Parse verb, in order to discover how the key token was originally specified, if researching an old partial key. Partial keys can also be passed to the Key Test, Key Test2, and Key Test Extended verbs.

Key parts can also be specified as **FIRST**, **MIDDLE**, or **LAST** in the *rule_array*. **ADD-PART** or **MIDDLE** can be executed multiple times for as many key parts as necessary. Only when the **LAST** part has been combined can the key token be used in any other service.

New applications should employ the **ADD-PART** and **COMPLETE** keywords in lieu of the **MIDDLE** and **LAST** keywords in order to ensure a separation of responsibilities between someone who can add key-part information and someone who can declare that appropriate information has been accumulated in a key.

The Key Part Import verb can also be used to import a key without using key parts. Call the Key Part Import verb FIRST with key part value X'0000...' then call the Key Part Import verb **LAST** with the complete value.

Keys created using this service have odd parity. The **FIRST** key part is adjusted to odd parity. All subsequent key parts are adjusted to even parity before being combined.

Format

The format of CSNBKPI.

```
CSNBKPI(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_part,
             key_identifier)
```

Parameters

The parameter definitions for CSNBKPI.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1 or 2.

rule_array

Direction: Input
Type: String array

The keyword that provides control information to the verb. The keywords must be eight bytes of contiguous storage with the keyword left-aligned in its 8-byte location and padded on the right with blanks. The *rule_array* keywords are described in Table 51.

Table 51. Keywords for Key Part Import control information

Keyword	Description
Key part (One, required)	
FIRST	This keyword specifies that an initial key part is being entered. This verb returns this key-part encrypted by the master key in the key token that you supplied.
ADD-PART	This keyword specifies that additional key-part information is provided.
COMPLETE	This keyword specifies that the key-part bit shall be turned off in the control vector of the key rendering the key fully operational. Note that no key-part information is added to the key with this keyword.
MIDDLE	This keyword specifies that an intermediate key part, which is neither the first key part nor the last key part, is being entered. Note that the command control point for this keyword is the same as that for the **LAST** keyword and different from that for the **ADD-PART** keyword.
LAST	This keyword specifies that the last key part is being entered. The key-part bit is turned off in the control vector.

Table 51. Keywords for Key Part Import control information (continued)

Keyword	Description
RETRKPR	A key label must be passed as the *key_identifier*. This key label corresponds to a key stored in a KPIT register inside the crypto-card (not in host key storage). The key in that register has been loaded by label and key part using the KPIT verb by the TKE. This keyword for KPI allows the user to tell the card to wrap that key (it must be in the complete state) using the master key, place it in an internal token, and return that token to the user. This keyword applies only when using IBM z Systems .
Key-wrapping method (One, optional)	
USECONFG	This is the default. Specifies to wrap the key using the configuration setting for the default wrapping method. The default wrapping method configuration setting may be changed using the TKE. This keyword is ignored for AES keys.
WRAP-ENH	Specifies to wrap the key using the legacy wrapping method. This keyword is ignored for AES keys. This keyword was introduced with CCA 4.1.0.
WRAP-ECB	Specifies to wrap the key using the enhanced wrapping method. Valid only for DES keys. This keyword was introduced with CCA 4.1.0.

key_part

Direction: Input
Type: String

A 16-byte field containing the clear key part to be entered. If the key is a single-length key, the key part must be left-aligned and padded on the right with zeros. This field is ignored if **COMPLETE** is specified.

key_identifier

Direction: Input/Output
Type: String

A 64-byte field containing an internal token or a label of an existing key in the key storage file. If *rule_array* is **FIRST**, this field is the skeleton of an internal token of a single- or double-length key with the **KEY-PART** marking. If *rule_array* is **MIDDLE** or **LAST**, this is an internal token or key label of a partially combined key. Depending on the input format, the accumulated partial or complete key is returned as an internal token or as an updated key storage file record. The returned *key_identifier* will be encrypted under the current master key.

Restrictions

The restrictions for CSNBKPI.

If a label is specified on *key_identifier*, the label must be unique. If more than one record is found, the verb fails.

You must have access control point 'Key Part Import - Unrestricted' explicitly enabled. Otherwise, current applications will fail with either of the following conditions:

- The first eight bytes of key identifier is different than the second eight bytes AND the first eight bytes of the combined key are the same as the last second eight bytes
- The first eight bytes of key identifier is the same as the second eight bytes AND the first eight bytes of the combined key are different than the second eight bytes.

Required commands

The required commands for CSNBKPI.

This verb requires the following commands to be enabled in the active role:

Rule-array keyword	Offset	Command
FIRST	X'001B'	**Key Part Import - first key part**
ADD-PART	X'0278'	**Key Part Import - ADD-PART**
COMPLETE	X'0279'	**Key Part Import - COMPLETE**
MIDDLE or **LAST**	X'001C'	**Key Part Import - middle and last**
MIDDLE or **LAST**	X'001C'	**Key Part Import - middle and last**
WRAP-ECB or **WRAP-ENH** used, and default key-wrapping method setting does not match keyword	X'0140'	**Key Part Import - Allow wrapping override keywords**

The Key Part Import verb enforces the key-halves restriction when the **Key Part Import - Unrestricted** command (offset X'027A') is disabled in the active role. Enabling this command results in less secure operation and is not recommended.

To disable the wrapping of a key with a weaker master key, the **Prohibit weak wrapping - Master keys** command (offset X'0333') must be enabled in the active role.

To receive a warning when wrapping a key with a weaker master key, enable the **Warn when weak wrap - Master keys** command (offset X'0332') in the active role. The **Prohibit weak wrapping - Master keys** command overrides this command.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBKPI.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKPIJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBKPIJ(
          hikmNativeInteger return_code,
          hikmNativeInteger reason_code,
          hikmNativeInteger exit_data_length,
          byte[]            exit_data,
          hikmNativeInteger rule_array_count,
          byte[]            rule_array,
          byte[]            key_part,
          byte[]            key_identifier);
```

Key Part Import2 (CSNBKPI2)

Use the Key Part Import2 verb to combine, by XORing, the clear key parts of any key type and return the combined key value either in a variable-length internal key token or as an update to the key storage file.

Before you use the Key Part Import2 verb for the first key part, you must use the Key Token Build2 verb to create the variable-length internal key token into which the key will be imported. Subsequent key parts are combined with the first part in variable-length internal key token form, or as a label from the key storage file.

The preferred way to specify key parts is **FIRST**, **ADD-PART**, and **COMPLETE** in the *rule_array*. Only when the combined key parts have been marked as complete can the key token be used in any cryptographic operation. The partial key can be passed to the Key Token Change2 verb for re-encipherment, in case building the key was started during a master key change operation. The partial key can be passed to the Key Token Parse verb, in order to discover how the key token was originally specified, if researching an old partial key. Partial keys can also be passed to the Key Test, Key Test2, and Key Test Extended verbs.

Key parts can also be specified as **FIRST**, **MIDDLE**, or **LAST** in the *rule_array*. **ADD-PART** or **MIDDLE** can be executed multiple times for as many key parts as necessary. Only when the **LAST** part has been combined can the key token be used by any other verb.

New applications should employ the **ADD-PART** and **COMPLETE** keywords in lieu of the **MIDDLE** and **LAST** keywords in order to ensure a separation of responsibilities between someone who can add key-part information and someone who can declare that appropriate information has been accumulated in a key.

On each call to Key Part Import2 (except with the **COMPLETE** keyword), specify the number of bits to use for the clear key part. Place the clear key part in the *key_part* parameter, and specify the number of bits using the *key_part_length* variable. Any extraneous bits of *key_part* data will be ignored.

Consider using the Key Test2 verb to ensure a correct key value has been accumulated prior to using the **COMPLETE** option to mark the key as fully operational.

Format

The format of CSNBKPI2.

```
CSNBKPI2(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_part_bit_length,
             key_part,
             key_identifier_length,
             key_identifier)
```

Parameters

The parameters for CSNBKPI2.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

The number of keywords you supplied in the *rule_array* parameter. This value must be 2 or 3.

rule_array

Direction: Input
Type: String array

The *rule_array* contains keywords that provide control information to the verb. The keywords must be in contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. The *rule_array* keywords are described in Table 52.

Table 52. Keywords for Key Part Import2 control information

Keyword	Description
Token algorithm (Required)	
HMAC	Specifies to import an HMAC key token.
AES	Specifies to import an AES key token.
Key part (One required)	
FIRST	This keyword specifies that an initial key part is being entered. This verb returns this key-part encrypted by the master key in the key token that you supplied.
ADD-PART	This keyword specifies that additional key-part information is provided.
COMPLETE	This keyword specifies that the key-part bit shall be turned off in the control vector of the key rendering the key fully operational. Note that no key-part information is added to the key with this keyword.
Split knowledge (Optional, required when keyword **FIRST** is used)	
MIN3PART	Specifies that the key must be entered in at least three parts.
MIN2PART	Specifies that the key must be entered in at least two parts.
MIN1PART	Specifies that the key must be entered in at least one part.

key_part_bit_length

Direction: Input
Type: Integer

The length of the clear key in bits. This indicates the bit length of the key supplied in the *key_part* field. For **FIRST** and **ADD-PART** keywords, valid values are 80 - 2048 for HMAC keys, or 128, 192, or 256 for AES keys. The value must be 0 for the **COMPLETE** keyword.

key_part

Direction: Input
Type: String

This parameter is the clear key value to be applied. The key part must be left-aligned. This parameter is ignored if **COMPLETE** is specified.

key_identifier_length

Direction: Input/Output
Type: Integer

On input, the length of the buffer for the *key_identifier* parameter. For labels, the value is 64. The *key_identifier* must be left-aligned in the buffer. The buffer must be large enough to receive the updated token. The maximum value is 725. The output token will be longer when the first key part is imported.

On output, the actual length of the token returned to the caller. For labels, the value will be 64.

key_identifier

Direction: Input/Output
Type: String

The parameter containing an internal token or a 64-byte label of an existing key storage file record. If *rule_array* is **FIRST**, the key is a skeleton token. If *rule_array* is **ADD-PART**, this is an internal token or the label of a key storage file record of a partially combined key. Depending on the input format, the accumulated partial or complete key is returned as an internal token or as an updated record in a key storage file. The returned *key_identifier* will be encrypted under the current master key.

Restrictions

The restrictions for CSNBKPI2.

This verb was introduced with CCA 4.1.0.

Required commands

The required commands for CSNBKPI2.

This verb requires the following commands to be enabled in the active role:

Rule-array keyword	Offset	Command
FIRST and **MIN3PART**	X'0297'	**Key Part Import2 - Load first key part, require 3 key parts**
FIRST and **MIN2PART**	X'0298'	**Key Part Import2 - Load first key part, require 2 key parts**
FIRST and **MIN1PART**	X'0299'	**Key Part Import2 - Load first key part, require 1 key parts**
ADD-PART	X'029A'	**Key Part Import2 - Add second of 3 or more key parts**
	X'029B'	**Key Part Import2 - Add last required key part**
	X'029C'	**Key Part Import2 - Add optional key part**
COMPLETE	X'029D'	**Key Part Import2 - Complete key**

To disable the wrapping of a key with a weaker master key, the **Prohibit weak wrapping - Master keys** command (offset X'0333') must be enabled in the active role.

To receive a warning when wrapping a key with a weaker master key, enable the **Warn when weak wrap - Master keys** command (offset X'0332') in the active role. The **Prohibit weak wrapping - Master keys** command overrides this command.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBKPI2.

On each call to Key Part Import2, also specify a rule-array keyword to define the service action: **FIRST**, **ADD-PART**, or **COMPLETE**.

- With the **FIRST** keyword, the input key-token must be a skeleton token (no key material). Use of the **FIRST** keyword requires that the Load First Key Part2 access control point be enabled in the default role.
- With the **ADD-PART** keyword, the service XORs the clear key-part with the key value in the input key-token. Use of the **ADD-PART** keyword requires that an Add Key Part2 access control point be enabled in the default role. The key remains incomplete in the updated key token returned from the service.
- With the **COMPLETE** keyword, the KEY-PART bit is set off in the updated key token that is returned from the service. Use of the **COMPLETE** keyword requires that the Complete Key Part2 access control point be enabled in the default role. The *key_part_bit_length* parameter must be set to zero.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKPI2J.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBKPI2J(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger key_part_bit_length,
           hikmNativeInteger key_part,
           hikmNativeInteger key_identifier_length,
           hikmNativeInteger key_identifier);
```

Key Test (CSNBKYT)

Use the Key Test verb to generate or verify the value of either a master key, an internal AES key or key-part, or an internal DES key or key-part.

A key to test can be in the clear or encrypted under the master key. Keywords in the *rule_array* parameter specify whether the verb generates or verifies a verification pattern.

This algorithm is supported for clear and encrypted single and double length keys. Single, double and triple length keys are also supported with the **ENC-ZERO** algorithm. Clear triple length keys are not supported. See “Cryptographic key-verification techniques” on page 927.

With the default method, the verb generates a verification pattern and it creates and cryptographically processes a random number. This verb returns the random number with the verification pattern.

For historical reasons, the verification information is passed in two 8-byte variables pointed to by the *value_1* and *value_2* parameters. The **GENERATE** option uses these variables for output, and the **VERIFY** option uses these variables as input. For **VERIFY**, the verb returns a warning of return code 4, reason code 1 if the information provided in these variables does not match the calculated values.

Table 54 describes the use of the *value_1* and *value_2* variables for each of the available verification-process rule keywords.

This document uses new names for two of the parameters. The former names were misleading because they no longer reflected the use of these parameters. The header file, `csulincl.h`, continues to use the former names. See Table 53.

Table 53. Key Test parameter changes

Current name (used in this document)	**Former name (used in header file)**
value_1	*random_number*
value_2	*verification_pattern*

Table 54. Key Test GENERATE outputs and VERIFY inputs

Verification-process rule	**GENERATE outputs and VERIFY inputs**	
	value_1 **variable**	*value_2* **variable**
ENC-ZERO	Unused	Contains the 4-byte KVP in the high-order 4 bytes of the variable, taken from the high-order 4 bytes of the encrypted result. The low-order 4 bytes of the variable are unspecified.
MDC-4	Contains the 8-byte KVP taken from the high-order 8 bytes of the MDC-4 hash value.	Contains the low-order 8 bytes of the MDC-4 hash value.
SHA-1	Contains the 8-byte KVP taken from the high-order 8 bytes of the SHA-1 hash value.	Contains the low-order 8 bytes of the SHA-1 hash value.
SHA-256	Unused	Contains the 8-byte KVP taken from the high-order 8 bytes of the SHA-256 hash value.
No keyword, and first and third parts of the master key have different values	Same as **SHA-1**	Same as **SHA-1**
No keyword, and first and third parts of the master key have the same value	Contains the 8-byte KVP taken from the result of the z/OS-based master-key verification method.	Unused

Format

The format of CSNBKYT.

```
CSNBKYT(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_identifier,
             value_1,
             value_2)
```

This document uses new names for two of the parameters. The former names were misleading because they no longer reflected the use of these parameters. The header file, `csulincl.h`, continues to use the former names. See Table 53 on page 200.

Parameters

The parameters for CSNBKYT.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 2, 3, 4, or 5.

`rule_array`

Direction: Input
Type: String array

Two to five keywords provide control information to the verb. The keywords must be in contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. The *rule_array* keywords are described in Table 55.

Table 55. Keywords for Key Test control information

Keyword	Description
Key rule (One, required)	
KEY-CLR	Specifies the key supplied in *key_identifier* is a single-length clear key.
KEY-CLRD	Specifies the key supplied in *key_identifier* is a double-length clear key.
KEY-ENC	Specifies the key supplied in *key_identifier* is a single-length encrypted key.
KEY-ENCD	Specifies the key supplied in *key_identifier* is a double-length encrypted key.
KEY-KM	Specifies that the target is the master key register.
KEY-NKM	Specifies that the target is the new master-key register.
KEY-OKM	Specifies that the target is the old master-key register.
CLR-A128	Process a 128-bit AES clear-key or clear-key part.

Table 55. Keywords for Key Test control information (continued)

Keyword	Description
CLR-A192	Process a 192-bit AES clear-key or clear-key part.
CLR-A256	Process a 256-bit AES clear-key or clear-key part.
TOKEN	Process an AES clear or encrypted key contained in an AES key-token.
Master-key selector (One, optional). Use only with **KEY-KM**, **KEY-NKM**, or **KEY-OKM** keywords.	
AES-MK	Process one of the AES master-key registers.
APKA-MK	Process one of the APKA master-key registers. This keyword was introduced with CCA 4.1.0.
ASYM-MK	Specifies use of only the asymmetric master-key registers.
SYM-MK	Specifies use of only the symmetric master-key registers.
Process rule (One, required)	
GENERATE	Generate a verification pattern for the key supplied in *key_identifier*.
VERIFY	Verify a verification pattern for the key supplied in *key_identifier*.
Parity adjustment (One, optional)	
ADJUST	Adjust the parity of test key to odd before generating or verifying the verification pattern. The *key_identifier* field itself is not adjusted.
NOADJUST	Do not adjust the parity of test key to odd before generating or verifying the verification pattern. This is the default.
Verification process rule (One, optional). See “Cryptographic key-verification techniques” on page 927.	
ENC-ZERO	Specifies use of the "encrypted zeros" method. Use only with **KEY-CLR**, **KEY-CLRD**, **KEY-ENC**, or **KEY-ENCD** keywords.
MDC-4	Specifies use of the MDC-4 master key verification method. Use only with the **KEY-KM**, **KEY-NKM**, or **KEY-OKM** keywords. You must specify one master-key selector keyword to use this keyword.
SHA-1	Specifies use of the SHA-1 master-key-verification method. Use only with **KEY-KM**, **KEY-NKM**, or **KEY-OKM** keywords. You must specify one master-key selector keyword to use this keyword.
SHA-256	Specifies use of the SHA-256 master-key-verification method.
No keyword, and first and third parts of the master key have different values.	Defaults to the use of the SHA-1 master-key verification method when the **ASYM-MK** or **SYM-MK** master-key selector keyword is specified.
No keyword, and first and third parts of the master key have the same value.	Defaults to the use of the IBM z/OS-based master-key verification method when the **ASYM-MK** or **SYM-MK** master-key selector keyword is specified.

key_identifier

Direction: Input/Output
Type: String

The key for which to generate or verify the verification pattern. The parameter is a 64-byte string of an internal token, key label, or a clear key value left-aligned.

Note: If you supply a key label for this parameter, it must be unique in the key storage file.

value_1

Direction: Input/Output
Type: String

A pointer to a string variable. See Table 54 on page 200 for how this variable is used. For process rule **GENERATE** this parameter is output only, and for process rule **VERIFY** it is input only. This variable must be specified, even if it is not used. With the **ENC-ZERO** method, this parameter is not used.

value_2

Direction: Input/Output
Type: String

A pointer to a string variable. See Table 54 on page 200 for how this variable is used. For process rule **GENERATE** this parameter is output only, and for process rule **VERIFY** it is input only. This variable must be specified, even if it is not used. With the **ENC-ZERO** method, the high-order four bytes contain the verification data. For more detail, see "Cryptographic key-verification techniques" on page 927.

Restrictions

The restrictions for CSNBKYT.

None.

Required commands

The required commands for CSNBKYT.

In order to access key storage, this verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBKYT.

You can generate the verification pattern for a key when you generate the key. You can distribute the pattern with the key and it can be verified at the receiving node. In this way, users can ensure using the same key at the sending and receiving locations. You can generate and verify keys of any combination of key forms, that is, clear, operational or external.

The parity of the key is not tested.

For triple-length keys, use **KEY-ENC** or **KEY-ENCD** with **ENC-ZERO**. Clear triple-length keys are not supported.

In the Transaction Security System, **KEY-ENC** or **KEY-ENCD** both support enciphered single-length and double-length keys. They use the key-form bits in byte 5 of CV to determine the length of the key. To be consistent, in this implementation of CCA, both **KEY-ENC** and **KEY-ENCD** handle single- and double-length keys. Both products effectively ignore the keywords, which are supplied only for compatibility reasons.

This document uses new names for two of the parameters. The former names were misleading because they no longer reflected the use of these parameters. The header file, csulincl.h, continues to use the former names. See Table 53 on page 200.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKYTJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBKYTJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  hikmNativeInteger rule_array_count,
  byte[]            rule_array,
  byte[]            key_identifier,
  byte[]            value_1,
  byte[]            value_2);
```

Key Test2 (CSNBKYT2)

Use the Key Test2 verb to generate or verify a secure, cryptographic verification pattern for keys contained in a variable-length symmetric key-token.

A key to test can be in the clear or encrypted under the master key. In addition, the verb permits you to test the CCA master keys. Keywords in the *rule_array* parameter specify whether the verb generates or verifies a verification pattern. See "Cryptographic key-verification techniques" on page 927.

When the verb tests a verification pattern against a key, you must supply the verification pattern from a previous call to Key Test2. This verb returns the verification result in the return code and reason code.

Format

The format of CSNBKYT2.

```
CSNBKYT2(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_identifier_length,
             key_identifier,
             key_encrypting_key_identifier_length,
             key_encrypting_key_identifier,
             reserved_length,
             reserved,
             verification_pattern_length,
             verification_pattern)
```

Parameters

The parameters for CSNBKYT2.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

The number of keywords you supplied in the *rule_array* parameter. This value must be 2, 3, 4, or 5.

rule_array

Direction: Input
Type: String array

The *rule_array* contains keywords that provide control information to the verb. The keywords must be in contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. The *rule_array* keywords are described in Table 56.

Table 56. Keywords for Key Test2 control information

Keyword	Description
Token algorithm (Required)	
AES	Specifies that the key token is an AES key token.
DES	Specifies that the key token is a DES token. CCA internal, CCA external, and TR-31 token types are supported. Clear keys are not supported for this rule.
HMAC	Specifies that the key token is an HMAC key token.
Process rule (One required)	
GENERATE	Generate a verification pattern and an associated random number for the input key or key part for the specified key.
VERIFY	Verify that a verification pattern matches the specified key.
Verification pattern calculation algorithm (One optional)	
ENC-ZERO	Verification pattern for AES and DES keys calculated by encrypting a data block filled with X'00' bytes. This is the default and only method available for **DES**. Not valid with **HMAC**. This method is only available for **AES** if the Key Test2 - AES, ENC-ZERO (offset X'0021)' access control point is enabled.
SHA-256	Verification pattern will be calculated for an AES token using the same method as the Key Test verb, with the SHA-256 rule. This rule can be used to verify that the same key value is present in a version X'04' DATA token and version X'05' AES CIPHER token or to verify that the same key value is present in a version X'05' AES IMPORTER/EXPORTER pair.
SHA2VP1	Specifies to use the SHA-256 based verification pattern calculation algorithm. Valid only with **HMAC**. This is the default for **HMAC**. For more information, see "SHAVP1 algorithm" on page 930.
Token type rule (Required if TR-31 token passed and token algorithm DES is specified. Not valid otherwise.)	
TR-31	Specifies that *key_identifier* contains a TR-31 key block.

Table 56. Keywords for Key Test2 control information (continued)

Keyword	Description
AESKWCV	Specifies that the **key_identifier** contains an external variable length symmetric key token whose type is DESUSECV. The IKEK-AES keyword must be specified for the KEK identifier rule.
KEK identifier rules (Optional - see defaults)	
IKEK-AES	The wrapping KEK for the key to test is an AES KEK. This is the default for **AES** and **HMAC** token algorithms, and is not allowed with **DES**.
IKEK-DES	The wrapping KEK for the key to test is a DES KEK. This is the default for **DES** token algorithm, and is only allowed with **DES** token algorithm.
IKEK-PKA	The wrapping KEK for the key to test is an RSA or (other key stored in PKA key storage.) This is not the default for any token algorithm, and must be specified if an RSA KEK is used. This rule is not allowed with **DES** token algorithm.

key_identifier_length

Direction: Input
Type: Integer

The length of the *key_identifier* in bytes. The maximum value is 9992.

key_identifier

Direction: Input
Type: String

A pointer to the key for which to generate or verify the verification pattern. The parameter is a variable length string of an internal token or the 64-byte label of a key in key storage. This token may be a DES internal or external token, AES internal version X'04' token, internal or external variable-length symmetric token, or a TR-31 key block. Clear DES tokens are not supported. If an internal token was supplied and was encrypted under the old master key, the token will be returned encrypted under the current master key.

key_encrypting_key_identifier_length

Direction: Input
Type: Integer

The byte length of the *key_encrypting_key_identifier* parameter. When *key_identifier* is an internal token, the value must be zero.

If *key_encrypting_key_identifier* is a label for a record in key storage, the value must be 64. If the *key_encrypting_key_identifier* is an AES KEK, the value must be between the actual length of the token and 725. If the *key_encrypting_key_identifier* is a DES KEK, the value must be 64. If *key_encrypting_key_identifier* is an RSA KEK, the maximum length is 3500.

key_encrypting_key_identifier

Direction: Input/Output
Type: String

When *key_encrypting_key_identifier_length* is non-zero, the *key_encrypting_key_identifier* contains an internal key token containing the key-encrypting key, or a key label. If the key identifier supplied was an AES or

DES token encrypted under the old master key, the token will be returned encrypted under the current master key.

reserved_length

Direction: Input
Type: Integer

The byte length of the *reserved* parameter. This value must be 0.

reserved

Direction: Input/Output
Type: String

This parameter is ignored.

verification_pattern_length

Direction: Input/Output
Type: Integer

The length in bytes of the *verification_pattern* parameter.

On input: for **GENERATE** the length must be at least 8 bytes; for **VERIFY** the length must be 8 bytes.

On output for **GENERATE** the length of the verification pattern returned.

verification_pattern

Direction: Input/Output
Type: String

For **GENERATE**, the verification pattern generated for the key.

For **VERIFY**, the supplied verification pattern to be verified.

Restrictions

The restrictions for CSNBKYT2.

The **key_identifier** parameter must not identify a key label when the input key is in a TR-31 key block.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Required commands

The required commands for CSNBKYT2.

This verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

For verification rule keyword **ENC-ZERO** together with algorithm keyword **AES**, the **Key Test2 - AES, ENC-ZERO** command (offset X'0021') must be enabled in the active rule. This command is not required for algorithm keyword **DES**.

Usage notes

The usage notes for CSNBKYT2.

You can generate the verification pattern for a key when you generate the key. You can distribute the pattern with the key and it can be verified at the receiving node. In this way, users can ensure using the same key at the sending and receiving locations. You can generate and verify keys of any combination of key forms, that is, clear, operational or external.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKYT2J.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBKYT2J(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  hikmNativeInteger rule_array_count,
  byte[]            rule_array,
  hikmNativeInteger key_identifier_length,
  byte[]            key_identifier,
  hikmNativeInteger key_encrypting_key_identifier_length,
  byte[]            key_encrypting_key_identifier,
  hikmNativeInteger reserved_length,
  byte[]            reserved,
  hikmNativeInteger verification_pattern_length,
  byte[]            verification_pattern);
```

Key Test Extended (CSNBKYTX)

This verb is essentially the same as Key Test (CSNBKYT).

For further information, see "Key Test (CSNBKYT)" on page 199. The differences are:

- In addition to operating on internal keys and key parts, this verb also operates on external keys and key parts.
- This verb does not operate on clear keys, and does not accept *rule_array* keywords **CLR-A128**, **CLR-A192**, **CLR-A256**, **KEY-CLR**, and **KEY-CLRD**.

See also "Key Test (CSNBKYT)" on page 199 for operating only on internal keys.

Use this verb to verify the value of a key or key part in an external or internal key token. This verb supports two options:

GENERATE
To compute and return a verification pattern for a specified key.

VERIFY
To verify that a passed verification pattern is correct for the specified key.

The verification pattern and the verification process do not reveal any information about the value of the tested key, other than equivalency of two key values. Several verification algorithms are supported.

This verb supports testing of AES (Release 3.30 or later), DES, and PKA master keys, and enciphered keys or key parts. *rule_array* keywords are used to specify information about the target key that is not implicit from other verb parameters.

When testing the master keys, there are two sets of *rule_array* keywords to indicate what key to test:

1. The **SYM-MK**, **ASYM-MK**, and **AES-MK** (Release 3.30 or later) master-key selector keywords indicate whether to test the DES (symmetric) master key, the PKA (asymmetric) master key, or the AES master key.
2. The **KEY-KM**, **KEY-NKM**, and **KEY-OKM** key or key-part *rule_array* keywords choose among the current-master-key register, the new-master-key register, and the old-master-key register.

Not specifying a master-key selector keyword (**SYM-MK**, **ASYM-MK**, or **AES-MK**) means that the DES (symmetric) and PKA (asymmetric) master keys have the same value, and that you want to test that value.

Several key test algorithms are supported by the verb. See "Cryptographic key-verification techniques" on page 927. Some are implicitly selected based on the type of key you are testing, while others are optional and selected by specifying a verification process rule keyword. You can specify one of the following:

1. The **ENC-ZERO** keyword to encrypt a block of binary zeros with the specified key. This verb returns the leftmost 32 bits of the encryption result as the verification pattern. The encrypted block consists of 16 bytes of binary zeros for AES, and eight bytes for DES and Triple-DES keys. This method is valid only with the **TOKEN** keyword for AES, and **KEY-ENC** and **KEY-ENCD** keywords for DES.
2. The **MDC-4** keyword to compute a 16-byte verification pattern using the MDC-4 algorithm. This keyword is valid only when computing the verification pattern for a DES (symmetric) or PKA (asymmetric) master key.
3. The **SHA-1** keyword to compute the verification pattern using the SHA-1 hashing method. This keyword is valid only when computing the verification pattern for the DES (symmetric) or PKA (asymmetric) master key.
4. The **SHA-256** keyword to compute the verification pattern using the SHA-256 hashing method. This keyword is valid only when computing the verification pattern for an AES key.

Table 54 on page 200 describes the use of the *random_number* and *verification_pattern* fields for each of the available verification methods.

Note: For historical reasons, the verification information is passed in two 8-byte variables pointed to by the *random_number* and *verification_pattern* parameters. The **GENERATE** option returns information in these two variables, and the **VERIFY** option uses the information provided in these two variables. If the verb cannot verify the information provided, it returns a return code of 4 and a reason code of 1. For simplicity, these two variables can be two 8-byte elements of a 16-byte array, which is processed by your application program as a single quantity. Both parameters must be coded when calling the API.

DES and Triple-DES keys reserve the low-order bit of each byte for parity. If parity is used, the low-order bit is set so that the total number of B'1' bits in the byte is odd. These parity adjustment keywords allow you to control how the Key Test Extended verb handles the parity bits:

NOADJUST
Specifies not to alter the parity bit values in any way. This is the default.

ADJUST
Specifies to modify the low-order bit of each byte as necessary for odd parity.

This is done on the cleartext value of the key before the verification pattern is computed. The parity adjustment is performed only on a temporary copy of the key within the card, and does not affect the key value in the *key_identifier* parameter.

Format

The format of CSNBKYTX.

```
CSNBKYTX(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_identifier,
             random_number,
             verification_pattern)
```

Parameters

The parameters for CSNBKYTX.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 2, 3, 4, or 5.

rule_array

Direction: Input
Type: String array

Between two and five keywords provide control information to the verb. The keywords must be in contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. The *rule_array* keywords are described in Table 57.

Table 57. Keywords for Key Test Extended control information

Keyword	Description
Process rule (One required)	
GENERATE	Generate a verification pattern for the key supplied in *key_identifier*.
VERIFY	Verify a verification pattern for the key supplied in *key_identifier*.
Key or key-part rule (One required)	
KEY-ENC	Specifies that the key supplied in *key_identifier* is a single-length encrypted key.
KEY-ENCD	Specifies that the key supplied in *key_identifier* is a double-length encrypted key.
KEY-KM	Specifies that the target is the master key register.

Table 57. Keywords for Key Test Extended control information (continued)

Keyword	Description
KEY-NKM	Specifies that the target is the new master-key register.
KEY-OKM	Specifies that the target is the old master-key register.
TOKEN	Process an AES clear or encrypted key contained in an AES key-token.
Master-key selector (One, optional). Use only with **KEY-KM**, **KEY-NKM**, or **KEY-OKM** keywords. The default is to process the **ASYM-MK** and **SYM-MK** key registers, which must have the same key for the default to be valid.	
AES-MK	Process one of the AES master-key registers.
APKA-MK	Process one of the APKA master-key registers. This keyword was introduced with CCA 4.1.0.
ASYM-MK	Specifies use of only the asymmetric master-key registers.
SYM-MK	Specifies use of only the symmetric master-key registers.
Parity adjustment (One, optional) Not valid with the **AES-MK** *Master-key selector* keyword.	
ADJUST	Adjust the parity of test key to odd before generating or verifying the verification pattern. The *key_identifier* field itself is not adjusted.
NOADJUST	Do not adjust the parity of test key to odd before generating or verifying the verification pattern. This is the default.
Verification process rule (One, optional) For the AES master key, **SHA-256** is the default. For the DES or PKA master keys, the default is **SHA-1** if the first and third parts of the key are different, or the IBM z/OS method if the first and third parts of the key are the same.	
ENC-ZERO	Specifies use of the "encrypted zeros" method. Use only with the **KEY-CLR**, **KEY-CLRD**, **KEY-ENC**, or **KEY-ENCD** keywords.
MDC-4	Specifies use of the MDC-4 master key verification method. Use only with the **KEY-KM**, **KEY-NKM**, or **KEY-OKM** keywords. You must specify one master-key selector keyword to use this keyword.
SHA-1	Specifies use of the SHA-1 master-key-verification method. Use only with the **KEY-KM**, **KEY-NKM**, or **KEY-OKM** keywords. You must specify one master-key selector keyword to use this keyword.
SHA-256	Specifies use of the SHA-256 master-key-verification method.

key_identifier

Direction: Input
Type: String

A pointer to a string variable containing an internal or external key-token, a key label that identifies an internal or external key-token record, or a clear key.

The key token contains the key or the key part used to generate or verify the verification pattern.

random_number

Direction: Input/Output
Type: String

A pointer to a string variable containing a number the verb might use in the verification process. When you specify the **GENERATE** keyword, the verb returns the random number. When you specify the **VERIFY** keyword, you must supply the number. With the **ENC-ZERO** method, the *random_number* variable is not used but must be specified.

verification_pattern

Direction: Input/Output
Type: String

A pointer to a string variable containing the binary verification pattern. When you specify the **GENERATE** keyword, the verb returns the verification pattern. When you specify the **VERIFY** keyword, you must supply the verification pattern. With the **ENC-ZERO** method, the verification data occupies the high-order four bytes, while the low-order four bytes are unspecified (the data is passed between your application and the cryptographic engine but is otherwise unused). For more detail, see "Cryptographic key-verification techniques" on page 927.

kek_key_identifier

Direction: Input
Type: String

A pointer to a string variable containing an operational key-token or the key label of an operational key-token record containing an IMPORTER or EXPORTER key-encrypting key. If the *key_identifier* parameter does not identify an external key-token, the contents of the *kek_key_identifier* variable should contain a null DES key-token.

Restrictions

The restrictions for CSNBKYTX.

1. Releases earlier than Release 3.20 do not support the **ADJUST** and **NOADJUST** parity adjustment keywords.
2. AES keys and keywords are not supported in releases before Release 3.30.

Required commands

The required commands for CSNBKYTX.

In order to access key storage, this verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

To enable warning for the case when the rule-array keyword is inconsistent with the key length, enable the command **Key Test - Warn when keyword inconsistent with key length** (offset X'01CB').

Usage notes

The usage notes for CSNBKYTX.

You can generate the verification pattern for a key when you generate the key. You can distribute the pattern with the key and it can be verified at the receiving node. In this way, users can ensure using the same key at the sending and receiving locations. You can generate and verify keys of any combination of key forms: clear, operational, or external.

The parity of the key is not tested.

For triple-length keys, use **KEY-ENC** or **KEY-ENCD** with **ENC-ZERO**. Clear triple-length keys are not supported.

In the Transaction Security System, **KEY-ENC** and **KEY-ENCD** both support enciphered single-length and double-length keys. They use the key-form bits in byte 5 of the control vector (CV) to determine the length of the key. To be consistent, in this implementation of CCA, both **KEY-ENC** and **KEY-ENCD** handle single- and double-length keys. Both products effectively ignore the keywords, which are supplied only for compatibility reasons.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKYTXJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBKYTXJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            key_identifier,
           byte[]            random_number,
           byte[]            verification_pattern,
           byte[]            kek_key_identifier);
```

Key Token Build (CSNBKTB)

The Key Token Build verb assembles a fixed-length symmetric key-token in application storage from information you supply, either as an internal fixed-length AES or DES key-token, or as an external fixed-length DES token. CCA does not support fixed-length external AES key tokens,

This verb can include a control vector that you supply or can build a control vector based on the key type and the control vector related keywords in the *rule_array*. The Key Token Build verb does not perform cryptographic services on any key value. You cannot use this verb to change a key or to change the control vector related to a key.

Format

The format of CSNBKTB.

```
CSNBKTB(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             key_token,
             key_type,
             rule_array_count,
             rule_array,
             key_value,
             reserved_1,
             reserved_2,
             token_data,
             control_vector,
             reserved_4,
             reserved_5,
             reserved_6,
             master_key_verification_pattern)
```

Note: Previous implementations used the *reserved_1* parameter to point to a four-byte integer or string that represented the master key verification pattern. In current versions, CCA requires this parameter to point to a four-byte value equal to binary zero.

Parameters

The parameters for CSNBKTB.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

key_token

Direction: Input/Output
Type: String

The *key_token* parameter is a pointer to a string variable containing the assembled *key_token*.

Note: This variable cannot contain a key label.

key_type

Direction: Input
Type: String

The *key_type* parameter is a pointer to a string variable containing a keyword that defines the key type. The keyword is eight bytes in length and must be left-aligned and padded on the right with space characters.

Valid AES key type keywords are:

```
CLRAES      DATA
```

Valid DES key type keywords are:

```
CIPHER      CVARXCVL    DKYGENKY    MAC         USE-CV
CIPHERXI    CVARXCVR    ENCIPHER    MACVER
CIPHERXL    DATA        EXPORTER    OKEYXLAT
CIPHERXO    DATAC       IKEYXLAT    OPINENC
CVARDEC     DATAM       IMPORTER    PINGEN
CVARENC     DATAMV      IPINENC     PINVER
CVARPINE    DECIPHER    KEYGENKY    SECMSG
```

Specify the **USE-CV** keyword to indicate that the key type should be obtained from the *control_vector* variable.

rule_array_count

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1, 2, 3, 4, 5, or 6.

rule_array

Direction: Output
Type: String array

One to four keywords that provide control information to the verb. The keywords must be in contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. For any key type, there are no more than four valid *rule_array* values. The *rule_array* keywords are described in Table 58.

Table 58. Keywords for Key Token Build control information

Keyword	Description
Token type (One required)	

Table 58. Keywords for Key Token Build control information (continued)

Keyword	Description
EXTERNAL	An external key token.
INTERNAL	An internal key token.
Token algorithm (One, optional)	
AES	An AES key. Only valid for CLRAES or DATA. If CLRAES is specified, this is the default token algorithm.
DES	A DES key. Not valid for CLRAES. If CLRAES is not specified, this is the default token algorithm.
Key status (One, optional).	
KEY	The key token to build will contain an encrypted key. The *key_value* parameter identifies the field that contains the key.
NO-KEY	The key token to build will not contain a key. This is the default key status.
Key length (one keyword required for **AES** keys, one optional for **DES** keys)	
KEYLN8	Single-length or 8-byte key. Valid only for **DES** keys.
KEYLN16	Specifies that the key is 16 bytes long.
KEYLN24	Specifies that the key is 24 bytes long.
KEYLN32	Specifies that the key is 32 bytes long. Valid only for **AES** keys.
DOUBLE	Double-length or 16-byte key. Synonymous with KEYLN16. Valid only for **DES** keys.
DOUBLE-O	Double-length key with guaranteed unique 8-byte key halves. The key is 16 bytes long. Valid only for **DES** keys.
MIXED	Double-length key. Indicates that the key can either be a replicated single-length key (both key halves equal), or a double-length key with two different 8–byte values. Valid only for **DES** keys.
SINGLE	Single-length or 8-byte key. Synonymous with KEYLN8. Valid only for DES keys.
Key Part Indicator (optional). Valid only for **DES** keys.	
KEY-PART	This token is to be used as input to the Key Part Import service.
CV source (One, optional). Valid only for **DES** keys.	
CV	The verb is to obtain the control vector from the variable identified by the *control_vector* parameter.
NO-CV	The control vector is to be supplied based on the key type and the control vector related keywords. This is the default.
Control vector on the link specification (optional). Valid only for IMPORTER and EXPORTER.	
CV-KEK	This keyword indicates marking the KEK as a CV KEK. The control vector is applied to the KEK prior to its use in encrypting other keys. This is the default.
NOCV-KEK	This keyword indicates marking the KEK as a NOCV KEK. The control vector is not applied to the KEK prior to its use in encrypting other keys.
Master key verification pattern (optional). Valid only for **DES** keys.	

Table 58. Keywords for Key Token Build control information (continued)

Keyword	Description
MKVP	This keyword indicates that the **key_value** is enciphered under the master key which corresponds to the pattern specification in the **master_key_verification_pattern** parameter. If this keyword is not specified, the key contained in the **key_value** field must be enciphered under the current master key.
Key-wrapping method (One, optional). Valid only for **DES** keys.	
WRAP-ENH	Use enhanced key wrapping method, which is compliant with the ANSI X9.24 standard. This keyword was introduced with CCA 4.1.0.
WRAP-ECB	Use original key wrapping method, which uses ECB wrapping for DES key tokens and CBC wrapping for AES key tokens. This keyword was introduced with CCA 4.1.0.
Translation control (Optional). Valid only for **DES** keys.	
ENH-ONLY	Restrict re-wrapping of the *output_key_token*. After the token has been wrapped with the enhanced method, it cannot be re-wrapped using the original method. This keyword was introduced with CCA 4.1.0.

See Figure 3 on page 43 for the key usage keywords that can be specified for a given key type.

The difference between Key Token Parse (CSNBKTP) and Control Vector Generate (CSNBCVG) is that Key Token Parse returns the *rule_array* keywords that apply to a parsed token, such as **EXTERNAL**, **INTERNAL**, and so forth. These *rule_array* parameters are returned in addition to the *key_type* parameter.

```
AMEX-CSC        DKYL0       EPINGEN      KEYLN16      UKPT
ANSIX9.9        DKYL1       EPINGENA     LMTD-KEK     VISA-PVV
ANY             DKYL2       EPINVER      MIXED        WRAP-ECB
ANY-MAC         DKYL3       EXEX         NO-SPEC      WRAP-ENH
CLR8-ENC        DKYL4       EXPORT       NO-XPORT     XLATE
CPINENC         DKYL5       GBP-PIN      NOOFFSET     XPORT-OK
CPINGEN         DKYL6       GBP-PINO     NOT-KEK
CPINGENA        DKYL7       IBM-PIN      OPEX
CVVKEY-A        DMAC        IBM-PINO     OPIM
CVVKEY-B        DMKEY       IMEX         PIN
DALL            DMPIN       IMIM         REFORMAT
DATA            DMV         IMPORT       SINGLE
DDATA           DOUBLE      INBK-PIN     SMKEY
DEXP            DPVR        KEY-PART     SMPIN
DIMP            ENH-ONLY    KEYLN8       TRANSLAT
```

key_value

Direction: Output
Type: String

This parameter is a pointer to a string variable containing the enciphered key or AES clear-key value which is placed into the key field of the key token when you use the **KEY** *rule_array* keyword. If the **KEY** keyword is not specified, this parameter is ignored.

The length of this variable depends on the type of key that is provided. The length is 16 bytes for DES keys. A single-length DES key must be left-aligned and padded on the right with eight bytes of X'00'. For a clear AES key, the length is 16 bytes for **KEYLN16**, 24 bytes for **KEYLN24**, and 32 bytes for **KEYLN32**. An enciphered AES key is 32 bytes.

reserved_1

Direction: Output
Type: Integer

This parameter is a pointer to an integer variable or a 4-byte string variable. The value must be equal to an integer valued 0.

reserved_2

Direction: Output
Type: Integer

This parameters is a pointer to an integer variable. The value must be 0 or a null pointer.

token_data_1

Direction: Input
Type: String

This parameter is unused for DES keys and clear text AES keys. In either of those cases it must be a null pointer or point to a string variable containing eight bytes of binary zeros. For encrypted AES keys, this parameter is a pointer to a one-byte string variable containing the LRC value for the key passed in the *key_value* parameter. For more information on LRC values, see *IBM CCA Basic Services Reference and Guide for the IBM 4765 PCIe and IBM 4764 PCI-X Cryptographic Coprocessors*.

control_vector

Direction: Output
Type: String

A parameter is a pointer to a string variable. If you specify the **CV** keyword in the *rule_array*, the contents of this variable are copied to the control vector field of the fixed-length DES key token. If the **CV** keyword is not specified, this keyword is ignored.

reserved_4

Direction: Output
Type: String

This parameter is a pointer to a string variable. The value must be binary zeros or a null pointer.

reserved_5

Direction: Output
Type: Integer

This parameter is a pointer to an integer variable. The value must be 0 or a null pointer.

reserved_6

Direction: Output
Type: String

This parameter is a pointer to an 8-byte string variable. The value must eight space characters or a null pointer.

master_key_verification_pattern

Direction: Output
Type: String

This parameter is a pointer to a string variable containing the master-key verification pattern of the master key used to encipher the key in the internal key-token. The contents of the variable are copied into the MKVP field of the of the key token when keywords **INTERNAL** and **KEY** are specified, and *key_type* keyword **CLRAES** is not specified.

Restrictions

The restrictions for CSNBKTB.

None.

Required commands

The required commands for CSNBKTB.

None.

Usage notes

The usage notes for CSNBKTB.

Because 24-byte (TRIPLE) DES keys can only be generated as **DATA** keys, capability to create 24-byte DES tokens (with keywords **TRIPLE** or **KEYLN24** has not been added to Key Token Build (CSNBKTB). Instead, call Key Generate (CSNBKGN) directly.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKTBJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBKTBJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  byte[]            key_token,
  byte[]            key_type,
  hikmNativeInteger rule_array_count,
  byte[]            rule_array,
  byte[]            key_value,
  byte[]            reserved_1,
  hikmNativeInteger reserved_2,
  byte[]            reserved_3,
  byte[]            control_vector,
  hikmNativeInteger reserved_4,
  byte[]            reserved_5,
  byte[]            reserved_6,
  byte[]            master_key_verification_pattern);
```

Key Token Build2 (CSNBKTB2)

Use the Key Token Build2 verb to assemble an internal variable-length symmetric key-token in application storage from information that you supply.

This verb assembles the information as a skeleton keyed hash Message Authentication Code (HMAC) internal key token. This skeleton token can be supplied to the Key Generate2 verb, which then provides a completed key token with the attributes of the skeleton along with a randomly generated key. These attributes become cryptographically bound to the key when it is enciphered.

The Key Token Build2 verb cannot assemble a usable key-token that contains an enciphered key. It can assemble an internal HMAC key-token that has either a clear key, usable for a limited number of services, or no key, which is only usable for passing to the Key Generate2 verb in order to receive an enciphered key.

The Key Token Build2 verb is a host-only verb and it does not use the cryptographic coprocessor. This verb does not perform cryptographic services on any key value. You cannot use this verb to change a key or to change the control vector related to a key.

Format

The format of CSNBKTB2.

```
CSNBKTB2(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              rule_array_count,
              rule_array,
              clear_key_bit_length,
              clear_key_value,
              key_name_length,
              key_name,
              user_associated_data_length,
              user_associated_data,
              token_data_length,
              token_data,
              verb_data_length,
              verb_data,
              target_key_token_length,
              target_key_token)
```

Parameters

The parameters for CSNBKTB2.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

The number of keywords you supplied in the *rule_array* parameter. The minimum value is 4.

rule_array

Direction: Input
Type: String array

The *rule_array* contains keywords that provide control information to the verb.

The keywords must be in contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. The *rule_array* keywords are described in Table 59.

Table 59. Keywords for Key Token Build2

Keyword	Description
Header section	
Token identifier (one required)	
EXTERNAL	Specifies to build an external variable-length symmetric key-token.
INTERNAL	Specifies to build an internal variable-length symmetric key-token.
Wrapping information section	
Key status (one, optional)	
NO-KEY	Build the key token without a key value. This creates a skeleton key token that can later be supplied to the Key Generate2 (CSNBKGN2) verb. This is the default.
KEY-CLR	Build the key token with a clear key, AES CIPHER and HMAC MAC keys only.
Associated data section	
Type of algorithm for which the key can be used (one required)	
AES	Specifies to build an AES key token.
HMAC	Specifies to build an HMAC key token.
Key type (one required)	
CIPHER	Build a CIPHER key for encryption, decryption, and translation operations. AES algorithm only. • Refer to Figure 5 on page 223 for valid rule array keyword combinations. • Refer to Table 60 on page 224 for token offsets, offset values, and meanings of these keywords. • Refer to Table 235 on page 814 for the format of the key token.
DKYGENKY	Build a diversifying key generating key. AES algorithm only. • Refer to Figure 6 on page 226 for valid rule array keyword combinations. • Refer to Table 61 on page 227 for token offsets, offset values, and meanings of these keywords. • Refer to "AES DKYGENKY variable-length symmetric key token" on page 860 for the format of the key token.
EXPORTER	Build an EXPORTER key-encrypting key. AES algorithm only. • Refer to Figure 7 on page 230 for valid rule array keyword combinations. • Refer to Table 62 on page 231 for token offsets, offset values, and meanings of these keywords. • Refer to "AES EXPORTER and IMPORTER variable-length symmetric key token" on page 837 for the format of the key token.
IMPORTER	Build an IMPORTER key-encrypting key. AES algorithm only. • Refer to Figure 7 on page 230 for valid rule array keyword combinations. • Refer to Table 62 on page 231 for token offsets, offset values, and meanings of these keywords. • Refer to "AES EXPORTER and IMPORTER variable-length symmetric key token" on page 837 for the format of the key token.
MAC	Build a MAC key for message authentication code operations. AES (Release 4.4 or later) and HMAC algorithms only. For AES: • Refer to Figure 9 on page 238 for valid rule array keyword combinations. • Refer to Table 64 on page 238 for token offsets, offset values, and meanings of these keywords. • Refer to "AES MAC variable-length symmetric key token" on page 821 for the format of the key token. For HMAC: • Refer to Figure 10 on page 241 for valid rule array keyword combinations. • Refer to Table 65 on page 241 for token offsets, offset values, and meanings of these keywords. • Refer to "HMAC MAC variable-length symmetric key token" on page 830 for the format of the key token.

Table 59. Keywords for Key Token Build2 (continued)

Keyword	Description
PINCALC	Build a DK PIN calculating key. AES algorithm only. • Refer to Figure 11 on page 244 for valid rule array keyword combinations. • Refer to Table 66 on page 244 for token offsets, offset values, and meanings of these keywords. • Refer to "AES PINPROT, PINCALC, and PINPRW variable-length symmetric key token" on page 847 for the format of the key token.
PINPROT	Build a DK PIN protection key. AES algorithm only. • Refer to Figure 12 on page 246 for valid rule array keyword combinations. • Refer to Table 67 on page 247 for token offsets, offset values, and meanings of these keywords. • Refer to "AES PINPROT, PINCALC, and PINPRW variable-length symmetric key token" on page 847 for the format of the key token.
PINPRW	Build a DK PIN PRW key. AES algorithm only. • Refer to Figure 13 on page 249 for valid rule array keyword combinations. • Refer to Table 68 on page 249 for token offsets, offset values, and meanings of these keywords. • Refer to "AES PINPROT, PINCALC, and PINPRW variable-length symmetric key token" on page 847 for the format of the key token.
SECMSG (Release 4.4 or later)	Build a secure messaging key. AES algorithm only. • Refer to Figure 14 on page 251 for valid rule array keyword combinations. • Refer to Table 69 on page 252 for token offsets, offset values, and meanings of these keywords. • Refer to "AES SECMSG variable-length symmetric key token" on page 869 for the format of the key token.

clear_key_bit_length

Direction: Input
Type: Integer

The length of the clear key in bits. Specify 0 when no key value is supplied or a valid **HMAC** key bit length, between 80 and 2048.

clear_key_value

Direction: Input
Type: String

This parameter is used when the **KEY-CLR** keyword is specified. This parameter is the clear key value to be put into the token being built.

key_name_length

Direction: Input
Type: Integer

The length of the *key_name* parameter. Valid values are 0 and 64.

key_name

Direction: Input
Type: String

A 64-byte key store label to be stored in the associated data structure of the token.

user_associated_data_length

Direction: Input
Type: Integer

The length of the user-associated data. The valid values are 0 - 255 bytes.

user_associated_data

Direction: Input
Type: String

User-associated data to be stored in the associated data structure.

token_data_length

Direction: Input
Type: Integer

This parameter is reserved. This value must be 0.

token_data

Direction: n/a
Type: String

This parameter is ignored.

verb_data_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the **verb_data** variable. The value must be 0.

verb_data

Direction: Input
Type: String

A pointer to a string variable containing key-usage field keywords that are related to the type of key to diversify.

DKYUSAGE specifies that the **verb_data** variable contains all of the keywords necessary to define the key usage attributes related to the type of key to diversify. Based on the **verb_data** keywords, CSNBKTB2 appends the key usage attributes of the type of key to diversify to the key usage fields of the DKYGENKY key. The related key usage fields control which key usage attributes are permissible for the finally generated diversified key.

DKYUSAGE is not valid with D-ALL, because the type of key to diversify is unspecified. DKYUSAGE is optional with D-CIPHER, D-EXP, and D-IMP. For these key types, if DKYUSAGE is not specified, CSNBKTB2 assigns default key usage attributes to the related KUF fields. DKYUSAGE is required for the remaining values of type of key to diversify, because those key types do not have default key usage attributes.

target_key_token_length

Direction: Input/Output
Type: Integer

On input, the length of the *target_key_token* parameter supplied to receive the token. On output, the actual length of the token returned to the caller. Maximum length is 725 bytes.

target_key_token

Direction: Output

Type: String

The key token built by this verb.

Keywords reference

Here you find a keywords reference for the CSNBKTB2 **rule_array** parameter.

Figure 5 shows all the valid keyword combinations and their defaults for AES key type CIPHER. For a description of these keywords, refer to Table 60 on page 224.

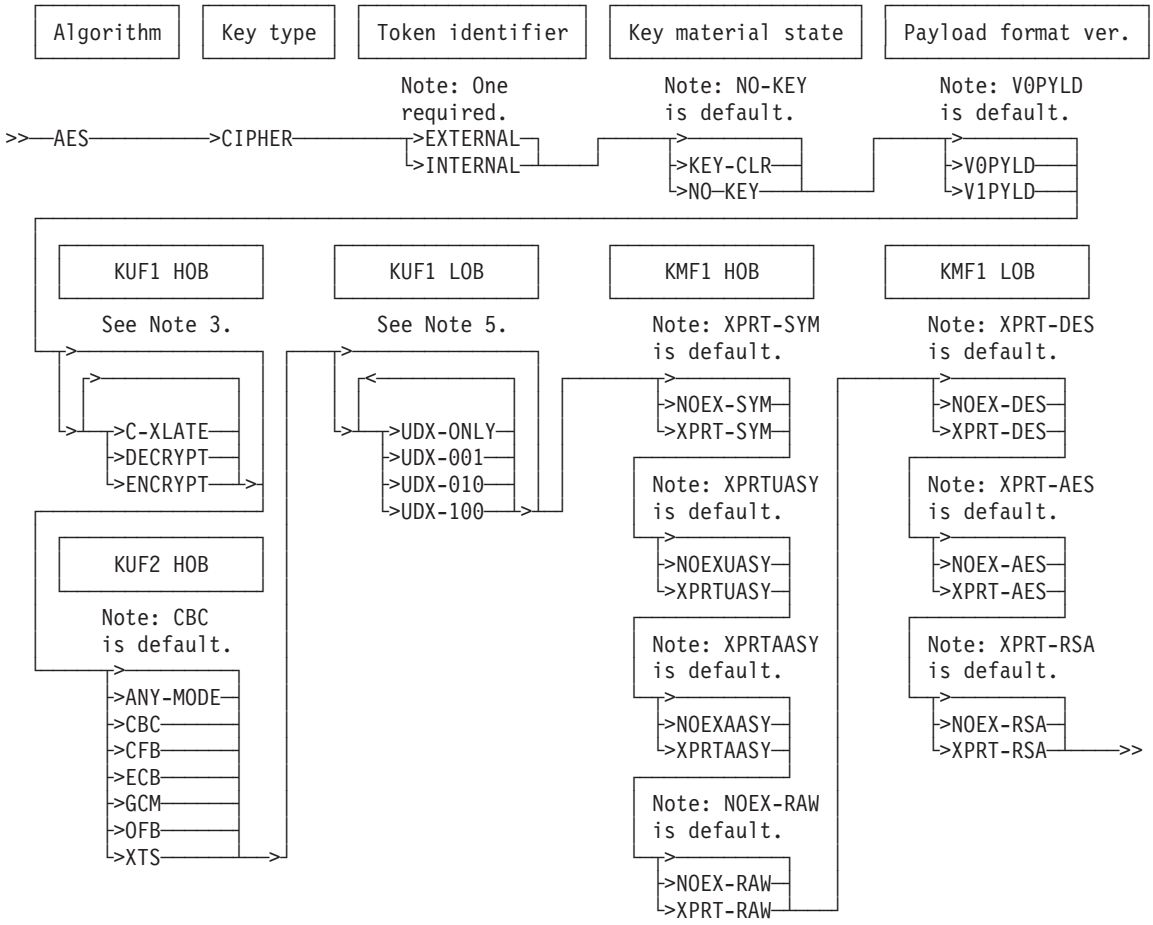


Figure 5. Key Token Build2 keyword combinations for AES CIPHER keys

Note: f

1. V0PYLD and V1PYLD are for Release 4.4 or later. V0PYLD is the default for compatibility reasons. V1PYLD is recommended because it provides improved security.
2. Each key-usage field (KUF) and key-management field (KMF) of a version X'05' variable-length symmetric key-token consists of a high-order byte (HOB) and a low-order byte (LOB).
3. DECRYPT and ENCRYPT are defaults if neither of these keywords is specified, regardless of whether C-XLATE is specified or not. C-XLATE is for Release 4.3 or later.
4. ANY-MODE is Release 4.4 or later.

5. Choose any number of keywords in this group. No keywords in this group are defaults.
6. NOEX-RAW and XPRT-RAW are defined for future use and their meanings are currently undefined. To avoid this export restriction in the future when the meaning is defined, specify XPRT-RAW.

Table 60. Key Token Build2 rule array keywords for AES CIPHER keys

Token offset	Rule-array keyword	Offset value	Meaning
Key-token header section			
Token identifier (one required)			
000	**EXTERNAL**	X'02'	Build a key token that is not to be used locally.
	INTERNAL	X'01'	Build a key token that is to be used locally.
Wrapping-information section			
Key status (one, optional)			
008	**KEY-CLR**	X'01'	Build a key token that contains a clear key.
	NO-KEY	X'00'	Build a key token that does not contain a key value. This is the default.
Payload format version (one, optional). Identifies format of the payload.			
000	**V0PYLD**	X'00'	Build a key token with a version 0 payload format. This format has a variable length and the key length can be inferred from the size of the payload. This format is compatible with all releases. This is the default.
	V1PYLD	X'01'	Build the key token with a version 1 payload format. This format has a fixed length and the key length cannot be inferred by the size of the payload. An obscured key length is considered more secure. This format is not compatible in releases before Release 4.4.
Associated data section			
Algorithm type (one required).			
041	**AES**	X'02'	Key can be used for AES algorithm.
Key type (one required)			
042	**CIPHER**	X'0001'	Key can be used for encryption, decryption, and translation of data.
Encryption and translation control (one or more, optional). Key-usage field 1, high-order byte. Keywords **DECRYPT** and **ENCRYPT** are defaults unless one or more keywords in the group are specified.			
045	**C-XLATE** (Release 4.3 or later)	B'xx1x xxxx'	Key can only be used for Cipher Text Translate2 (CSNBCTT2) operations. This is only valid with AES CIPHER keys.
	DECRYPT	B'x1xx xxxx'	Key can be used for decryption. Symmetric_Algorithm_Decipher.
	ENCRYPT	B'1xxx xxxx'	Key can be used for encryption. Symmetric_Algorithm_Encipher.
User-defined extension (UDX) control (one or more, optional). Key-usage field 1, low-order byte. No keywords in the group are defaults.			
046	**UDX-ONLY**	B'xxxx 1xxx'	Key can only be used in UDXs.
	UDX-100	B'xxxx x1uu'	Leftmost user-defined UDX bit is set on.
	UDX-010	B'xxxx xu1u'	Middle user-defined UDX bit is set on.
	UDX-001	B'xxxx xuu1'	Rightmost user-defined UDX bit is set on.

Table 60. Key Token Build2 rule array keywords for AES CIPHER keys (continued)

Token offset	Rule-array keyword	Offset value	Meaning
Encryption mode (one, optional). Key-usage field 2, high-order byte.			
047	**ANY-MODE** (Release 4.4 or later)	X'FF'	Key can be used for any encryption mode.
	CBC	X'00'	Key can be used for Cipher Block Chaining. This is the default.
	CFB	X'02'	Key can be used for Cipher Feedback.
	ECB	X'01'	Key can be used for Electronic Code Book.
	GCM	X'04'	Key can be used for Galois/Counter Mode.
	OFB	X'03'	Key can be used for Output Feedback.
	XTS	X'05'	Key can be used for Xor-Encrypt-Xor-based Tweaked Stealing.
Symmetric-key export control (one, optional). Key-management field 1, high-order byte.			
050	**NOEX-SYM**	B'0xxx xxxx'	Prohibit export using symmetric key.
	XPRT-SYM	B'1xxx xxxx'	Allow export using symmetric key. This is the default.
Unauthenticated asymmetric-key export control (one, optional). Key-management field 1, high-order byte.			
050	**NOEXUASY**	B'x0xx xxxx'	Prohibit export using an unauthenticated asymmetric key (not a trusted block).
	XPRTUASY	B'x1xx xxxx'	Allow export using an unauthenticated asymmetric key (not a trusted block). This is the default.
Authenticated asymmetric-key export control (one, optional). Key-management field 1, high-order byte.			
050	**NOEXAASY**	B'xx0x xxxx'	Prohibit export using an authenticated asymmetric key (trusted block).
	XPRTAASY	B'xx1x xxxx'	Allow export using an authenticated asymmetric key (trusted block). This is the default.
Raw-key export control (one, optional). Key-management field 1, high-order byte.			
050	**NOEX-RAW**	B'xxx0 xxxx'	Prohibit export using raw key. Defined for future use. Currently ignored. This is the default.
	XPRT-RAW	B'xxx1 xxxx'	Allow export using raw key. Defined for future use. Currently ignored.
DES-key export control (one, optional). Key-management field 1, low-order byte.			
051	**NOEX-DES**	B'1xxx xxxx'	Prohibit export using a DES key.
	XPRT-DES	B'0xxx xxxx	Allow export using a DES key. This is the default.
AES-key export control (one, optional). Key-management field 1, low-order byte.			
051	**NOEX-AES**	B'x1xx xxxx'	Prohibit export using an AES key.
	XPRT-AES	B'x0xx xxxx'	Allow export using an AES key.
RSA-key export control (one, optional). Key-management field 1, low-order byte.			
051	**NOEX-RSA**	B'xxxx 1xxx'	Prohibit export using an RSA key.
	XPRT-RSA	B'xxxx 0xxx'	Allow export using an RSA key. This is the default.

Figure 6 on page 226 shows all the valid keyword combinations and their defaults for AES key type DKYGENKY. For a description of these keywords, refer to Table 61 on page 227.

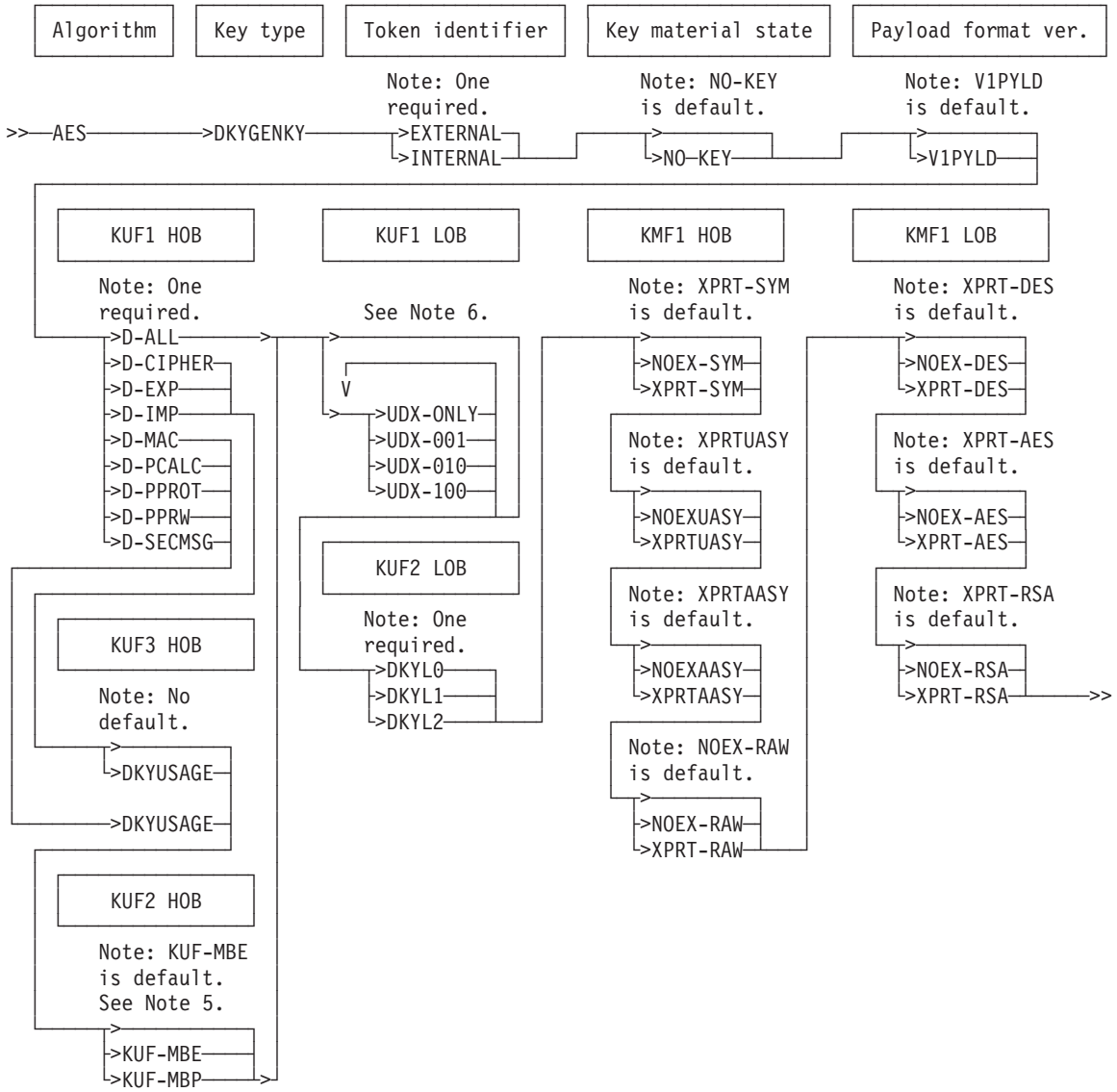


Figure 6. Key Token Build2 keyword combinations for AES DKYGENKY keys

Note:

1. Each key-usage field (KUF) and key-management field (KMF) of a version X'05' variable-length symmetric key-token consists of two bytes: a high-order byte (HOB) and a low-order byte (LOB).
2. D-PCALC, D-PPROT, and D-PPRW are Release 4.4 or later. D-SECMSG, DKYL1, and DKYL2 are Release 4.4 or later.
3. NOEX-RAW and XPRT-RAW are defined for future use and their meanings are currently undefined. To avoid this export restriction in the future when the meaning is defined, specify XPRT-RAW.
4. DKYUSAGE specifies that the verb_data variable contains all of the keywords necessary to define the key usage attributes related to the type of key to diversify. Based on the verb_data keywords, CSNBKTB2 appends the key usage attributes of the type of key to diversify to the key usage fields of the DKYGENKY key. The related key usage fields control which key usage attributes are permissible for the final generated diversified key. DKYUSAGE is not valid with D-ALL because the type of key to diversify is unspecified. DKYUSAGE is optional with D-CIPHER, D-EXP, and D-IMP because key types

CIPHER, EXPORTER and IMPORTER have default key usage attributes. For these key types, if DKYUSAGE is not specified, CSNBKTB2 assigns default key usage attributes to the related KUF fields. DKYUSAGE is required for the remaining values of type of key to diversify because those key types do not have default key usage attributes.

5. KUF-MBP is not valid if DKADMIN1, DKADMIN2, DKPINOP, or DKPINOPP is specified in the **verb_data** variable (that is, the type of key to diversify is DK enabled).
6. Choose any number of keywords in this group. No keywords in this group are defaults.

Table 61. Key Token Build2 rule array keywords for AES DKYGENKY keys

Token offset	Rule-array keyword	Offset value	Meaning
Key-token header section			
Token identifier (one required).			
000	**EXTERNAL**	X'02'	Build a key token that is not to be used locally.
	INTERNAL	X'01'	Build a key token that is to be used locally.
Wrapping-information section			
Key status (one, optional).			
008	**NO-KEY**	X'00'	Build a key token that does not contain a key value. This is the default.
Payload format version (one, optional). Identifies format of the payload. Release 4.4 or later.			
028	**V1PYLD**	X'01'	Build the key token with a version 1 payload format. This format has a fixed length and the key length cannot be inferred by the size of the payload. An obscured key length is considered more secure. This is the default.
Associated data section			
Algorithm type (one required).			
041	**AES**	X'02'	Key can be used for AES algorithm.
Key type (one required).			
042	**DKYGENKY**	X'0009'	Key can be used for generating a diversified key.
Type of key to diversify (one required). Key-usage field 1, high-order byte.			
045	**D-ALL**	X'00'	Key can generate a diversified key for any key type listed herafter.
	D-CIPHER	X'01'	Key can generate a diversified CIPHER key.
	D-EXP	X'03'	Key can generate a diversified EXPORTER key.
	D-IMP	X'04'	Key can generate a diversified IMPORTER key.
	D-MAC	X'02'	Key can generate a diversified MAC key.
	D-PCALC	X'06'	Key can generate a diversified PINCALC key.
	D-PPROT	X'05'	Key can generate a diversified PINPROT key.
	D-PPRW	X'07'	Key can generate a diversified PINPRW key.
User-defined extension (UDX) control (one or more, optional). Key-usage field 1, low-order byte. No keywords in the group are defaults.			

Table 61. Key Token Build2 rule array keywords for AES DKYGENKY keys (continued)

Token offset	Rule-array keyword	Offset value	Meaning
046	**UDX-ONLY**	B'xxxx 1xxx'	Key can only be used in UDXs.
	UDX-100	B'xxxx x1uu'	Leftmost user-defined UDX bit is set on.
	UDX-010	B'xxxx xu1u'	Middle user-defined UDX bit is set on.
	UDX-001	B'xxxx xuu1'	Rightmost user-defined UDX bit is set on.
Related generated key-usage field level of control (one required). Key-usage field 2, high-order byte. If **D-ALL** is specified, **KUF-MBE** and **KUF-MBP** are not valid and there is no default. Otherwise, the default is **KUF-MBE**. **KUF-MBP** is not valid if **DKYUSAGE** is specified and **DKPINOP**, **DKPINOPP**, **DKADMIN1**, or **DKADMIN2** is specified in the verb_data variable (that is, the type of key to diversify is DK enabled).			
047	**KUF-MBE**	B'1xxx xxxx'	The key usage fields of the key to be generated must be equal to the related generated key-usage fields that start with key usage field 3.
	KUF-MBP	B'0xxx xxxx'	The key usage fields of the key to be generated must be permissible based on the related generated key usage fields that start with key-usage field 3. A key to be diversified is not permitted to have a higher level of usage than any of the related key usage fields permit. The key to be diversified is only permitted to have key usage that is less than or equal to the related key usage fields. One exception is the UDX-ONLY setting in the generated key usage fields. The UDX-ONLY setting must always be equal to the UDX-ONLY setting in the related key usage fields.
Key-derivation sequence level (one required). Key-usage field 2, low-order byte.			
048	**DKYL0**	X'00'	Use this diversifying key to generate a Level 0 diversified key. The type of key to diversify (value at offset 45) determines the key type of the generated key. Level 0 is a completed key.
Related generated key usage fields (not allowed for **D-ALL,** one, optional, for **D-CIPHER**, **D-EXP**, and **D-IMP**, otherwise one required). Key-usage field 3, high-order byte.			
049	**DKYUSAGE**	Based on verb_data keywords.	The verb_data variable contains key-usage field keywords related to the type of key to diversify. These related attributes become part of the key usage fields of the DKYGENKY diversifying key, beginning with key-usage field 3, high-order byte. They are related because they are used to control which key usage attributes are permissible in the generated diversified key. To generate a diversified key, use the Diversified Key Generate2 (CSNBDKG2) verb.
Symmetric-key export control (one, optional). Key-management field 1, high-order byte.			
052	**NOEX-SYM**	B'0xxx xxxx'	Prohibit export using symmetric key.
	XPRT-SYM	B'1xxx xxxx'	Allow export using symmetric key. This is the default.
Unauthenticated asymmetric-key export control (one, optional). Key-management field 1, high-order byte.			
052	**NOEXUASY**	B'x0xx xxxx'	Prohibit export using an unauthenticated asymmetric key (not a trusted block).
	XPRTUASY	B'x1xx xxxx'	Allow export using an unauthenticated asymmetric key (not a trusted block). This is the default.
Authenticated asymmetric-key export control (one, optional). Key-management field 1, high-order byte.			
052	**NOEXAASY**	B'xx0x xxxx'	Prohibit export using an authenticated asymmetric key (trusted block).
	XPRTAASY	B'xx1x xxxx'	Allow export using an authenticated asymmetric key (trusted block). This is the default.

Table 61. Key Token Build2 rule array keywords for AES DKYGENKY keys (continued)

Token offset	Rule-array keyword	Offset value	Meaning
RAW-key export control (one, optional). Key-management field 1, high-order byte.			
052	**NOEX-RAW**	B'xxx0 xxxx'	Prohibit export using raw key. Defined for future use. Currently ignored. This is the default.
	XPRT-RAW	B'xxx1 xxxx'	Allow export using raw key. Defined for future use. Currently ignored.
DES-key export control (one, optional). Key-management field 1, low-order byte.			
053	**NOEX-DES**	B'1xxx xxxx'	Prohibit export using a DES key.
	XPRT-DES	B'0xxx xxxx	Allow export using a DES key. This is the default.
AES-key export control (one, optional). Key-management field 1, low-order byte.			
053	**NOEX-AES**	B'x1xx xxxx'	Prohibit export using an AES key.
	XPRT-AES	B'x0xx xxxx'	Allow export using an AES key.
RSA-key export control (one, optional). Key-management field 1, low-order byte.			
053	**NOEX-RSA**	B'xxxx 1xxx'	Prohibit export using an RSA key.
	XPRT-RSA	B'xxxx 0xxx'	Allow export using an RSA key. This is the default.

Figure 7 on page 230 shows all the valid keyword combinations and their defaults for AES key type EXPORTER. For a description of these keywords, refer to Table 62 on page 231.

Key Token Build2 (CSNBKTB2)

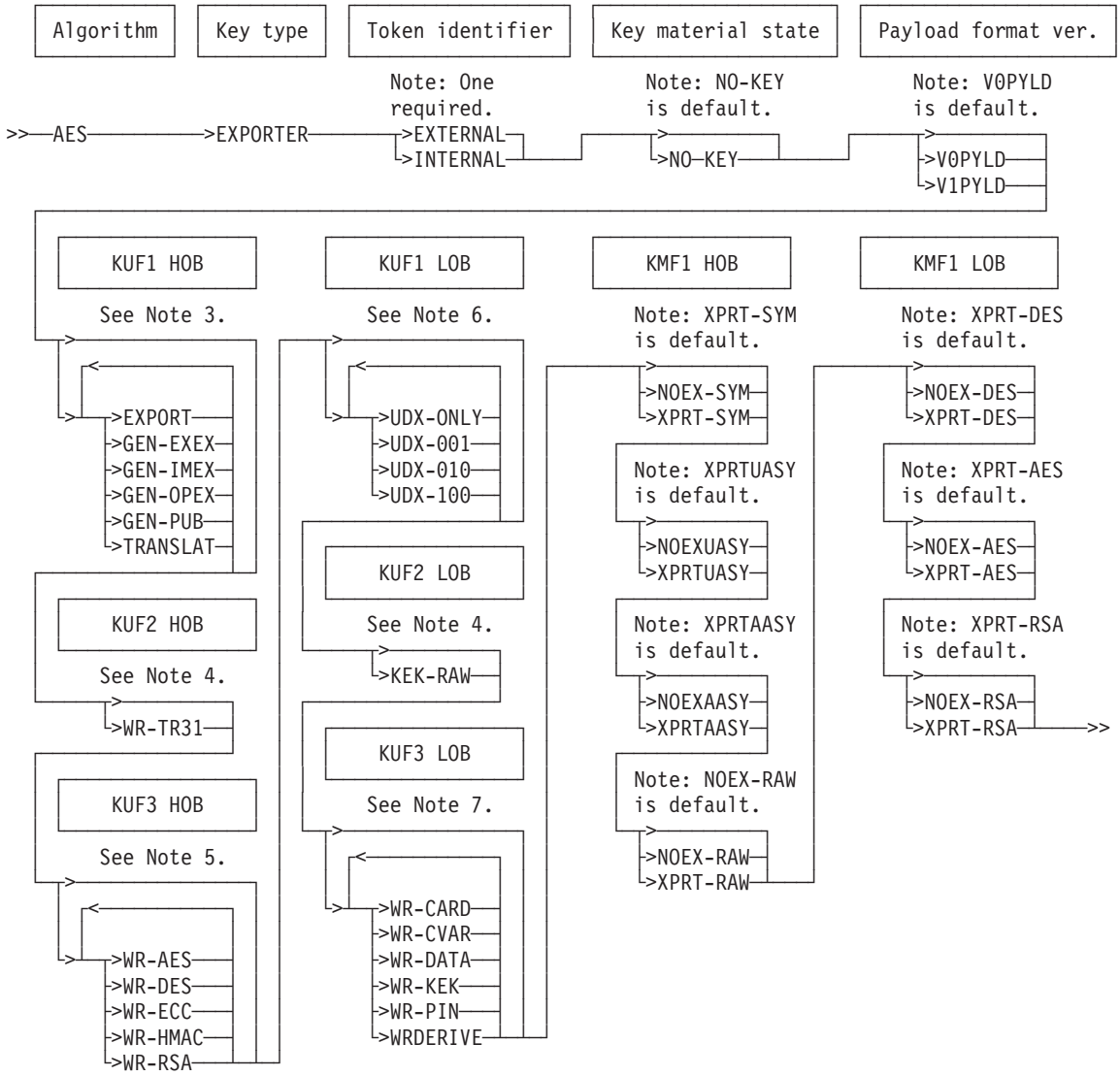


Figure 7. Key Token Build2 keyword combinations for AES EXPORTER keys

Note:

1. **V0PYLD** and **V1PYLD** are for Release 4.4 or later. **V0PYLD** is the default for compatibility reasons. **V1PYLD** is recommended because it provides improved security.
2. Each key-usage field (KUF) and key-management field (KMF) of a version X'05' variable-length symmetric key-token consists of a high-order byte (HOB) and a low-order byte (LOB).
3. All keywords in this group are defaults unless one or more keywords in this group are specified.
4. There is no default. This keyword is defined for future use and its meaning is currently undefined. To avoid this restriction in the future when the meaning is defined, specify this keyword.
5. **WR-AES**, **WR-DES**, and **WR-HMAC** are defaults unless one or more keywords in this group are specified.
6. Choose any number of keywords in this group. No keywords in this group are defaults.

7. **WR-CARD**, **WR-DATA**, **WR-KEK**, **WR-PIN**, and **WRDERIVE** are defaults unless one or more keywords in this group are specified. **WR-CVAR** is for Release 4.4 or later.
8. **NOEX-RAW** and **XPRT-RAW** are defined for future use and their meanings are currently undefined. To avoid this export restriction in the future when the meaning is defined, specify **XPRT-RAW**.

Table 62. Key Token Build2 rule array keywords for AES EXPORTER keys

Token offset	Rule-array keyword	Offset value	Meaning
Key-token header section			
Token identifier (one required).			
000	**EXTERNAL**	X'02'	Build a key token that is not to be used locally.
	INTERNAL	X'01'	Build a key token that is to be used locally.
Wrapping-information section			
Key status (one, optional)			
008	**NO-KEY**	X'00'	Build a key token that does not contain a key value. This is the default.
Payload format version (one, optional). Identifies format of the payload. Release 4.4 or later; otherwise, undefined.			
028	**V0PYLD**	X'00'	Build a key token with a version 0 payload format. This format has a variable length and the key length can be inferred from the size of the payload. This format is compatible with all releases. This is the default.
	V1PYLD	X'01'	Build the key token with a version 1 payload format. This format has a fixed length and the key length cannot be inferred by the size of the payload. An obscured key length is considered more secure.
Associated data section			
Algorithm type (one required)			
041	**AES**	X'02'	Key can be used for AES algorithm.
Key type (one required)			
042	**EXPORTER**	X'0003'	Key can be used to wrap an external key to be taken from this local node or to wrap an output key in the Key_Translate2 verb.
KEK control (one or more, optional). Key-usage field 1, high-order byte. All keywords in the group are defaults unless one or more keywords in the group are specified.			
045	**EXPORT**	B'1xxx xxxx'	Key can be used to wrap a key taken from this local node. Symmetric_Key_Export.
	GEN-EXEX	B'xxxx 1xxx'	Key can be used to wrap the first or the second key that is generated by the CSNBKGN2 verb as part of an EXEX key pair.
	GEN-IMEX	B'xxx1 xxxx'	Key can be used to wrap the second key that is generated by the CSNBKGN2 verb as part of an IMEX key pair.
	GEN-OPEX	B'xx1x xxxx'	Key can be used to wrap the second key that is generated by the CSNBKGN2 verb as part of an OPEX key pair.
	GEN-PUB	B'xxxx x1xx'	Key can be used to wrap the private key (to be used at another node) generated by the CSNDPKG verb as part of an ECC public-private key pair.
	TRANSLAT	B'x1xx xxxx'	Key can be used to wrap an output key in the Key_Translate2 verb. Key_Translate2.

Table 62. Key Token Build2 rule array keywords for AES EXPORTER keys (continued)

Token offset	Rule-array keyword	Offset value	Meaning
User-defined extension (UDX) control (one or more, optional). Key-usage field 1, low-order byte. No keywords in the group are defaults.			
046	**UDX-ONLY**	B'xxxx 1xxx'	Key can only be used in UDXs.
	UDX-100	B'xxxx x1uu'	Leftmost user-defined UDX bit is set on.
	UDX-010	B'xxxx xu1u'	Middle user-defined UDX bit is set on.
	UDX-001	B'xxxx xuu1'	Rightmost user-defined UDX bit is set on.
TR-31 wrap control (one, optional). Key-usage field 2, high-order byte.			
047	**WR-TR31**	B'1xxx xxxx'	Key can wrap a TR-31 key. Defined for future use. Currently ignored.
Raw key wrap control (one, optional). Key-usage field 2, low-order byte.			
048	**KEK-RAW**	B'xxxx xxx1'	Key can wrap a raw key. Defined for future use. Currently ignored.
Algorithm wrap control (one or more, optional). Key-usage field 3, high-order byte. Keywords **WR-AES**, **WR-DES**, and **WR-HMAC** are defaults unless one or more keywords in the group are specified.			
000	**WR-AES**	B'x1xx xxxx'	Key can wrap AES keys.
	WR-DES	B'1xxx xxxx'	Key can wrap DES keys.
	WR-ECC	B'xxxx 1xxx'	Key can wrap ECC keys.
	WR-HMAC	B'xx1x xxxx'	Key can wrap HMAC keys.
	WR-RSA	B'xxx1 xxxx'	Key can wrap RSA keys.
Class wrap control (one or more, optional). Key-usage field 4, high-order byte. Keywords **WR-CARD**, **WR-DATA**, **WR-KEK**, **WR-PIN**, and **WRDERIVE** are defaults unless one or more keywords in the group are specified.			
051	**WR-CARD**	B'xxxx 1xxx'	Key can wrap card class keys.
	WR-CVAR (Release 4.4 or later)	B'xxxx x1xx'	Key can wrap cryptovariable class keys.
	WR-DATA	B'1xxx xxxx'	Key can wrap data class keys.
	WR-KEK	B'x1xx xxxx'	Key can wrap KEK class keys.
	WR-PIN	B'xx1x xxxx'	Key can wrap PIN class keys.
	WRDERIVE	B'xxx1 xxxx'	Key can wrap derivation class keys.
Symmetric-key export control (one, optional). Key-management field 1, high-order byte.			
054	**NOEX-SYM**	B'0xxx xxxx'	Prohibit export using symmetric key.
	XPRT-SYM	B'1xxx xxxx'	Allow export using symmetric key. This is the default.
Unauthenticated asymmetric-key export control (one, optional). Key-management field 1, high-order byte.			
054	**NOEXUASY**	B'x0xx xxxx'	Prohibit export using an unauthenticated asymmetric key (not a trusted block).
	XPRTUASY	B'x1xx xxxx'	Allow export using an unauthenticated asymmetric key (not a trusted block). This is the default.
Authenticated asymmetric-key export control (one, optional). Key-management field 1, high-order byte.			
054	**NOEXAASY**	B'xx0x xxxx'	Prohibit export using an authenticated asymmetric key (trusted block).
	XPRTAASY	B'xx1x xxxx'	Allow export using an authenticated asymmetric key (trusted block). This is the default.
RAW-key export control (one, optional). Key-management field 1, high-order byte.			

Table 62. Key Token Build2 rule array keywords for AES EXPORTER keys (continued)

Token offset	Rule-array keyword	Offset value	Meaning
054	**NOEX-RAW**	B'xxx0 xxxx'	Prohibit export using raw key. Defined for future use. Currently ignored. This is the default.
	XPRT-RAW	B'xxx1 xxxx'	Allow export using raw key. Defined for future use. Currently ignored.
DES-key export control (one, optional). Key-management field 1, low-order byte.			
055	**NOEX-DES**	B'1xxx xxxx'	Prohibit export using a DES key.
	XPRT-DES	B'0xxx xxxx	Allow export using a DES key. This is the default.
AES-key export control (one, optional). Key-management field 1, low-order byte.			
055	**NOEX-AES**	B'x1xx xxxx'	Prohibit export using an AES key.
	XPRT-AES	B'x0xx xxxx'	Allow export using an AES key.
RSA-key export control (one, optional). Key-management field 1, low-order byte.			
055	**NOEX-RSA**	B'xxxx 1xxx'	Prohibit export using an RSA key.
	XPRT-RSA	B'xxxx 0xxx'	Allow export using an RSA key. This is the default.

Figure 8 on page 234 shows all the valid keyword combinations and their defaults for AES key type IMPORTER. For a description of these keywords, refer to Table 63 on page 235.

Key Token Build2 (CSNBKTB2)

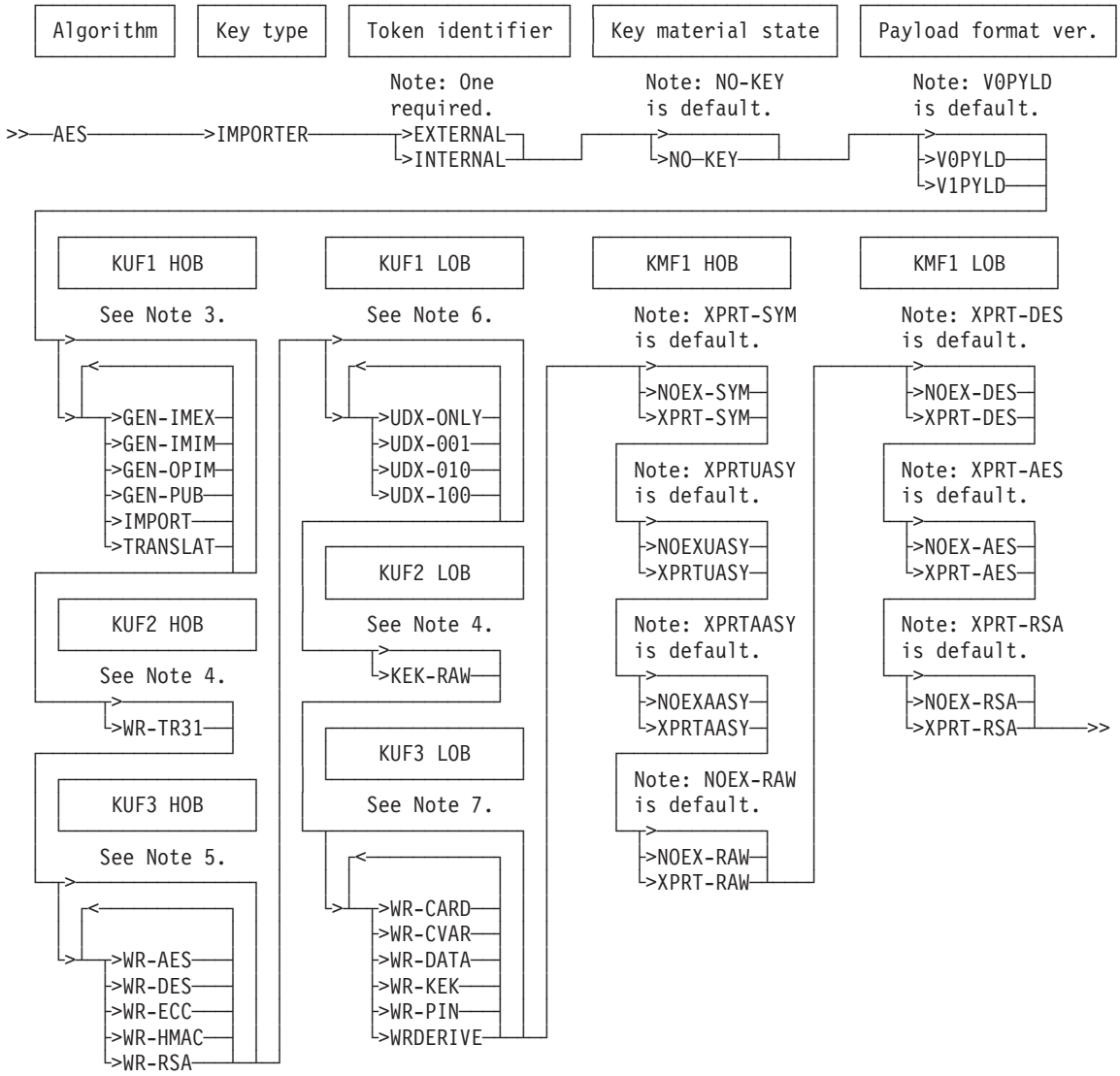


Figure 8. Key Token Build2 keyword combinations for AES IMPORTER keys

Note:

1. V0PYLD and V1PYLD are for Release 4.4 or later. V0PYLD is the default for compatibility reasons. V1PYLD is recommended because it provides improved security.
2. Each key-usage field (KUF) and key-management field (KMF) of a version X'05' variable-length symmetric key-token consists of a high-order byte (HOB) and a low-order byte (LOB).
3. All keywords in this group are defaults unless one or more keywords in this group are specified.
4. This keyword is defined for future use and its meaning is currently undefined. To avoid this restriction in the future when the meaning is defined, specify this keyword.
5. WR-AES, WR-DES, and WR-HMAC are defaults unless one or more keywords in this group are specified.
6. Choose any number of keywords in this group. No keywords in this group are defaults.

7. WR-CARD, WR-DATA, WR-KEK, WR-PIN, and WRDERIVE are defaults unless one or more keywords in this group are specified. WR-CVAR is for Release 4.4 or later.
8. NOEX-RAW and XPRT-RAW are defined for future use and their meanings are currently undefined. To avoid this export restriction in the future when the meaning is defined, specify XPRT-RAW.

Table 63. Key Token Build2 rule array keywords for AES IMPORTER keys

Token offset	Rule-array keyword	Offset value	Meaning
Key-token header section			
Token identifier (one required)			
000	**EXTERNAL**	X'02'	Build a key token that is not to be used locally.
	INTERNAL	X'01'	Build a key token that is to be used locally.
Wrapping-information section			
Key status (one, optional)			
008	**NO-KEY**	X'00'	Build a key token that does not contain a key value. This is the default.
Payload format version (one, optional). Identifies format of the payload.			
028	**V0PYLD**	X'00'	Build a key token with a version 0 payload format. This format has a variable length and the key length can be inferred from the size of the payload. This format is compatible with all releases. This is the default.
	V1PYLD	X'01'	Build the key token with a version 1 payload format. This format has a fixed length and the key length cannot be inferred by the size of the payload. An obscured key length is considered more secure. This format is not compatible in releases before Release 4.4.
Associated data section			
Algorithm type (one required)			
041	**AES**	X'02'	Key can be used for AES algorithm.
Key type (one required)			
042	**IMPORTER**	X'0004'	Key can be used to unwrap an external key brought to this local node, wrap a generated key to be brought to this local node, or unwrap an input key in the Key_Translate2 verb.
KEK control (one or more, optional). Key-usage field 1, high-order byte. All keywords in the group are defaults unless one or more keywords in the group are specified.			

Table 63. Key Token Build2 rule array keywords for AES IMPORTER keys (continued)

Token offset	Rule-array keyword	Offset value	Meaning
045	**GEN-IMEX**	B'xxx1 xxxx'	Key can be used to wrap the first key that is generated by the CSNBKGN2 verb as part of an IMEX key pair.
	GEN-IMIM	B'xxxx 1xxx'	Key can be used to wrap the first or second key that is generated by the CSNBKGN2 verb as part of an IMIM key pair.
	GEN-OPIM	B'xx1x xxxx'	Key can be used to wrap the second key that is generated by the CSNBKGN2 verb as part of an OPIM key pair.
	GEN-PUB	B'xxxx x1xx'	Key can be used to wrap the private key (to be used at the local node) generated by the CSNDPKG verb as part of an ECC public-private key pair.
	IMPORT	B'1xxx xxxx'	Key can be used to unwrap a key brought to this local node. Symmetric_Key_Import.
	TRANSLAT	B'x1xx xxxx'	Key can be used to unwrap an input key in the Key_Translate2 verb. Key_Translate2.
User-defined extension (UDX) control (one or more, optional). Key-usage field 1, low-order byte. No keywords in the group are defaults.			
046	**UDX-ONLY**	B'xxxx 1xxx'	Key can only be used in UDXs.
	UDX-100	B'xxxx x1uu'	Leftmost user-defined UDX bit is set on.
	UDX-010	B'xxxx xu1u'	Middle user-defined UDX bit is set on.
	UDX-001	B'xxxx xuu1'	Rightmost user-defined UDX bit is set on.
TR-31 wrap control (one, optional). Key-usage field 2, high-order byte.			
047	**WR-TR31**	B'1xxx xxxx'	Key can unwrap a TR-31 key. Defined for future use. Currently ignored.
Raw key wrap control (one, optional). Key-usage field 2, low-order byte.			
048	**KEK-RAW**	B'xxxx xxx1'	Key can unwrap a raw key. Defined for future use. Currently ignored.
Algorithm wrap control (one or more, optional). Key-usage field 3, high-order byte. Keywords **WR-AES**, **WR-DES**, and **WR-HMAC** are defaults unless one or more keywords in the group are specified.			
000	**WR-AES**	B'x1xx xxxx'	Key can unwrap AES keys.
	WR-DES	B'1xxx xxxx'	Key can unwrap DES keys.
	WR-ECC	B'xxxx 1xxx'	Key can unwrap ECC keys.
	WR-HMAC	B'xx1x xxxx'	Key can unwrap HMAC keys.
	WR-RSA	B'xxx1 xxxx'	Key can unwrap RSA keys.
Class wrap control (one or more, optional). Key-usage field 4, high-order byte. Keywords **WR-CARD**, **WR-DATA**, **WR-KEK**, **WR-PIN**, and **WRDERIVE** are defaults unless one or more keywords in the group are specified.			
051	**WR-CARD**	B'xxxx 1xxx'	Key can unwrap card class keys.
	WR-CVAR (Release 4.4 or later)	B'xxxx x1xx'	Key can unwrap cryptovariable class keys.
	WR-DATA	B'1xxx xxxx'	Key can unwrap data class keys.
	WR-KEK	B'x1xx xxxx'	Key can unwrap KEK class keys.
	WR-PIN	B'xx1x xxxx'	Key can unwrap PIN class keys.
	WRDERIVE	B'xxx1 xxxx'	Key can wrap derivation class keys.
Symmetric-key export control (one, optional). Key-management field 1, high-order byte.			

Table 63. Key Token Build2 rule array keywords for AES IMPORTER keys (continued)

Token offset	Rule-array keyword	Offset value	Meaning
054	**NOEX-SYM**	B'0xxx xxxx'	Prohibit export using symmetric key.
	XPRT-SYM	B'1xxx xxxx'	Allow export using symmetric key. This is the default.
Unauthenticated asymmetric-key export control (one, optional). Key-management field 1, high-order byte.			
054	**NOEXUASY**	B'x0xx xxxx'	Prohibit export using an unauthenticated asymmetric key (not a trusted block).
	XPRTUASY	B'x1xx xxxx'	Allow export using an unauthenticated asymmetric key (not a trusted block). This is the default.
Authenticated asymmetric-key export control (one, optional). Key-management field 1, high-order byte.			
054	**NOEXAASY**	B'xx0x xxxx'	Prohibit export using an authenticated asymmetric key (trusted block).
	XPRTAASY	B'xx1x xxxx'	Allow export using an authenticated asymmetric key (trusted block). This is the default.
RAW-key export control (one, optional). Key-management field 1, high-order byte.			
054	**NOEX-RAW**	B'xxx0 xxxx'	Prohibit export using raw key. Defined for future use. Currently ignored. This is the default.
	XPRT-RAW	B'xxx1 xxxx'	Allow export using raw key. Defined for future use. Currently ignored.
DES-key export control (one, optional). Key-management field 1, low-order byte.			
055	**NOEX-DES**	B'1xxx xxxx'	Prohibit export using a DES key.
	XPRT-DES	B'0xxx xxxx	Allow export using a DES key. This is the default.
AES-key export control (one, optional). Key-management field 1, low-order byte.			
055	**NOEX-AES**	B'x1xx xxxx'	Prohibit export using an AES key.
	XPRT-AES	B'x0xx xxxx'	Allow export using an AES key.
RSA-key export control (one, optional). Key-management field 1, low-order byte.			
055	**NOEX-RSA**	B'xxxx 1xxx'	Prohibit export using an RSA key.
	XPRT-RSA	B'xxxx 0xxx'	Allow export using an RSA key. This is the default.

Figure 9 on page 238 shows all the valid keyword combinations and their defaults for AES key type MAC. For a description of these keywords, refer to Table 64 on page 238.

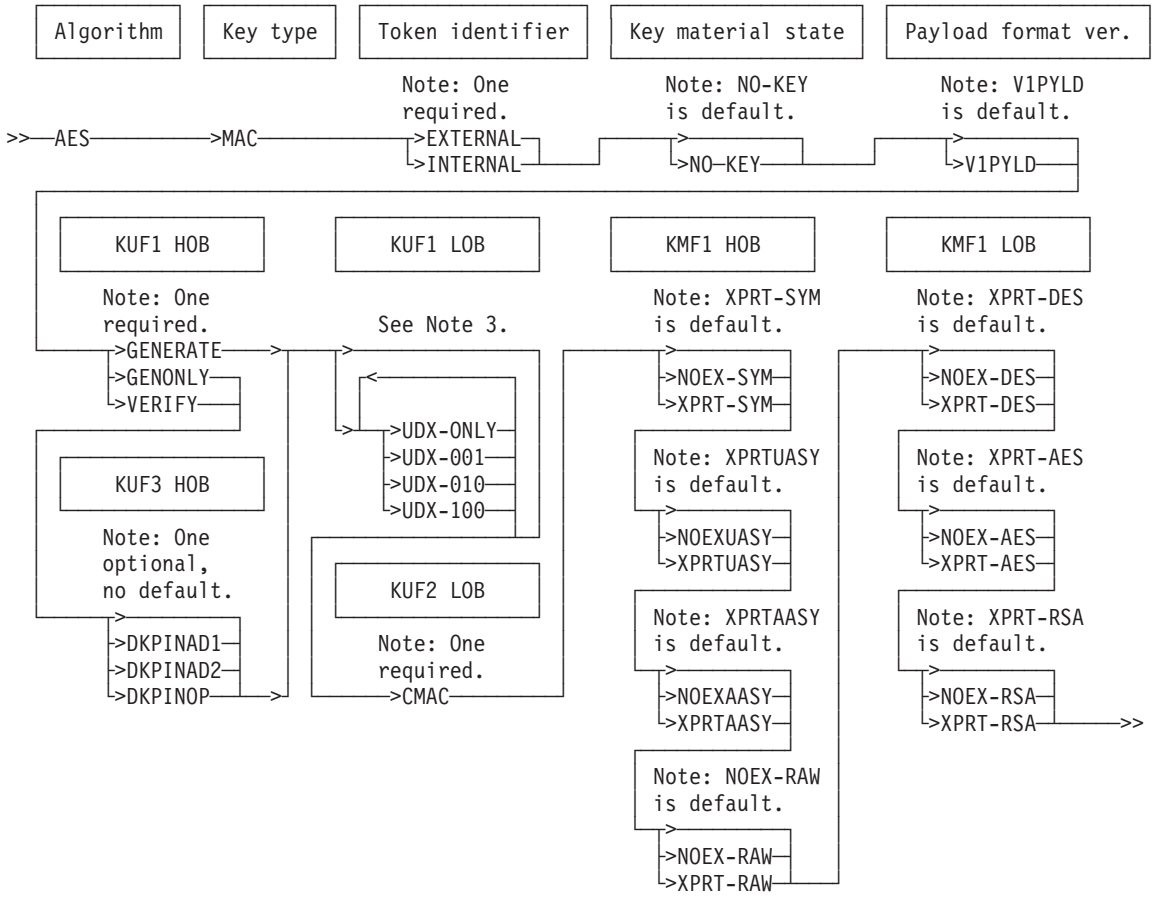


Figure 9. Key Token Build2 keyword combinations for AES MAC keys

Note:

1. Each key-usage field (KUF) and key-management field (KMF) of a version X'05' variable-length symmetric key-token consists of two bytes: a high-order byte (HOB) and a low-order byte (LOB).
2. NOEX-RAW and XPRT-RAW are defined for future use and their meanings are currently undefined. To avoid this export restriction in the future when the meaning is defined, specify XPRT-RAW.
3. Choose any number of keywords in this group. No keywords in this group are defaults.

Table 64. Key Token Build2 rule array keywords for AES MAC keys

Token offset	Rule-array keyword	Offset value	Meaning
Key-token header section			
Token identifier (one required).			
000	**EXTERNAL**	X'02'	Build a key token that is not to be used locally.
	INTERNAL	X'01'	Build a key token that is to be used locally.
Wrapping-information section			
Key status (one, optional).			
008	**NO-KEY**	X'00'	Build a key token that does not contain a key value. This is the default.
Payload format version (one, optional). Identifies format of the payload.			

Table 64. Key Token Build2 rule array keywords for AES MAC keys (continued)

Token offset	Rule-array keyword	Offset value	Meaning
028	**V1PYLD**	X'01'	Build the key token with a version 1 payload format. This format has a fixed length and the key length cannot be inferred by the size of the payload. An obscured key length is considered more secure. This is the default.
Associated data section			
Algorithm type (one required).			
041	**AES**	X'02'	Key can be used for AES algorithm.
Key type (one required)			
042	**MAC**	X'0002'	Key can be used for generation and verification of message authentication codes.
MAC operation (one required). Key-usage field 1, high-order byte.			
045	**GENERATE**	B'11xx xxxx'	Key can be used for generate; key can be used for verify. Not valid with keywords **DKPINOP**, **DKPINAD1**, and **DKPINAD2**. MAC_Generate2 and MAC_Verify2.
	GENONLY	B'10xx xxxx'	Key can be used for generate; key cannot be used for verify.
	VERIFY	B'01xx xxxx'	Key cannot be used for generate; key can be used for verify. MAC_Verify2.
User-defined extension (UDX) control (one or more, optional). Key-usage field 1, low-order byte. No keywords in the group are defaults.			
046	**UDX-ONLY**	B'xxxx 1xxx'	Key can only be used in UDXs.
	UDX-100	B'xxxx x1uu'	Leftmost user-defined UDX bit is set on.
	UDX-010	B'xxxx xu1u'	Middle user-defined UDX bit is set on.
	UDX-001	B'xxxx xuu1'	Rightmost user-defined UDX bit is set on.
MAC mode (one required). Key-usage field 2, high-order byte.			
047	**CMAC**	X'01'	Key can be used for block cipher-based MAC algorithm, called CMAC (NIST SP 800-38B).
Common control (one, optional). Key-usage field 3, high-order byte. Use of a common control keyword causes key-usage field 3, low-order byte (field format identifier at token offset 050) to be set to X'01' (DK enabled).			
049	**DKPINOP**	X'01'	PIN_OP
	DKPINAD1	X'03'	PIN_AD1
	DKPINAD2	X'04'	PIN_AD2
Symmetric-key export control (one, optional). Key-management field 1, high-order byte.			
052 if DK enabled, else 050	**NOEX-SYM**	B'0xxx xxxx'	Prohibit export using symmetric key.
	XPRT-SYM	B'1xxx xxxx'	Allow export using symmetric key. This is the default.
Unauthenticated asymmetric-key export control (one, optional). Key-management field 1, high-order byte.			
052 if DK enabled, else 050	**NOEXUASY**	B'x0xx xxxx'	Prohibit export using an unauthenticated asymmetric key (not a trusted block).
	XPRTUASY	B'x1xx xxxx'	Allow export using an unauthenticated asymmetric key (not a trusted block). This is the default.
Authenticated asymmetric-key export control (one, optional). Key-management field 1, high-order byte.			

Table 64. Key Token Build2 rule array keywords for AES MAC keys (continued)

Token offset	Rule-array keyword	Offset value	Meaning
052 if DK enabled, else 050	**NOEXAASY**	B'xx0x xxxx'	Prohibit export using an authenticated asymmetric key (trusted block).
	XPRTAASY	B'xx1x xxxx'	Allow export using an authenticated asymmetric key (trusted block). This is the default.
RAW-key export control (one, optional). Key-management field 1, high-order byte.			
052 if DK enabled, else 050	**NOEX-RAW**	B'xxx0 xxxx'	Prohibit export using raw key. Defined for future use. Currently ignored. This is the default.
	XPRT-RAW	B'xxx1 xxxx'	Allow export using raw key. Defined for future use. Currently ignored.
DES-key export control (one, optional). Key-management field 1, low-order byte.			
053 if DK enabled, else 051	**NOEX-DES**	B'1xxx xxxx'	Prohibit export using a DES key.
	XPRT-DES	B'0xxx xxxx	Allow export using a DES key. This is the default.
AES-key export control (one, optional). Key-management field 1, low-order byte.			
053 if DK enabled, else 051	**NOEX-AES**	B'x1xx xxxx'	Prohibit export using an AES key.
	XPRT-AES	B'x0xx xxxx'	Allow export using an AES key.
RSA-key export control (one, optional). Key-management field 1, low-order byte.			
053 if DK enabled, else 051	**NOEX-RSA**	B'xxxx 1xxx'	Prohibit export using an RSA key.
	XPRT-RSA	B'xxxx 0xxx'	Allow export using an RSA key. This is the default.

Figure 10 on page 241 shows all the valid keyword combinations and their defaults for HMAC key type MAC. For a description of these keywords, refer to Table 65 on page 241.

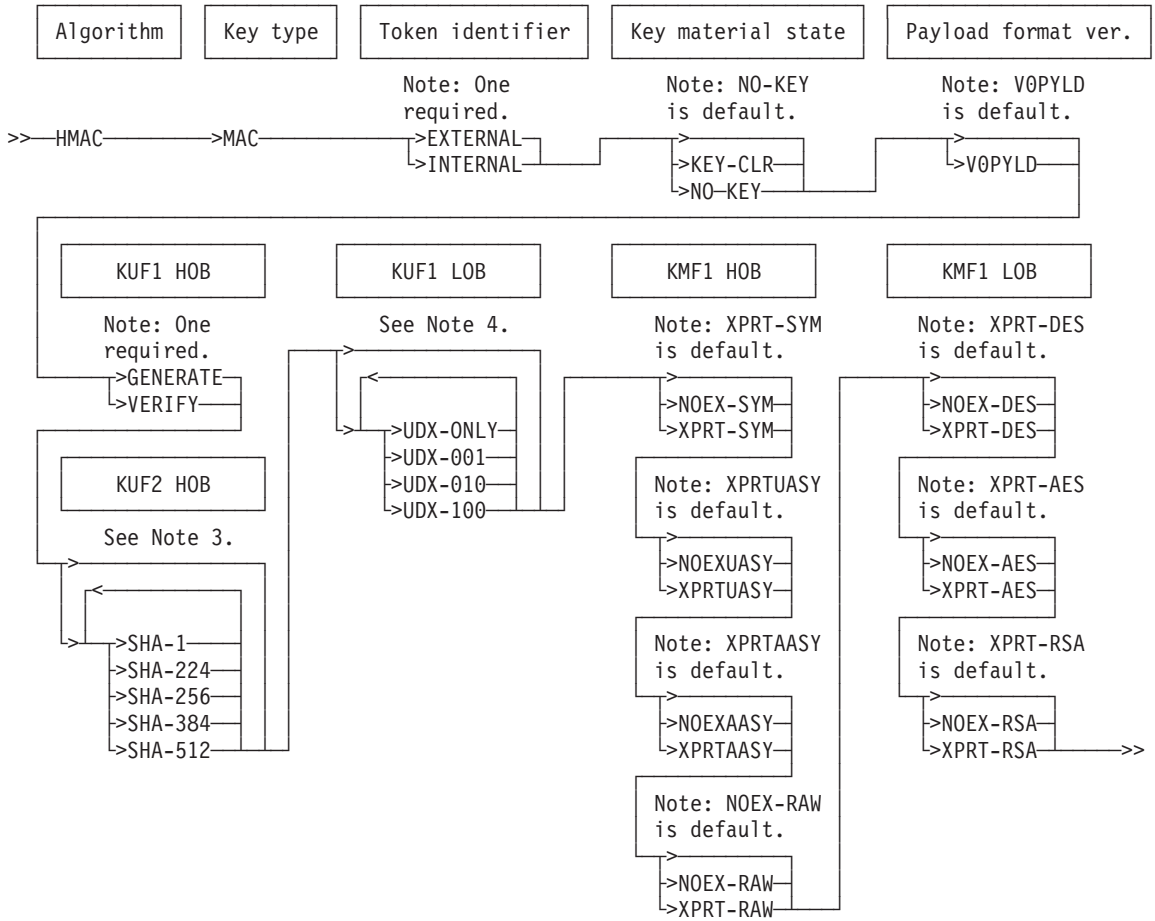


Figure 10. Key Token Build2 keyword combinations for HMAC MAC keys

Note:

1. Each key-usage field (KUF) and key-management field (KMF) of a version X'05' variable-length symmetric key-token consists of two bytes: a high-order byte (HOB) and a low-order byte (LOB).
2. NOEX-RAW and XPRT-RAW are defined for future use and their meanings are currently undefined. To avoid this export restriction in the future when the meaning is defined, specify XPRT-RAW.
3. All keywords in this group are defaults unless one or more keywords in this group are specified.
4. Choose any number of keywords in this group. No keywords in this group are defaults.

Table 65. Key Token Build2 rule array keywords for HMAC MAC keys

Token offset	Rule-array keyword	Offset value	Meaning
Key-token header section			
Token identifier (one required).			
000	**EXTERNAL**	X'02'	Build a key token that is not to be used locally.
	INTERNAL	X'01'	Build a key token that is to be used locally.
Wrapping-information section			
Key status (one, optional)			

Table 65. Key Token Build2 rule array keywords for HMAC MAC keys (continued)

Token offset	Rule-array keyword	Offset value	Meaning
008	**KEY-CLR**	X'01'	Build a key token that contains a clear key.
	NO-KEY	X'00'	Build a key token that does not contain a key value. This is the default.
Payload format version (one, optional). Identifies format of the payload. Release 4.4 or later; otherwise, undefined.			
028	**V0PYLD**	X'00'	Build a key token with a version 0 payload format. This format has a variable length and the key length can be inferred from the size of the payload. This is the default. This format is compatible with all releases.
Associated data section			
Algorithm type (one required).			
041	**HMAC**	X'03'	Key can be used for HMAC algorithm.
Key type (one required)			
042	**MAC**	X'0002'	Key can be used for generation or verification of message authentication codes.
MAC operation (one required). Key-usage field 1, high-order byte.			
045	**GENERATE**	B'11xx xxxx'	Key can be used for generate; key can be used for verify. HMAC_Generate and HMAC_Verify.
	VERIFY	B'01xx xxxx'	Key cannot be used for generate; key can be used for verify. HMAC_Verify.
User-defined extension (UDX) control (one or more, optional). Key-usage field 1, low-order byte. No keywords in the group are defaults.			
046	**UDX-ONLY**	B'xxxx 1xxx'	Key can only be used in UDXs.
	UDX-100	B'xxxx x1uu'	Leftmost user-defined UDX bit is set on.
	UDX-010	B'xxxx xu1u'	Middle user-defined UDX bit is set on.
	UDX-001	B'xxxx xuu1'	Rightmost user-defined UDX bit is set on.
Hash method (one, optional). Key-usage field 2, high-order byte. All keywords in the group are defaults unless one or more keywords in the group are specified.			
047	**SHA-1**	B'1xxx xxxx'	SHA-1 hash method is allowed for the key.
	SHA-224	B'x1xx xxxx'	SHA-224 hash method is allowed for the key.
	SHA-256	B'xx1x xxxx'	SHA-256 hash method is allowed for the key.
	SHA-384	B'xxx1 xxxx'	SHA-384 hash method is allowed for the key.
	SHA-512	B'xxxx 1xxx'	SHA-512 hash method is allowed for the key.
Symmetric-key export control (one, optional). Key-management field 1, high-order byte.			
050	**NOEX-SYM**	B'0xxx xxxx'	Prohibit export using symmetric key.
	XPRT-SYM	B'1xxx xxxx'	Allow export using symmetric key. This is the default.
Unauthenticated asymmetric-key export control (one, optional). Key-management field 1, high-order byte.			
050	**NOEXUASY**	B'x0xx xxxx'	Prohibit export using an unauthenticated asymmetric key (not a trusted block).
	XPRTUASY	B'x1xx xxxx'	Allow export using an unauthenticated asymmetric key (not a trusted block). This is the default.
Authenticated asymmetric-key export control (one, optional). Key-management field 1, high-order byte.			

Table 65. Key Token Build2 rule array keywords for HMAC MAC keys (continued)

Token offset	Rule-array keyword	Offset value	Meaning
050	**NOEXAASY**	B'xx0x xxxx'	Prohibit export using an authenticated asymmetric key (trusted block).
	XPRTAASY	B'xx1x xxxx'	Allow export using an authenticated asymmetric key (trusted block). This is the default.
RAW-key export control (one, optional). Key-management field 1, high-order byte.			
050	**NOEX-RAW**	B'xxx0 xxxx'	Prohibit export using raw key. Defined for future use. Currently ignored. This is the default.
	XPRT-RAW	B'xxx1 xxxx'	Allow export using raw key. Defined for future use. Currently ignored.
DES-key export control (one, optional). Key-management field 1, low-order byte.			
051	**NOEX-DES**	B'1xxx xxxx'	Prohibit export using a DES key.
	XPRT-DES	B'0xxx xxxx	Allow export using a DES key. This is the default.
AES-key export control (one, optional). Key-management field 1, low-order byte.			
051	**NOEX-AES**	B'x1xx xxxx'	Prohibit export using an AES key.
	XPRT-AES	B'x0xx xxxx'	Allow export using an AES key.
RSA-key export control (one, optional). Key-management field 1, low-order byte.			
051	**NOEX-RSA**	B'xxxx 1xxx'	Prohibit export using an RSA key.
	XPRT-RSA	B'xxxx 0xxx'	Allow export using an RSA key. This is the default.

Figure 11 on page 244 shows all the valid keyword combinations and their defaults for AES key type PINCALC. For a description of these keywords, refer to Table 66 on page 244.

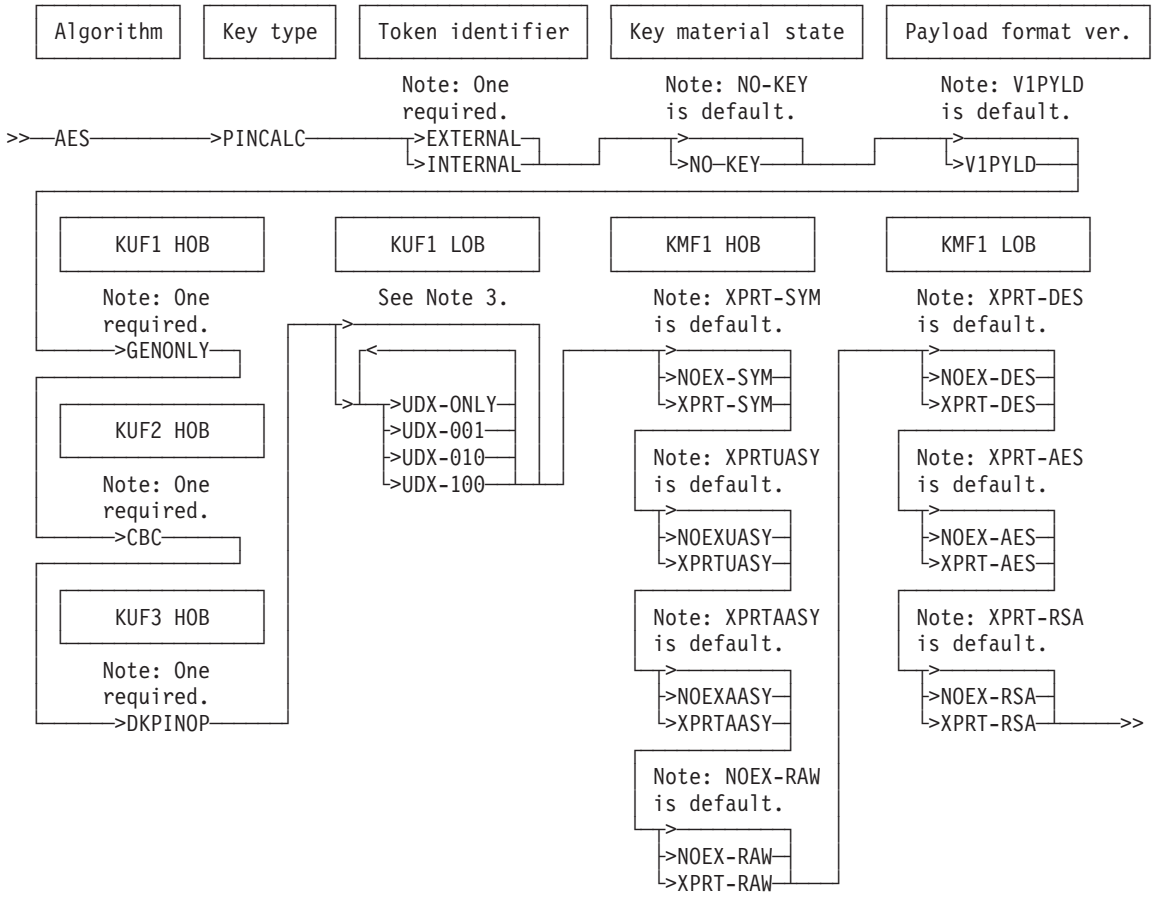


Figure 11. Key Token Build2 keyword combinations for AES PINCALC keys

Note:

1. Each key-usage field (KUF) and key-management field (KMF) of a version X'05' variable-length symmetric key-token consists of two bytes: a high-order byte (HOB) and a low-order byte (LOB).
2. NOEX-RAW and XPRT-RAW are defined for future use and their meanings are currently undefined. To avoid this export restriction in the future when the meaning is defined, specify XPRT-RAW.
3. Choose any number of keywords in this group. No keywords in the group are defaults.

Table 66. Key Token Build2 rule array keywords for AES PINCALC keys

Token offset	Rule-array keyword	Offset value	Meaning
Key-token header section			
Token identifier (one required)			
000	**EXTERNAL**	X'02'	Build a key token that is not to be used locally.
	INTERNAL	X'01'	Build a key token that is to be used locally.
Wrapping-information section			
Key status (one, optional)			
008	**NO-KEY**	X'00'	Build a key token that does not contain a key value. This is the default.
Payload format version (one, optional). Identifies format of the payload.			

Table 66. Key Token Build2 rule array keywords for AES PINCALC keys (continued)

Token offset	Rule-array keyword	Offset value	Meaning
028	**V1PYLD**	X'01'	Build the key token with a version 1 payload format. This format has a fixed length and the key length cannot be inferred by the size of the payload. An obscured key length is considered more secure. This is the default.
Associated data section			
Algorithm type (one required).			
041	**AES**	X'02'	Key can be used for AES algorithm.
Key type (one required)			
042	**PINCALC**	X'0006'	Key can be used for generation and verification of message authentication codes.
MAC operation (one required). Key-usage field 1, high-order byte.			
045	**GENONLY**	B'1xxx xxxx'	Key can only be used for generate.
User-defined extension (UDX) control (one or more, optional). Key-usage field 1, low-order byte. No keywords in the group are defaults.			
046	**UDX-ONLY**	B'xxxx 1xxx'	Key can only be used in UDXs.
	UDX-100	B'xxxx x1uu'	Leftmost user-defined UDX bit is set on.
	UDX-010	B'xxxx xu1u'	Middle user-defined UDX bit is set on.
	UDX-001	B'xxxx xuu1'	Rightmost user-defined UDX bit is set on.
Encryption mode (one required). Key-usage field 2, high-order byte.			
047	**CBC**	X'00'	Key can be used for Cipher Block Chaining.
Common control (one required). Key-usage field 3, high-order byte. Use of a common control keyword causes key-usage field 3, low-order byte (field format identifier at token offset 050) to be set to X'01' (DK enabled).			
049	**DKPINOP**	X'01'	PIN_OP
Symmetric-key export control (one, optional). Key-management field 1, high-order byte.			
052	**NOEX-SYM**	B'0xxx xxxx'	Prohibit export using symmetric key.
	XPRT-SYM	B'1xxx xxxx'	Allow export using symmetric key. This is the default.
Unauthenticated asymmetric-key export control (one, optional). Key-management field 1, high-order byte.			
052	**NOEXUASY**	B'x0xx xxxx'	Prohibit export using an unauthenticated asymmetric key (not a trusted block).
	XPRTUASY	B'x1xx xxxx'	Allow export using an unauthenticated asymmetric key (not a trusted block). This is the default.
Authenticated asymmetric-key export control (one, optional). Key-management field 1, high-order byte.			
052	**NOEXAASY**	B'xx0x xxxx'	Prohibit export using an authenticated asymmetric key (trusted block).
	XPRTAASY	B'xx1x xxxx'	Allow export using an authenticated asymmetric key (trusted block). This is the default.
RAW-key export control (one, optional). Key-management field 1, high-order byte.			
052	**NOEX-RAW**	B'xxx0 xxxx'	Prohibit export using raw key. Defined for future use. Currently ignored. This is the default.
	XPRT-RAW	B'xxx1 xxxx'	Allow export using raw key. Defined for future use. Currently ignored.
DES-key export control (one, optional). Key-management field 1, low-order byte.			

Table 66. Key Token Build2 rule array keywords for AES PINCALC keys (continued)

Token offset	Rule-array keyword	Offset value	Meaning
053	**NOEX-DES**	B'1xxx xxxx'	Prohibit export using a DES key.
	XPRT-DES	B'0xxx xxxx	Allow export using a DES key. This is the default.
AES-key export control (one, optional). Key-management field 1, low-order byte.			
053	**NOEX-AES**	B'x1xx xxxx'	Prohibit export using an AES key.
	XPRT-AES	B'x0xx xxxx'	Allow export using an AES key.
RSA-key export control (one, optional). Key-management field 1, low-order byte.			
053	**NOEX-RSA**	B'xxxx 1xxx'	Prohibit export using an RSA key.
	XPRT-RSA	B'xxxx 0xxx'	Allow export using an RSA key. This is the default.

Figure 12 shows all the valid keyword combinations and their defaults for AES key type PINPROT. For a description of these keywords, refer to Table 67 on page 247.

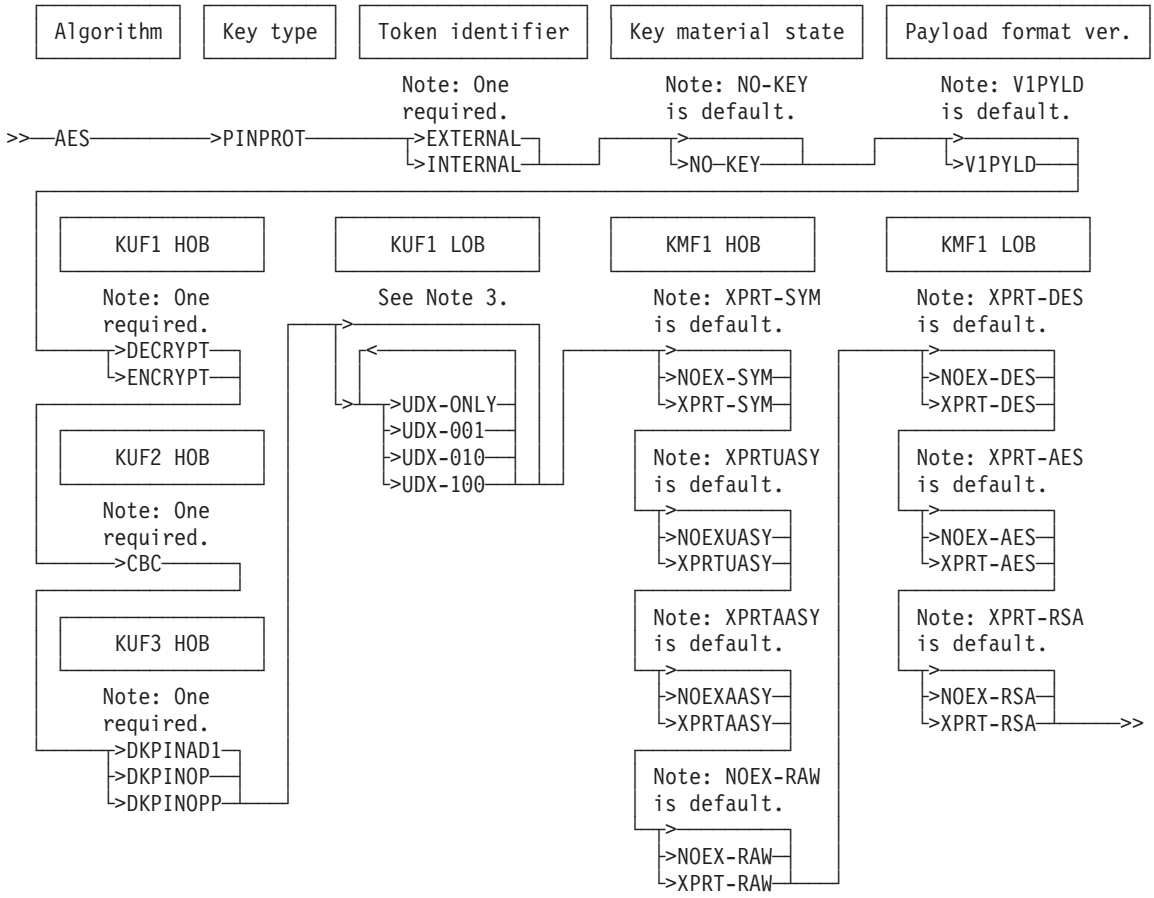


Figure 12. Key Token Build2 keyword combinations for AES PINPROT keys

Note:

1. Each key-usage field (KUF) and key-management field (KMF) of a version X'05' variable-length symmetric key-token consists of two bytes: a high-order byte (HOB) and a low-order byte (LOB).

2. NOEX-RAW and XPRT-RAWare defined for future use and their meanings are currently undefined. To avoid this export restriction in the future when the meaning is defined, specify XPRT-RAW.
3. Choose any number of keywords in this group. No keywords in the group are defaults.

Table 67. Key Token Build2 rule array keywords for AES PINPROT keys

Token offset	Rule-array keyword	Offset value	Meaning
Key-token header section			
Token identifier (one required)			
000	**EXTERNAL**	X'02'	Build a key token that is not to be used locally.
	INTERNAL	X'01'	Build a key token that is to be used locally.
Wrapping-information section			
Key status (one, optional)			
008	**NO-KEY**	X'00'	Build a key token that does not contain a key value. This is the default.
Payload format version (one, optional). Identifies format of the payload.			
028	**V1PYLD**	X'01'	Build the key token with a version 1 payload format. This format has a fixed length and the key length cannot be inferred by the size of the payload. An obscured key length is considered more secure. This is the default.
Associated data section			
Algorithm type (one required)			
041	**AES**	X'02'	Key can be used for AES algorithm.
Key type (one required).			
042	**PINPROT**	X'0005'	Key can be used for encrypting PIN blocks.
Encryption operation (one required). Key-usage field 1, high-order byte.			
045	**DECRYPT**	B'01xx xxxx'	Key cannot be used for encryption; key can be used for decryption.
	ENCRYPT	B'10xx xxxx'	Key can be used for encryption; key cannot be used for decryption.
User-defined extension (UDX) control (one or more, optional). Key-usage field 1, low-order byte. No keywords in the group are defaults.			
046	**UDX-ONLY**	B'xxxx 1xxx'	Key can only be used in UDXs.
	UDX-100	B'xxxx x1uu'	Leftmost user-defined UDX bit is set on.
	UDX-010	B'xxxx xu1u'	Middle user-defined UDX bit is set on.
	UDX-001	B'xxxx xuu1'	Rightmost user-defined UDX bit is set on.
Encryption mode (one required). Key-usage field 2, high-order byte.			
047	**CBC**	X'00'	Key can be used for Cipher Block Chaining.
Common control (one required). Key-usage field 3, high-order byte. Use of a common control keyword causes key-usage field 3, low-order byte (field format identifier at token offset 050) to be set to X'01' (DK enabled).			
049	**DKPINOP**	X'01'	PIN_OP
	DKPINOPP	X'02'	PIN_OPP
	DKPINAD1	X'03'	PIN_AD1
Symmetric-key export control (one, optional). Key-management field 1, high-order byte.			

Table 67. Key Token Build2 rule array keywords for AES PINPROT keys (continued)

Token offset	Rule-array keyword	Offset value	Meaning
052	**NOEX-SYM**	B'0xxx xxxx'	Prohibit export using symmetric key.
	XPRT-SYM	B'1xxx xxxx'	Allow export using symmetric key. This is the default.
Unauthenticated asymmetric-key export control (one, optional). Key-management field 1, high-order byte.			
052	**NOEXUASY**	B'x0xx xxxx'	Prohibit export using an unauthenticated asymmetric key (not a trusted block).
	XPRTUASY	B'x1xx xxxx'	Allow export using an unauthenticated asymmetric key (not a trusted block). This is the default.
Authenticated asymmetric-key export control (one, optional). Key-management field 1, high-order byte.			
052	**NOEXAASY**	B'xx0x xxxx'	Prohibit export using an authenticated asymmetric key (trusted block).
	XPRTAASY	B'xx1x xxxx'	Allow export using an authenticated asymmetric key (trusted block). This is the default.
RAW-key export control (one, optional). Key-management field 1, high-order byte.			
052	**NOEX-RAW**	B'xxx0 xxxx'	Prohibit export using raw key. Defined for future use. Currently ignored. This is the default.
	XPRT-RAW	B'xxx1 xxxx'	Allow export using raw key. Defined for future use. Currently ignored.
DES-key export control (one, optional). Key-management field 1, low-order byte.			
053	**NOEX-DES**	B'1xxx xxxx'	Prohibit export using a DES key.
	XPRT-DES	B'0xxx xxxx	Allow export using a DES key. This is the default.
AES-key export control (one, optional). Key-management field 1, low-order byte.			
053	**NOEX-AES**	B'x1xx xxxx'	Prohibit export using an AES key.
	XPRT-AES	B'x0xx xxxx'	Allow export using an AES key.
RSA-key export control (one, optional). Key-management field 1, low-order byte.			
053	**NOEX-RSA**	B'xxxx 1xxx'	Prohibit export using an RSA key.
	XPRT-RSA	B'xxxx 0xxx'	Allow export using an RSA key. This is the default.

Figure 13 on page 249 shows all the valid keyword combinations and their defaults for AES key type PINPRW. For a description of these keywords, refer to Table 68 on page 249.

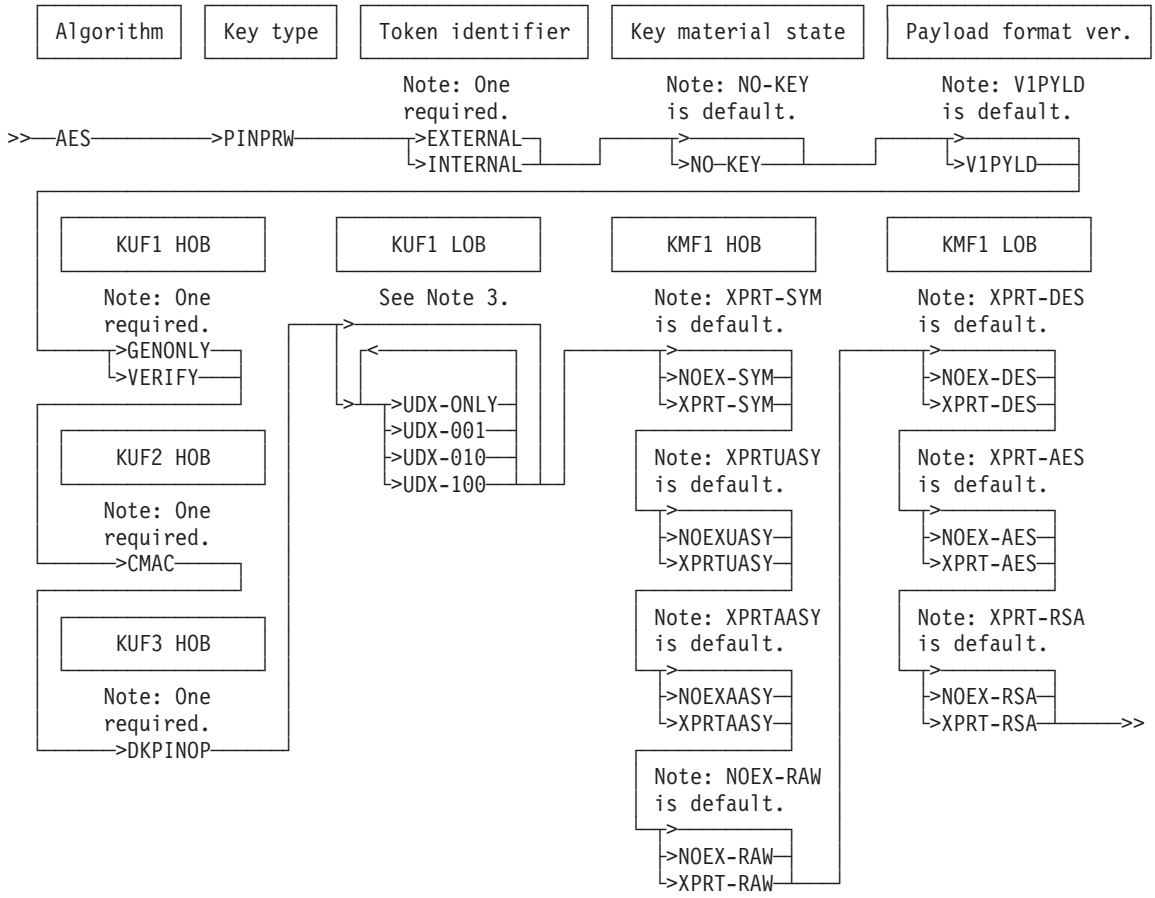


Figure 13. Key Token Build2 keyword combinations for AES PINPRW keys

Note:

1. Each key-usage field (KUF) and key-management field (KMF) of a version X'05' variable-length symmetric key-token consists of two bytes: a high-order byte (HOB) and a low-order byte (LOB).
2. NOEX-RAW and XPRT-RAWare defined for future use and their meanings are currently undefined. To avoid this export restriction in the future when the meaning is defined, specify XPRT-RAW.
3. Choose any number of keywords in this group. No keywords in the group are defaults.

Table 68. Key Token Build2 PINPRW related key words

Token offset	Rule-array keyword	Offset value	Meaning
Key-token header section			
Token identifier (one required).			
000	**EXTERNAL**	X'02'	Build a key token that is not to be used locally.
	INTERNAL	X'01'	Build a key token that is to be used locally.
Wrapping-information section			
Key status (one, optional).			
008	**NO-KEY**	X'00'	Build a key token that does not contain a key value. This is the default.
Payload format version (one, optional). Identifies format of the payload.			

Table 68. Key Token Build2 PINPRW related key words (continued)

Token offset	Rule-array keyword	Offset value	Meaning
028	**V1PYLD**	X'01'	Build the key token with a version 1 payload format. This format has a fixed length and the key length cannot be inferred by the size of the payload. An obscured key length is considered more secure. This is the default.
Associated data section			
Algorithm type (one required).			
041	**AES**	X'02'	Key can be used for AES algorithm.
Key type (one required)			
042	**PINPRW**	X'0007'	Key can be used for generation and verification of message authentication codes.
MAC operation (one required). Key-usage field 1, high-order byte.			
045	**GENONLY**	B'10xx xxxx'	Key can be used for generate; key cannot be used for verify.
	VERIFY	B'01xx xxxx	Key cannot be used for generate; key can be used for verify.
User-defined extension (UDX) control (one or more, optional). Key-usage field 1, low-order byte. No keywords in the group are defaults.			
046	**UDX-ONLY**	B'xxxx 1xxx'	Key can only be used in UDXs.
	UDX-100	B'xxxx x1uu'	Leftmost user-defined UDX bit is set on.
	UDX-010	B'xxxx xu1u'	Middle user-defined UDX bit is set on.
	UDX-001	B'xxxx xuu1'	Rightmost user-defined UDX bit is set on.
MAC mode (one required). Key-usage field 2, high-order byte.			
047	**CMAC**	X'01'	Key can be used for block cipher-based MAC algorithm, called CMAC (NIST SP 800-38B).
Common control (one, required). Key-usage field 3, high-order byte. Use of a common control keyword causes key-usage field 3, low-order byte (field format identifier at token offset 050) to be set to X'01' (DK enabled).			
049	**DKPINOP**	X'01'	PIN_OP
Symmetric-key export control (one, optional). Key-management field 1, high-order byte.			
052	**NOEX-SYM**	B'0xxx xxxx'	Prohibit export using symmetric key.
	XPRT-SYM	B'1xxx xxxx'	Allow export using symmetric key. This is the default.
Unauthenticated asymmetric-key export control (one, optional). Key-management field 1, high-order byte.			
052	**NOEXUASY**	B'x0xx xxxx'	Prohibit export using an unauthenticated asymmetric key (not a trusted block).
	XPRTUASY	B'x1xx xxxx'	Allow export using an unauthenticated asymmetric key (not a trusted block). This is the default.
Authenticated asymmetric-key export control (one, optional). Key-management field 1, high-order byte.			
052	**NOEXAASY**	B'xx0x xxxx'	Prohibit export using an authenticated asymmetric key (trusted block).
	XPRTAASY	B'xx1x xxxx'	Allow export using an authenticated asymmetric key (trusted block). This is the default.
RAW-key export control (one, optional). Key-management field 1, high-order byte.			
052	**NOEX-RAW**	B'xxx0 xxxx'	Prohibit export using raw key. Defined for future use. Currently ignored. This is the default.
	XPRT-RAW	B'xxx1 xxxx'	Allow export using raw key. Defined for future use. Currently ignored.

Table 68. Key Token Build2 PINPRW related key words (continued)

Token offset	Rule-array keyword	Offset value	Meaning
DES-key export control (one, optional). Key-management field 1, low-order byte.			
053	**NOEX-DES**	B'1xxx xxxx'	Prohibit export using a DES key.
	XPRT-DES	B'0xxx xxxx	Allow export using a DES key. This is the default.
AES-key export control (one, optional). Key-management field 1, low-order byte.			
053	**NOEX-AES**	B'x1xx xxxx'	Prohibit export using an AES key.
	XPRT-AES	B'x0xx xxxx'	Allow export using an AES key.
RSA-key export control (one, optional). Key-management field 1, low-order byte.			
053	**NOEX-RSA**	B'xxxx 1xxx'	Prohibit export using an RSA key.
	XPRT-RSA	B'xxxx 0xxx'	Allow export using an RSA key. This is the default.

Figure 14 shows all the valid keyword combinations and their defaults for AES key type SECMSG.

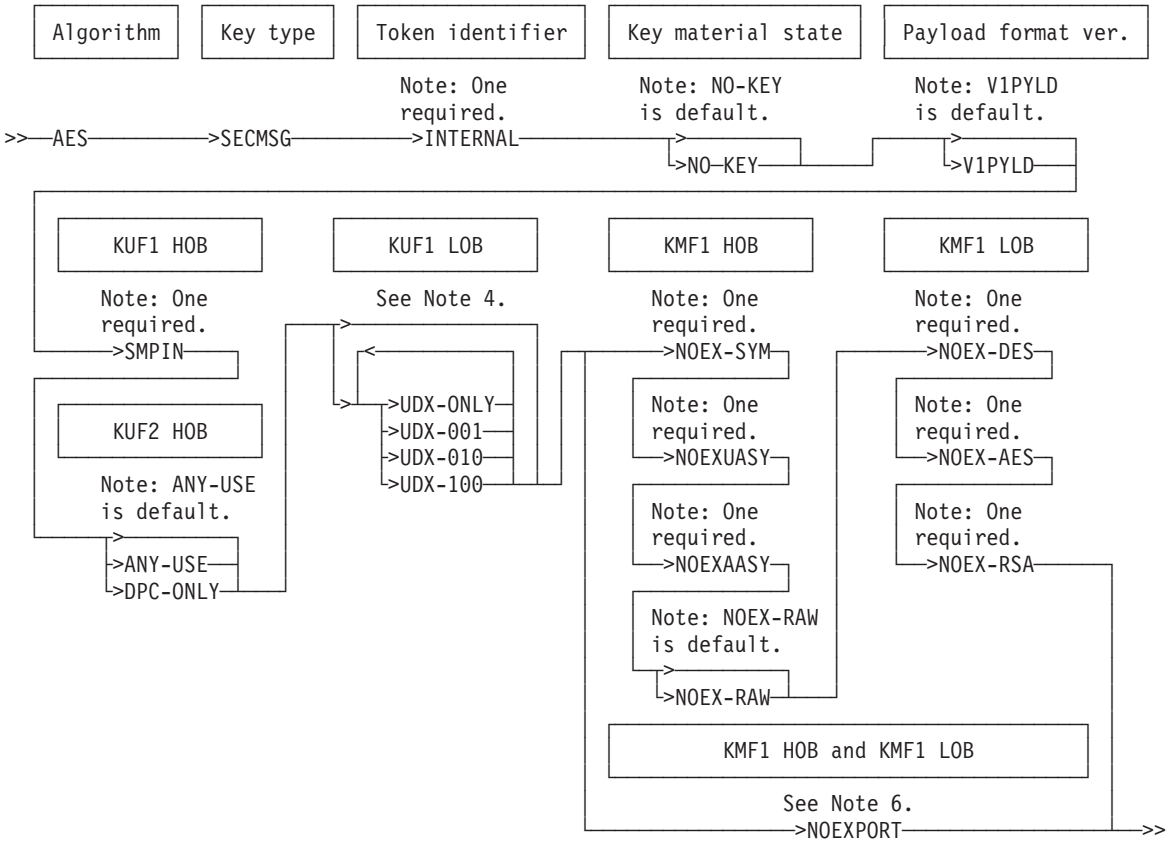


Figure 14. Key Token Build2 keyword combinations for AES SECMSG keys

Note:

1. AES SECMSG is Release 4.4 or later.
2. An AES SECMSG key is always derived. The derived key is the result of a key derivation function (KDF) applied to a fixed diversified key generating key (DKYGENKY) and derivation data. The final derived key is used as a session key and is typically used to encipher and decipher PIN information between

devices. An AES SECMSG key can only be wrapped by an AES master key and cannot be stored in an external key-token.

3. Each key-usage field (KUF) and key-management field (KMF) of a version X'05' variable-length symmetric key-token consists of two bytes: a high-order byte (HOB) and a low-order byte (LOB).
4. Choose any number of keywords in this group. No keywords in this group are defaults.
5. **NOEX-RAW** is defined for future use.
6. There is no default. Specifying **NOEXPORT** is equivalent to specifying all of the export control keywords (**NOEX-SYM**, **NOEXUASY**, **NOEXAASY**, **NOEX-RAW,NOEX-DES**, **NOEX-AES**, and **NOEX-RSA**). Do not specify any export control keywords together with **NOEXPORT**. If **NOEXPORT** is not specified, **NOEX-SYM**, **NOEXUASY**, **NOEXAASY**, **NOEX-DES**, **NOEX-AES**, and **NOEX-RSA** all must be specified, and **NOEX-RAW** is optional.

Table 69. Key Token Build2 SECMSG related key words

Token offset	Rule-array keyword	Offset value	Meaning
Key-token header section			
Token identifier (one required).			
000	**INTERNAL**	X'01'	Build a key token that is to be used locally.
Wrapping-information section			
Key status (one, optional).			
008	**NO-KEY**	X'00'	Build a key token that does not contain a key value. This is the default.
Payload format version (one, optional). Identifies format of the payload.			
028	**V1PYLD**	X'01'	Build the key token with a version 1 payload format. This format has a fixed length and the key length cannot be inferred by the size of the payload. An obscured key length is considered more secure. This is the default.
Associated data section			
Algorithm type (one required)			
041	**AES**	X'02'	Key can be used for AES algorithm.
Associated data section			
Key type (one required)			
042	**SECMSG**	X'000A'	Key can be used as an EMV secure messaging key for encrypting PINs or for encrypting keys.
Secure message encryption enablement (one required). Key-usage field 1, high-order byte			
045	**SMPIN**	X'00'	Enable the encryption of PINs in an EMV secure message.
User-defined extension (UDX) control (one or more, optional). Key-usage field 1, low-order byte. No keywords in the group are defaults.			
046	**UDX-ONLY**	B'xxxx 1xxx'	Key can only be used in UDXs.
	UDX-100	B'xxxx x1uu'	Leftmost user-defined UDX bit is set on.
	UDX-010	B'xxxx xu1u'	Middle user-defined UDX bit is set on.
	UDX-001	B'xxxx xuu1'	Rightmost user-defined UDX bit is set on.
Verb restriction (one, optional)			

Table 69. Key Token Build2 SECMSG related key words (continued)

Token offset	Rule-array keyword	Offset value	Meaning
047	**ANY-USE**	X'00'	Any verb (service) can use this key. This is the default.
	DPC-ONLY	X'01'	Only CSNBDPC can use this key.
General export control (one, optional). Equivalent to specifying all export control keywords (**NOEX-SYM**, **NOEXUASY**, **NOEXAASY**, **NOEX-RAW** , **NOEX-DES**, **NOEX-AES**, and **NOEX-RSA**). Not valid with any other export control keyword. There is no default. Key-management field 1, high-order byte and low-order byte.			
050	**NOEXPORT**	B'0000 xxxx'	Prohibits the export of this key in all cases. Equivalent to specifiying **NOEX-SYM**, **NOEXUASY**, **NOEXAASY**, **NOEX-RAW, NOEX-DES**, **NOEX-AES**, and **NOEX-RSA**.
051		B'11xx 1xxx'	
Symmetric-key export control (one required if **NOEXPORT** not specified, otherwise not valid). Key-management field 1, high-order byte.			
050	**NOEX-SYM**	B'0xxx xxxx'	Prohibit export using symmetric key.
Unauthenticated asymmetric-key export control (one required if **NOEXPORT** not specified, otherwise not valid). Key-management field 1, high-order byte.			
050	**NOEXUASY**	B'x0xx xxxx'	Prohibit export using an unauthenticated asymmetric key (not a trusted block).
Authenticated asymmetric-key export control (one required if **NOEXPORT** not specified, otherwise not valid). Key-management field 1, high-order byte.			
050	**NOEXAASY**	B'xx0x xxxx'	Prohibit export using an authenticated asymmetric key (trusted block).
RAW-key export control (one optional if **NOEXPORT** not specified; otherwise not valid). Key-management field 1, high-order byte.			
050	**NOEX-RAW**	B'xxx0 xxxx'	Prohibit export using RAW key. Defined for future use. Currently ignored. This is the default.
DES-key export control (one required if **NOEXPORT** not specified, otherwise not valid). Key-management field 1, low-order byte.			
051	**NOEX-DES**	B'1xxx xxxx'	Prohibit export using a DES key.
AES-key export control (one required if **NOEXPORT** not specified, otherwise not valid). Key-management field 1, low-order byte			
051	**NOEX-AES**	B'x1xx xxxx'	Prohibit export using an AES key.
RSA-key export control (one required if **NOEXPORT** not specified, otherwise not valid). Key-management field 1, low-order byte.			
051	**NOEX-RSA**	B'xxxx 1xxx'	Prohibit export using an RSA key.

Restrictions

The restrictions for CSNBKTB2.

This verb was introduced with CCA 4.1.0.

Required commands

The required commands for CSNBKTB2.

None.

Usage notes

The usage notes for CSNBKTB2.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKTB2J.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBKTB2J(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  hikmNativeInteger rule_array_count,
  byte[]            rule_array,
  hikmNativeInteger clear_key_bit_length,
  byte[]            clear_key_value,
  hikmNativeInteger key_name_length,
  byte[]            key_name,
  hikmNativeInteger user_associated_data_length,
  byte[]            user_associated_data,
  hikmNativeInteger token_data_length,
  byte[]            token_data,
  hikmNativeInteger verb_data_length,
  byte[]            verb_data,
  hikmNativeInteger target_key_token_length,
  byte[]            target_key_token);
```

Key Token Change (CSNBKTC)

Use the Key Token Change verb to re-encipher a DES or AES key from encryption under the old master-key to encryption under the current master-key and to update the keys in internal DES or AES key-tokens.

Note:

1. An application system is responsible for keeping all of its keys in a usable form. When the master key is changed, the CEX*C implementations can use an internal key that is enciphered by either the current or the old master-key. Before the master key is changed a second time, it is important to have a key re-enciphered under the current master-key for continued use of the key. Use the Key Token Change verb to re-encipher such a keys.
2. Previous implementations of IBM CCA products had additional capabilities with this verb such as deleting key records and key tokens in key storage. Also, use of a wild card (*) was supported in those implementations.

Format

The format of CSNBKTC.

```
CSNBKTC(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_identifier)
```

Parameters

The parameters for CSNBKTC.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1, 2, 3, or 4.

rule_array

Direction: Input
Type: String array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* keywords are described in Table 70.

Table 70. Keywords for Key Token Change control information

Keyword	Description
Re-encipherment method (Required)	
RTCMK	Re-enciphers a DES or AES key to the current master-key in an internal key-token in application storage or in key storage. If the supplied key is already enciphered under the current master-key the verb returns a positive response (return code 0, reason code 0). If the supplied key is enciphered under the old master-key, the key is updated to encipherment by the current master-key and the verb returns a positive response (return code 0, reason code 0). Other cases return some form of abnormal response.

Table 70. Keywords for Key Token Change control information (continued)

Keyword	Description
RTNMK	Re-enciphers an internal DES or AES key to the new master-key. A key enciphered under the new master key is not usable. It is expected that the user will use this keyword (**RTNMK**) to take a preparatory step in re-enciphering an external key store that they manage themselves to a new master-key, before the set operation has occurred. Note also that the new master-key register must be full; it must have had the last key part loaded and therefore not be empty or partially full (partially full means that one or more key parts have been loaded but not the last key part). The SET operation makes the new master-key operational, moving it to the current master-key register, and the current master-key is displaced into the old master-key register. When this happens, all the keys that were re-enciphered to the new master-key are now usable, because the new master-key is not 'new' any more, it is 'current'. Because the **RTNMK** keyword is added primarily for support of externally managed key storage (see "Key Storage on z/OS (RTNMK-focused)" on page 405, it is not valid to pass a *key_identifer* when the **RTNMK** keyword is used. Only a full internal key token (encrypted under the current master-key) can be passed for re-encipherment with the **RTNMK** keyword. When a key label is passed along with the **RTNMK** keyword, the error return code 8 with reason code 181 will be returned. For more information, see "Key storage with Linux on z Systems, in contrast to z/OS on z Systems" on page 403.
VALIDATE	Validate an internal key token as described in the **key_identifier** which is enciphered under the current master key. That is, the same processing as RTNMK is applied. However, after a successful checking of the token, no re-enciphering of the token to the new master key takes place. There is just a return code for a successful validation.
REFORMAT	Rewrap the *input_key_token* with the key wrapping method specified. Only the *input_KEK_identifier* will be used. The *output_KEK_identifier* is ignored. This keyword was introduced with CCA 4.1.0.
Algorithm (Optional)	
AES	Specifies that the key token is for an AES key.
DES	Specifies that the key token is for a DES key. This is the default.
Key wrapping method (Optional)	
USECONFG	This is the default. Specifies to wrap the key using the configuration setting for the default wrapping method. The default wrapping method configuration setting may be changed using the TKE. This keyword is ignored for AES keys.
WRAP-ENH	Use enhanced key wrapping method, which is compliant with the ANSI X9.24 standard.
WRAP-ECB	Use original key wrapping method, which uses ECB wrapping for DES key tokens and CBC wrapping for AES key tokens.
Translation control (Optional)	
ENH-ONLY	Restrict rewrapping of the *output_key_token*. After the token has been wrapped with the enhanced method, it cannot be rewrapped using the original method.

key_identifier

Direction: Input/Output
Type: String

The *key_identifier* parameter is a pointer to a string variable containing the DES internal key-token or the key label of an internal key-token record in key storage.

Restrictions

The restrictions for CSNBKTC.

None.

Required commands

The required commands for CSNBKTC.

If you specify the **RTCMK** keyword, the Key Token Change verb requires the **Symmetric Key Token Change - RTCMK** command (offset X'0090') to be enabled in the active role.

If you specify the **REFORMAT** keyword, the Key Token Change verb requires the **CKDS Conversion2 - Allow use of REFORMAT** command (offset X'014C') to be enabled in the active role.

If you specify the **WRAP-ECB** or **WRAP-ENH** key wrapping method, and the default key-wrapping method setting does not match this keyword, the **CKDS Conversion2 - Allow wrapping override keywords** command (offset X'0146') must be enabled in the active role.

If the **WRAP-ECB** translation-control keyword is specified, and the key in the input key token is wrapped by the enhanced wrapping method (**WRAP-ENH**), the verb requires the **CKDS Conversion2 - Convert from enhanced to original** command (offset X'0147') to be enabled. An active role with offset X'0147' enabled can also use the Key Translate2 verb to translate a key from the enhanced key-wrapping method to the less-secure legacy method.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBKTC.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKTCJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBKTCJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            key_label);
```

Key Token Change2 (CSNBKTC2)

Use the Key Token Change2 verb to re-encipher a variable-length HMAC or AES key from encryption under the old master-key to encryption under the current master-key and to update the keys in internal HMAC or AES key tokens.

Note:

1. An application system is responsible for keeping all of its keys in a usable form. When the master key is changed, the CEX*C implementations can use an internal key that is enciphered by either the current or the old master-key. Before the master key is changed a second time, it is important to have a key re-enciphered under the current master-key for continued use of the key. Use the Key Token Change2 verb to re-encipher such a keys.
2. Previous implementations of IBM CCA products had additional capabilities with this verb such as deleting key records and key tokens in key storage. Also, use of a wild card (*) was supported in those implementations.

Format

The format of CSNBKTC2.

```
CSNBKTC2(
          return_code,
          reason_code,
          exit_data_length,
          exit_data,
          rule_array_count,
          rule_array,
          key_identifier_length
          key_identifier)
```

Parameters

The parameters for CSNBKTC2.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 2.

rule_array

Direction: Input
Type: String array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* keywords are described in Table 71.

Table 71. Keywords for Key Token Change2 control information

Keyword	Description
Algorithm (One, required)	

Table 71. Keywords for Key Token Change2 control information (continued)

Keyword	Description
HMAC	Specifies that the key token is for an HMAC key.
AES	Specifies that the key token is for an AES key in a variable-length symmetric key token.
Re-encipherment method (Required)	
RTCMK	Re-enciphers the internal key provided by the **key_identifier** to the current master-key in an internal key-token in application storage or in key storage. If the supplied key is already enciphered under the current master-key the verb returns a positive response (return code 0, reason code 0). If the supplied key is enciphered under the old master-key, the key is updated to encipherment by the current master-key and the verb returns a positive response (return code 0, reason code 0). Other cases return some form of abnormal response.
RTNMK	Re-enciphers the internal key provided by the **key_identifier** to the new master-key. A key enciphered under the new master key is not usable. It is expected that the user will use this keyword (**RTNMK**) to take a preparatory step in re-enciphering an external key store that they manage themselves to a new master-key, before the SET operation has occurred. Note also that the new master-key register must be full. It must have had the last key part loaded and therefore not be empty or partially full (partially full means that one or more key parts have been loaded, but not the last key part). The SET operation makes the new master-key operational, moving it to the current master-key register, and the current master-key is displaced into the old master-key register. When this happens, all the keys that were re-enciphered to the new master-key are now usable, because the new master-key is not new anymore, it is current. Because the **RTNMK** keyword is added primarily for support of externally managed key storage (see "Key Storage on z/OS (RTNMK-focused)" on page 405), it is not valid to pass a *key_identifer* when the **RTNMK** keyword is used. Only a full internal key token (encrypted under the current master-key) can be passed for re-encipherment with the **RTNMK** keyword. When a key label is passed along with the **RTNMK** keyword, the error return code 8 with reason code 181 will be returned. For more information, see "Key storage with Linux on z Systems, in contrast to z/OS on z Systems" on page 403.
VALIDATE	Validate an internal key token as described in the **key_identifier** which is enciphered under the current master key. That is, the same processing as RTNMK is applied. However, after a successful checking of the token, no re-enciphering of the token to the new master key takes place. There is just a return code for a successful validation.

key_identifier_length

Direction: Input/Output
Type: Integer

The **key_identifier_length** parameter is a pointer to a string variable containing the length in bytes of the **key_identifier** parameter. On input, this variable contains the number of bytes for the *key_identifier* buffer, and must be large enough to hold the key token or key label. On output, this variable contains the number of bytes of data returned in the *key_identifier* variable.

key_identifier

Direction: Input/Output
Type: String

A pointer to a string variable containing an internal variable-length symmetric key-token, or a key label of such a key in AES key-storage. The key token referred to is processed according to the rule-array keywords..

Restrictions

The restrictions for CSNBKTC2.

This verb was introduced with CCA 4.1.0.

Required commands

The required commands for CSNBKTC2.

If you specify the **RTNMK** keyword, this verb requires the **Reencipher CKDS2** command (offset X'00F0') to be enabled in the active role.

If you specify the **RTCMK** keyword, this verb requires the **Symmetric Key Token Change2 - RTCMK** command (offset X'00F1') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBKTC2.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKTC2J.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBKTC2J(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  hikmNativeInteger rule_array_count,
  byte[]            rule_array,
  hikmNativeInteger key_identifier_length,
  byte[]            key_identifier);
```

Key Token Parse (CSNBKTP)

The Key Token Parse verb disassembles a key token into separate pieces of information.

This verb can disassemble an external key token or an internal key token in application storage.

Use the *key_token* parameter to specify the key token to disassemble.

This verb returns some of the key token information in a set of variables identified by individual parameters and the remaining key token information as keywords in the *rule_array*.

Control vector information is returned in keywords found in the *rule_array* when the verb can fully parse the control vector. Otherwise, the verb returns return code 4, reason code 2039.

The Key Token Parse verb performs no cryptographic services.

Format

The format of CSNBKTP.

```
CSNBKTP(
     return_code,
     reason_code,
     exit_data_length,
     edit_data,
     key_token,
     key_type,
     rule_array_count,
     rule_array,
     key_value,
     masterkey_verification_pattern_v03,
     reserved_2,
     reserved_3,
     control_vector,
     reserved_4,
     reserved_5,
     reserved_6,
     master_key_verification_pattern_v00)
```

Parameters

The parameters for CSNBKTP.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

key_token

Direction: Input
Type: String

The *key_token* parameter is a pointer to a string variable in application storage containing an external or internal key-token to be disassembled.

Note: You cannot use a key label for a key-token record in key storage. The key token must be in application storage.

key_type

Direction: Output
Type: String

The *key_type* parameter is a pointer to a string variable containing a keyword defining the key type. The keyword is eight bytes in length and must be left-aligned and padded on the right with space characters. Valid *key_type* keywords are shown here:

CIPHER	CVARXCVL	DKYGENKY	MAC
CIPHERXI	CVARXCVR	ENCIPHER	MACVER
CIPHERXL	DATA	EXPORTER	OKEYXLAT
CIPHERXO	DATAC	IKEYXLAT	OPINENC
CVARDEC	DATAM	IMPORTER	PINGEN
CVARENC	DATAMV	IPINENC	PINVER
CVARPINE	DECIPHER	KEYGENKY	SECMSG

Key types are described in "Types of keys" on page 36.

rule_array_count

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be a minimum of 4.

On input, specify the maximum number of usable array elements that are allocated. On output, the verb sets the value to the number of keywords returned to the application.

rule_array

Direction: Output
Type: String array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords that expresses the contents of the key token. The keywords are eight bytes in length and are left-aligned and padded on the right with space characters. The **rule_array** keywords are described in Table 72.

Table 72. Keywords for Key Token Parse control information

Keyword	Description
Token type (One returned)	
INTERNAL	Specifies an internal key-token.
EXTERNAL	Specifies an external key-token.
Key status (One returned)	
KEY	Indicates the key token contains a key. The *key_value* parameter contains the key.
NO-KEY	Indicates the key token does not contain a key.
Key-wrapping method (One returned)	
WRAP-ECB	The wrapping method for this key is legacy. This keyword was introduced with CCA 4.1.0.
WRAP-ENH	The wrapping method for this key is enhanced. This keyword was introduced with CCA 4.1.0.
Control-vector (CV) status (One returned)	
CV	The key token specifies that a control vector is present. The verb sets the control vector variable with the value of the control vector found in the key token.
NO-CV	The key token does not specify the presence of a control vector. The verb sets the control vector variable with the value of the control vector variable found in the key token.

The difference between Key Token Parse (CSNBKTP) and Control Vector Generate (CSNBCVG) is that Key Token Parse returns the *rule_array* keywords that apply to a parsed token, such as **EXTERNAL**, **INTERNAL**, and so forth. These *rule_array* parameters are returned in addition to *key_type* parameter. Refer to "Key Token Build (CSNBKTB)" on page 213 and "Key Token Build2 (CSNBKTB2)" on page 218 for keyword discussion.

```
AMEX-CSC        DKYL0       EPINGEN      KEYLN16      UKPT
ANSIX9.9        DKYL1       EPINGENA     LMTD-KEK     VISA-PVV
ANY             DKYL2       EPINVER      MIXED        WRAP-ECB
ANY-MAC         DKYL3       EXEX         NO-SPEC      WRAP-ENH
CLR8-ENC        DKYL4       EXPORT       NO-XPORT     XLATE
CPINENC         DKYL5       GBP-PIN      NOOFFSET     XPORT-OK
CPINGEN         DKYL6       GBP-PINO     NOT-KEK
CPINGENA        DKYL7       IBM-PIN      OPEX
CVVKEY-A        DMAC        IBM-PINO     OPIM
CVVKEY-B        DMKEY       IMEX         PIN
DALL            DMPIN       IMIM         REFORMAT
DATA            DMV         IMPORT       SINGLE
```

| DDATA | DOUBLE | INBK-PIN | SMKEY
| DEXP | DPVR | KEY-PART | SMPIN
| DIMP | ENH-ONLY | KEYLN8 | TRANSLAT

key_value

Direction: Input
Type: String

The **key_value** parameter is a pointer to a string variable. If the verb returns the **KEY** keyword in the **rule_array**, the **key_value** parameter contains the 16-byte enciphered key.

| **masterkey_verification_pattern_v03**

Direction: Output
Type: Integer

The **masterkey_verification_pattern_v03** parameter is a pointer to an integer variable. The verb writes zero into the variable except when parsing a version X'03' internal key-token.

reserved_2/5

Direction: Output
Type: Integer

The **reserved_2** and **reserved_5** parameters are either null pointers or pointers to integer variables. If the parameter is not a null pointer, the verb writes zero into the reserved variable.

reserved_3/4

Direction: Output
Type: String

The **reserved_3** and **reserved_4** parameters are either null pointers or pointers to string variables. If the parameter is not a null pointer, the verb writes eight bytes of X'00' into the reserved variable.

reserved_6

Direction: Output
Type: String

The **reserved_6** parameter is either a null pointer or a pointer to a string variable. If the parameter is not a null pointer, the verb writes eight space characters into the reserved variable.

control_vector

Direction: Output
Type: String

The **control_vector** parameter is a pointer to a string variable in application storage. If the verb returns the **NO-CV** keyword in the **rule_array**, the key token did not contain a control-vector value and the control vector variable is filled with 16 space characters.

| **master_key_verification_pattern_v00**

Direction: Output
Type: String

The **master_key_verification_pattern_v00** parameter is a pointer to a string variable in application storage. For version 0 key-tokens that contain a key, the 8-byte master key version number is copied to the variable. Otherwise the variable is filled with eight space characters.

Restrictions

The restrictions for CSNBKTP.

None.

Required commands

The required commands for CSNBKTP.

None.

Usage notes

The usage notes for CSNBKTP.

Be aware that Key Token Parse (CSNBKTP) will fail (return code 8, reason code 49) when given a DES INTERNAL key token that is version X'01'. These tokens are DOUBLE and TRIPLE length DES INTERNAL DATA key tokens.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKTPJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBKTPJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           byte[]            key_token,
           byte[]            key_type,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            key_value,
           hikmNativeInteger master_key_verification_pattern_v03,
           hikmNativeInteger reserved_2,
           byte[]            reserved_3,
           byte[]            control_vector,
           byte[]            reserved_4,
           hikmNativeInteger reserved_5,
           byte[]            reserved_6,
           byte[]            master_key_verification_pattern_v00);
```

Key Token Parse2 (CSNBKTP2)

Use the Key Token Parse2 verb to disassemble a variable-length symmetric key-token into separate pieces of information.

The verb can disassemble an external or internal variable-length symmetric key-token in application storage into separate pieces of information. To parse a fixed-length symmetric key-token, see "Key Token Parse (CSNBKTP)" on page 260.

The **key_token** input parameter specifies the external or internal key token to disassemble. The verb returns some of the key-token information in a set of variables identified by individual parameters, and returns the remaining information as keywords in the rule array.

The key-usage field and key-management field information is returned in keywords found in the rule array when the verb can fully parse the fields. See the **rule_array** parameter on page "rule_array" on page 267 for a table of supported keywords. If the token cannot be parsed successfully, the verb returns a warning using reason code 2039 X'7F7'). If a warning or error occurs during processing, the verb updates all of the count and length variables with a value of zero.

The Key Token Parse2 verb performs no cryptographic services.

To use this verb, specify the following:

- An external or internal variable-length symmetric key-token (version X'05') to be parsed

  This parameter does not accept a key label. The key token must be provided from application storage. If a key token located in key storage needs to be parsed, use the AES Key Record Read verb to retrieve it into application storage before calling this verb.

  See "HMAC key token" on page 799 for the format of this key token. A review of this format information will greatly assist in understanding the output variables of this verb.
- A rule-array-count value large enough for the verb to return keywords about the input key-token in the rule-array buffer

  To determine the exact count required, and also the required lengths of the other string variables, specify a value of zero. This causes the verb to return all count and length values without updating any string variables.
- Adequate buffer sizes for all of the output variables using the length parameters

Format

The format of CSNBKTP2.

```
CSNBKTP2(
          return_code,
          reason_code,
          exit_data_length,
          exit_data,
          key_token_length,
          key_token,
          key_type,
          rule_array_count,
          rule_array,
          key_material_state,
          payload_bit_length,
          payload,
          key_verification_pattern_type,
          key_verification_pattern_length,
          key_verification_pattern,
          key_wrapping_method,
          key_hash_algorithm,
          key_name_length,
          key_name,
          TLV_data_length,
          TLV_data,
          user_associated_data_length,
          user_associated_data,
          verb_data_length,
          verb_data)
```

Parameters

The parameters for CSNBKTP2.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

key_token_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the **key_token** variable.

key_token

Direction: Input
Type: String

A pointer to a string variable containing an external or internal variable-length symmetric key-token to be disassembled. This parameter must not point to a key label.

key_type

Direction: Output
Type: String

A pointer to a string variable containing a keyword for the key type of the input key. The keyword is 8 bytes in length and is left-aligned and padded on

the right with space characters. Valid *key_type* keywords are shown here:

```
CIPHER      DKYGENKY    DESUESCV    EXPORTER    IMPORTER
MAC         PINCALC     PINPROT     PINPRW      SECMSG
```

Key types are described in "Types of keys" on page 36.

rule_array_count

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of 8-byte elements in the *rule_array* variable. The minimum returned value is 3, and the maximum returned value is approximately 50. To determine the exact count required, and also the required lengths of the other string variables, specify a value of zero. This causes the verb to return all count and length values without updating any string variables.

On output, the variable is updated with the actual count of the rule-array keywords. An error is returned if a key token cannot be parsed or any of the output buffers are too small.

rule_array

Direction: Output
Type: String array

A pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length, and are left-aligned and padded on the right with space characters. The returned rule array keywords express the contents of the token.

While Key Token Build2 (CSNBKTB2) assembles an internal variable-length symmetric key-token in application storage from information that you supply via the key words in the verb's **rule_array** parameter, Key Token Parse2 disassembles an input key-token into separate pieces of information which are stored in this verb's **rule_array** as output. Some of the keywords of Key Token Build2 and Key Token Parse2 therefore have the same meaning.

Table 73. Keywords for Key Token Parse2

Keyword	Description
Header section	
Token identifier (one required)	
EXTERNAL	Specifies to disassemble an external variable-length symmetric key-token.
INTERNAL	Specifies to disassemble an internal variable-length symmetric key-token.
Wrapping information section	
Key status (one returned). Refer to the **key_material_state** variable for additional details.	
NO-KEY	Key token does not contain a key. The payload variable is empty.
KEY	Key token contains a partial or complete key. The payload variable contains the clear or encrypted key.
Key verification pattern (KVP) type **Note:** Not a keyword. Value returned in **key_verification_pattern_type** variable.	
KVP **Note:** Not a keyword. Value returned in **key_verification_pattern** variable.	
Encrypted section key-wrapping method **Note:** Not a keyword. Value returned in **key_wrapping_method** variable.	

Table 73. Keywords for Key Token Parse2 (continued)

Keyword	Description
Hash algorithm used for wrapping **Note:** Not a keyword. Value returned in **key_hash_algorithm** variable.	
Associated data section	
Type of algorithm for which the key can be used (one returned)	
AES	Specifies the AES algorithm.
DES	Specifies the DES algorithm.
HMAC	Specifies the HMAC algorithm.
Key type **Note:** Not a keyword. Value returned in **key_type** variable.	
Key-usage field keywords depend on key type: **Key type** **Reference** **CIPHER** Refer to Table 60 on page 224. **DESUESCV** No key-usage field keywords. **DKYGENKY** Refer to Table 61 on page 227. **EXPORTER** Refer to Table 62 on page 231. **IMPORTER** Refer to Table 63 on page 235. **MAC** For AES token algorithm, refer to Table 64 on page 238. For HMAC token algorithm, refer to Table 65 on page 241. **PINCALC** Refer to Table 66 on page 244. **PINPROT** Refer to Table 67 on page 247. **PINPRW** Refer to Table 68 on page 249. **SECMSG** Refer to Table 69 on page 252.	
Key-management field 1, high order byte	
Symmetric-key export control (one returned, all key types)	
NOEX-SYM	Prohibit export using symmetric key.
XPRT-SYM	Allow export using symmetric key.
Unauthenticated asymmetric-key export control (one returned, all key types)	
NOEXUASY	Prohibit export using an unauthenticated asymmetric key.
XPRTUASY	Allow export using unauthenticated asymmetric key.
Authenticated asymmetric-key export control (one returned, all key types)	
NOEXAASY	Prohibit export using authenticated asymmetric key.
XPRTAASY	Allow export using authenticated asymmetric key.
Key-management field 1, low-order byte	

Table 73. Keywords for Key Token Parse2 (continued)

Keyword	Description
DES-key export control (one returned, AES algorithm only)	
NOEX-DES	Prohibit export using DES key
XPRT-DES	Allow export using DES key.
AES-key export control (one returned, AES algorithm only)	
NOEX-AES	Prohibit export using AES key.
XPRT-AES	Allow export using AES key.
RSA-key export control (one returned, AES algorithm only)	
NOEX-RSA	Prohibit export using RSA key
XPRT-RSA	Allow export using RSA key.
NOEX-RAW	Prohibit export using raw key
XPRT-RAW	Allow export using raw key.
Key-management field 2, high order byte	
Key completeness (one returned, all key types)	
MIN3PART	Key if present is incomplete. Key requires at least 2 more parts.
MIN2PART	Key if present is incomplete. Key requires at least 1 more part.
MIN1PART	Key if present is incomplete. Key can be completed or have more parts added.
KEYCMPLT	Key if present is complete. No more parts can be added.
Key-management field 2, low-order byte	
Security history (one returned, all key types)	
UNTRUSTD	Key was encrypted with an untrusted KEK.
WOTUATTR	Key was in a format without type/usage attributes.
WWEAKKEY	Key was encrypted with key weaker than itself.
NOTCCAFM	Key was in a non-CCA format.
WECBMODE	Key was encrypted in ECB mode.
Key-management field 3, high order byte	
Pedigree original rules (one returned, all key types)	
POUNKNWN	Unknown.
POOTHER	Other. Method other than those defined here, probably used in UDX.
PORANDOM	Randomly generated.
POKEYAGR	Established by key agreement such as Diffie-Hellman.
POCLRKC	Created from cleartext key components.
POCLRKV	Entered as a cleartext key value.
PODERVD	Derived from another key.
POKPSEC	Cleartext keys or key parts that were entered at TKE and secured from there to the target card.
Key-management field 3, low-order byte	
Pedigree current rule (one returned, all key types)	
PCUNKNWN	Unknown.
PCOTHER	Other. Method other than those defined here, probably used in UDX.
PCRANDOM	Randomly generated.
PCKEYAGR	Established by key agreement such as Diffie-Hellman.

Table 73. Keywords for Key Token Parse2 (continued)

Keyword	Description
PCCLCOMP	Created from clear text key components.
PCCLVAL	Entered as a clear text key value.
PCDERVD	Derived from another key.
PCMVARWP	Imported from CCA version X'05' variable-length symmetric key-token with pedigree field.
PCMVARNP	Imported from CCA version X'05' variable-length symmetric key-token with no pedigree field.
PCMWCV	Imported from CCA key-token that contained a nonzero control vector.
PCMNOCV	Imported from CCA key-token that had no control vector or contained a zero control vector.
PCMT31WC	Imported from a TR-31 key block that contained a control vector (ATTR-CV option).
PCMT31NC	Imported from a TR-31 key block that did not contain a control vector.
PCMPK1-2	Imported using PKCS 1.2 RSA encryption.
PCMOAEP	Imported using PKCS OAEP encryption.
PCMPKA92	Imported using PKA92 RSA encryption.
PCMZ-PAD	Imported using RSA ZERO-PAD encryption.
PCCNVTWC	Converted from a CCA key-token that contained a nonzero control vector.
PCCNVTNC	Converted from a CCA key-token that had no control vector or contained a zero control vector.
PCKPSEC	Cleartext keys or key parts that were entered at TKE and secured from there to the target card.
PCXVARWP	Exported from CCA version X'05' variable-length symmetric key-token with pedigree field.
PCXVARNP	Exported from CCA version X'05' variable-length symmetric key-token with no pedigree field.
PCXOAEP	Exported using PKCS OAEP encryption.
Optional clear key or encrypted AESKW payload section	
Payload **Note:** Not a keyword. Value returned in the payload variable.	

key_material_state

Direction: Output
Type: Integer

A pointer to an integer variable containing the indicator for the current state of the key material. The valid values are:
0 No key present (internal or external)
1 Key is clear (internal), payload bit length is clear-key bit length
2 Key is encrypted under a KEK (external)
3 Key is encrypted under the master key (internal)

payload_bit_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bits in the token payload. If no key is present, the returned value is 0.

If a clear key is present, the returned value is in the following range:
AES 128, 192, or 256
HMAC
80 - 2048

payload

Direction: Output
Type: String

A pointer to a string variable containing the key material payload. The *payload* parameter must be addressable up to the nearest byte boundary above the *payload_bit_length* if the *payload_bit_length* is not a multiple of 8. This field will contain the clear key or the encrypted key material.

key_verification_pattern_type

Direction: Output
Type: Integer

A pointer to an integer variable containing the indicator for the type of key verification pattern used. The valid values are:

0 No KVP
1 AESMK (8 left-most bytes of SHA-256 hash(X'01' || clear AES MK))
2 KEK VP (8 left-most bytes of SHA-256 hash(X'01' || clear KEK))

key_verification_pattern_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *key_verification_pattern* parameter. The valid values are 0, 8, or 16. The value 16 is reserved.

key_verification_pattern

Direction: Output
Type: String

A pointer to a string variable containing the key verification pattern (KVP) of the key-encrypting key used to wrap this key. If the *key_verification_pattern_type* value indicates that a key verification pattern is present, the pattern will be copied from the token, otherwise this variable is empty.

key_wrapping_method

Direction: Output
Type: Integer

A pointer to an integer variable containing the indicator for the encrypted section key-wrapping method used to protect the key payload. The valid values are:

0 NONE (for clear keys or no key)
2 AESKW (for external or internal key wrapped with an AES KEK)
3 PKOAEP2 (for external tokens wrapped with an RSA public key)

key_hash_algorithm

Direction: Output
Type: Integer

A pointer to an integer variable containing the indicator for the hash algorithm used for wrapping in the key token. The valid values are as follows:

		Key-wrapping method		
Value	**Hash algorithm**	**X'00' (clear key)**	**X'02' (AESKW)**	**X'03' (PKOAEP2)**
0	No hash	X		

Value	Hash algorithm	Key-wrapping method		
		X'00' (clear key)	X'02' (AESKW)	X'03' (PKOAEP2)
1	SHA-1			X
2	SHA-256		X	X
4	SHA-384			X
8	SHA-512			X

key_name_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *key_name* variable. The returned value can be 0 or 64.

key_name

Direction: Output
Type: String

A pointer to a string variable containing the optional key label to be stored in the associated data structure of the key token. If there is no key name, then this variable is empty.

TLV_data_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *TLV_data* variable. The returned value is currently always zero.

TLV_data

Direction: Output
Type: String

A pointer to a string variable containing the optional tag-length-value (TLV) section. This field is currently unused.

user_associated_data_length

Direction: Input/Output
Type: Integer

The *user_associated_data_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *user_associated_data* variable. The returned value is 0 - 255.

user_associated_data

Direction: Output
Type: String

A pointer to a string variable containing the user-associated data to be stored in the key token. This user-definable data is cryptographically bound to the key if it is encrypted. If there is no user-defined associated data, this variable is empty.

verb_data_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the **verb_data** variable. The returned value is zero if the returned **key_type** variable is not DKYGENKY. Otherwise the value can be greater than zero and is a multiple of 8.

verb_data

Direction: Output
Type: String

A pointer to a string variable containing any related key-usage field keywords of an AES DKYGENKY key for the for the type of key to be diversified:

DKYGENKY type of key to diversify	Meaning of verb_data keywords
D-CIPHER	Key usage fields for AES CIPHER key.
D-EXP	Key usage fields for AES EXPORTER key.
D-IMP	Key usage fields for AES IMPORTER key.
D-MAC	Key usage fields for AES MAC key.
D-PPROT	Key-usage fields for AES PINPROT key.
D-PCALC	Key-usage fields for AES PINCALC key.
D-PPRW	Key-usage fields for AES PINPRW key.
SECMSG	Key-usage fields for AES SECMSG key.

Required commands

The required commands for CSNBKTP.

None.

Usage notes

The usage notes for CSNBKTP2.

If an error occurs while processing the input token, all output lengths are updated to zero and an error is returned.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKTP2J.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBKTP2J(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger key_token_length,
           byte[]            key_token,
           byte[]            key_type,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger key_material_state,
```

```
hikmNativeInteger payload_bit_length,
byte[]            payload,
hikmNativeInteger key_verification_pattern_type,
hikmNativeInteger key_verification_pattern_length,
byte[]            key_verification_pattern,
hikmNativeInteger key_wrapping_method,
hikmNativeInteger key_hash_algorithm,
hikmNativeInteger key_name_length,
byte[]            key_name,
hikmNativeInteger TLV_data_length,
byte[]            TLV_data,
hikmNativeInteger user_associated_data_length,
byte[]            user_associated_data,
hikmNativeInteger verb_data_length,
byte[]            verb_data);
```

Key Translate (CSNBKTR)

The Key Translate verb uses one key-encrypting key to decipher an input key and then enciphers this key using another key-encrypting key within the secure environment.

Note: All key labels must be unique.

Format

The format of CSNBKTR.

```
CSNBKTR(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             input_key_token,
             input_KEK_key_identifier,
             output_KEK_key_identifier,
             output_key_token)
```

Parameters

The parameters for CSNBKTR.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

input_key_token

Direction: Input
Type: String

A 64-byte string variable containing an external key token. The external key token contains the key to be re-enciphered (translated).

input_KEK_key_identifier

Direction: Input/Output
Type: String

A 64-byte string variable containing the internal key token or the key label of an internal key token record in the DES key storage file. The internal key token contains the key-encrypting key used to decipher the key. The internal key token must contain a control vector that specifies an IMPORTER or IKEYXLAT

key type. The control vector for an IMPORTER key must have the XLATE bit set to B'1'.

output_KEK_key_identifier

Direction: Input/Output
Type: String

A 64-byte string variable containing the internal key token or the key label of an internal key token record in the DES key storage file. The internal key token contains the key-encrypting key used to encipher the key. The internal key token must contain a control vector that specifies an EXPORTER or OKEYXLAT key type. The control vector for an EXPORTER key must have the XLATE bit set to B'1'.

output_key_token

Direction: Output
Type: String

A 64-byte string variable containing an external key token. The external key token contains the re-enciphered key.

Restrictions

The restrictions for CSNBKTR.

Triple length DATA key tokens are not supported.

Required commands

The required commands for CSNBKTR.

This verb requires the **Key Translate** command (offset X'001F') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBKTR.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKTRJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBKTRJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           byte[]            input_key_token,
           byte[]            input_KEK_key_identifier,
           byte[]            output_KEK_key_identifier,
           byte[]            output_key_token);
```

Key Translate2 (CSNBKTR2)

The Key Translate2 verb uses one key-encrypting key to decipher an input key and then enciphers this key using another key-encrypting key within the secure environment.

It can also be used to change the wrapping method of the key with a single key-encrypting key.

To reencipher a key token, specify the external key token, input and output key-encrypting keys. You can specify which key wrapping method to use. If no wrapping method is specified, the wrapping method of the *input_key_token* will be used.

To change the wrapping method of an external key token, specify the **REFORMAT** rule array keyword, the wrapping method to use, the external key token, and the input key-encrypting key. If no wrapping method is specified, the wrapping method of the *input_key_token* will be used. Note that the *output_KEK_identifier* will be ignored.

Note: All key labels must be unique.

Format

The format of CSNBKTR2.

```
CSNBKTR2(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              rule_array_count,
              rule_array,
              input_key_token_length,
              input_key_token,
              input_KEK_key_identifier_length,
              input_KEK_key_identifier,
              output_KEK_key_identifier_length,
              output_KEK_key_identifier,
              output_key_length,
              output_key_token)
```

Parameters

The parameters for CSNBKTR2.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters common to all verbs” on page 20.

rule_array_count

Direction: Input
Type: Integer

The number of keywords you supplied in the *rule_array* parameter. This value must be 0, 1, 2, or 3.

rule_array

Direction: Input

Type: String array

Keywords that provide control information to the verb. The keywords must be 8 bytes of contiguous storage with the keyword left-aligned in its 8-byte location and padded on the right with blanks. The *rule_array* keywords are described in Table 74.

Table 74. Keywords for Key Translate2 control information

Keyword	Description
Encipherment (Optional)	
REFORMAT	Reformat the **input_key_token**. • When the **input_key_token** is a DES key token, reformat with the Key Wrapping Method specified. • When the **input_key_token** is an operational AES key token, either reformat an AES DATA key (version X'04') to an AES CIPHER key (version X'05') or the reverse (version X'05' to version X'04').
TRANSLAT	Translate the **input_key_token** from encipherment under the **input_KEK_identifier** to encipherment under the **output_KEK_identifier**. This is the default.
V1PYLD	Reencipher an input variable-length AES key token (version X'05') to a payload version1 (fixed-length) key token. This keyword is only valid for the CIPHER, EXPORTER and IMPORTER key types.
V0PYLD	Reencipher an input variable-length AES key token (version X'05') to a payload version 0 (variable-length) key token. This keyword is only valid for the CIPHER, EXPORTER and IMPORTER key types.
Key-wrapping method (optional, valid only if input_key_token is an external DES key token)	
USECONFG	This is the default. Specifies to wrap the key using the configuration setting for the default wrapping method. The default wrapping method configuration setting may be changed using the TKE. This keyword is ignored for AES keys.
WRAP-ENH	Use enhanced key wrapping method, which is compliant with the ANSI X9.24 standard.
WRAP-ECB	Use original key wrapping method, which uses ECB wrapping for DES key tokens.
Translation control (Optional, valid only with WRAP-ENH)	
ENH-ONLY	Restrict the re-wrapping of the *output_key_token*. Once the token has been wrapped with the enhanced method, it cannot be re-wrapped using the original method.
Algorithm (One required, if the V0PYLD or V1PYLD keyword is specified)	
AES	Specifies that the input key is an AES key. Where used, the key-encrypting keys will be AES transport keys.
DES	Specifies that the input key is a DES key. Where used, the key-encrypting keys will be DES transport keys. This is the default.
HMAC	Specifies that the input key is an HMAC key. Where used, the key-encrypting keys will be AES transport keys.

input_key_token_length

Direction: Input
Type: Integer

The length of the *input_key_token* in bytes. The maximum value allowed is 725.

input_key_token

Direction: Input
Type: String

A variable length string variable containing the external key token. The external key token contains the key to be re-enciphered (or re-wrapped).

input_KEK_key_identifier_length

Direction: Input
Type: Integer

The length of the *input_KEK_key_identifier* in bytes. The maximum value allowed is 725.

input_KEK_key_identifier

Direction: Input/Output
Type: String

A variable length string variable containing the internal key token or the key label of an internal key token record in the key storage file. The internal key token contains the key-encrypting key used to decipher the key. The internal key token must contain a control vector that specifies an IMPORTER or IKEYXLAT key type. The control vector for an IMPORTER key must have the XLATE bit set to B'1'.

output_KEK_key_identifier_length

Direction: Input
Type: Integer

The length of the *output_KEK_key_identifier* in bytes. The maximum value is 725.

If the **REFORMAT** keyword is specified, this value must be 0.

output_KEK_key_identifier

Direction: Input/Output
Type: String

A variable length string variable containing the internal key token or the key label of an internal key token record in the key storage file. The internal key token contains the key-encrypting key used to encipher the key. The internal key token must contain a control vector that specifies an EXPORTER or OKEYXLAT key type. The control vector for an exporter key must have the XLATE bit set to B'1'.

If the **REFORMAT** keyword is specified, this parameter is ignored.

output_key_token_length

Direction: Input/Output
Type: Integer

On input, the length of the output area provided for the *output_key_token*. This must be at least 64 bytes. On output, the parameter is updated with the length of the token copied to the *output_key_token*.

output_key_token

Direction: Output
Type: String

A variable length string variable containing an external key token. The external key token contains the re-enciphered key.

Restrictions

The restrictions for CSNBKTR2.

This verb does not support version X'10' external DES key tokens (RKX key tokens).

This verb was introduced with CCA 4.1.0.

Required commands

The required commands for CSNBKTR2.

This verb requires the **Key Translate2 - Allow use of REFORMAT** command (offset X'014B') to be enabled in the active role if the **REFORMAT** reencipherment keyword is used.

Otherwise, the verb requires the **Key Translate2** command (offset X'0149') to be enabled.

To use the translation control keyword **WRAP-ECB** or **WRAP-ENH** when the default key-wrapping method setting does not match the keyword, the **Key Translate2 - Allow wrapping override keywords** command (offset X'014A') must be enabled.

If the **WRAP-ECB** translation-control keyword is specified and the key in the input key token is wrapped by the enhanced wrapping method (**WRAP-ENH**), the verb requires the **CKDS Conversion2 - Convert from enhanced to original** command (offset X'0147') to be enabled. An active role with offset X'0149' enabled can also use the Key Token Change verb to translate a key from the enhanced key-wrapping method to the less-secure legacy method.

The **Key Translate2 - Disallow AES ver 5 to ver 4 conversion** command (offset X'032A') prevents **CIPHER** keys, which are in variable-length AES key tokens (newer version X'05') and wrapped under the AES master-key, from being reformatted into **DATA** keys, which are in fixed-length AES key tokens (older version X'04') and wrapped under the less-secure DES master-key. This command overrides the **Key Translate2 - Allow use of REFORMAT** command (offset X'014B').

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBKTR2.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKTR2J.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBKTR2J(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
```

```
hikmNativeInteger exit_data_length,
byte[]            exit_data,
hikmNativeInteger rule_array_count,
byte[]            rule_array,
hikmNativeInteger input_key_token_length,
byte[]            input_key_token,
hikmNativeInteger input_KEK_key_identifier_length,
byte[]            input_KEK_key_identifier,
hikmNativeInteger output_KEK_key_identifier_length,
byte[]            output_KEK_key_identifier,
hikmNativeInteger output_key_token_length,
byte[]            output_key_token);
```

PKA Decrypt (CSNDPKD)

Use this verb to decrypt (unwrap) a formatted key value. This verb unwraps the key, parses it, and returns the parsed value to the application in the clear.

PKCS 1.2 and **ZERO-PAD** formatting are supported. For **PKCS 1.2**, the decrypted data is examined to ensure that it meets RSA DSI PKCS #1 block type 2 format specifications. **ZERO-PAD** is supported only for external or clear RSA private keys.

For the **PKCSOAEP** recovery method keyword, the decrypted data is examined to ensure that it meets the RSAES-OAEP scheme of the RSA PKCS #1 v2.0 standard. See also "PKCS #1 hash formats" on page 950.

This verb allows the use of clear or encrypted RSA private keys. If an external clear key token is used, the master keys are not required to be installed in any cryptographic coprocessor and PKA verbs do not have to be enabled.

Format

The format of CSNDPKD.

```
CSNDPKD(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              rule_array_count,
              rule_array,
              PKA_enciphered_keyvalue_length,
              PKA_enciphered_keyvalue,
              data_structure_length,
              data_structure,
              PKA_key_identifier_length,
              PKA_key_identifier,
              target_keyvalue_length,
              target_keyvalue)
```

Parameters

The parameters for CSNDPKD.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1 or 2.

rule_array

Direction: Input
Type: String array

The keyword that provides control information to the verb. The keyword is left-aligned in an 8-byte field and padded on the right with blanks. The *rule_array* keywords are described in Table 75.

Table 75. Keywords for PKA Decrypt control information

Keyword	Description
Recovery method (One, required). Specifies the method to use to recover the key value.	
PKCS-1.2	RSA DSI PKCS #1 block type 02 will be used to recover the key value. In the RSA PKCS #1 v2.0 standard, RSA terminology describes this as the RSAES-PKCS1-v1_5 format.
PKCSOAEP	Specifies that the key is formatted as defined in the RSA PKCS #1 v2.0 standard for the RSAES-OAEP encryption/decryption scheme. See "PKCS #1 hash formats" on page 950.
ZERO-PAD	The input *PKA_enciphered_keyvalue* is decrypted using the RSA private key. The entire result (including leading zeros) will be returned in the *target_keyvalue* field. The *PKA_key_identifier* must be an external RSA token or the label of an external token.
Hash method (One required for **PKCSOAEP**, not allowed for any other recovery method).	
SHA-1	Specifies to use the SHA-1 hash method to calculate the OAEP message hash.
SHA-256	Specifies to use the SHA-256 hash method to calculate the OAEP message hash.

PKA_enciphered_keyvalue_length

Direction: Input
Type: Integer

The length of the *PKA_enciphered_keyvalue* parameter in bytes. The maximum size that you can specify is 256 bytes. The length should be the same as the modulus length of the *PKA_key_identifier*.

PKA_enciphered_keyvalue

Direction: Input
Type: String

This field contains the key value protected under an RSA public key. This byte-length string is left-aligned within the *PKA_enciphered_keyvalue* parameter.

data_structure_length

Direction: Input
Type: Integer

This value must be 0.

data_structure

Direction: Input
Type: String

This parameter is ignored.

PKA_key_identifier_length

Direction: Input
Type: Integer

The length of the *PKA_key_identifier* parameter. When the *PKA_key_identifier* is a key label, this field specifies the length of the label. The maximum size that you can specify is 2500 bytes.

PKA_key_identifier

Direction: Input
Type: String

An internal RSA private key token, the label of an internal RSA private key token, or an external RSA private key token containing a clear RSA private key in Modulus-Exponent or Chinese Remainder Theorem format. The corresponding public key was used to wrap the key value.

target_keyvalue_length

Direction: Input/Output
Type: Integer

The length of the *target_keyvalue* parameter. The maximum size that you can specify is 256 bytes. On return, this field is updated with the actual length of *target_keyvalue*.

If **ZERO-PAD** is specified, this length will be the same as the *PKA_enciphered_keyvalue_length* which is equal to the RSA modulus byte length.

target_keyvalue

Direction: Output
Type: String

This field will contain the decrypted, parsed key value. If **ZERO-PAD** is specified, the decrypted key value, including leading zeros, will be returned.

Restrictions

The restrictions for CSNDPKD.

- The exponent of the RSA public key must be odd.
- Rule array keywords PKCSOAEP, SHA-1, and SHA-256 are not supported in releases before Release 4.4.
- The **PKA Decipher - Key Data Disallow PKCS-1.2** command (offset X'020A'), the **PKA Decipher - Key Data Disallow ZERO-PAD** command (offset X'020B'), and the **PKA Decipher - Key Data Disallow PKCSOAEP** command (offset X'020C') are not defined in releases before Release 4.4.

Required commands

The required commands for CSNDPKD.

This verb requires the **PKA Decrypt** command (offset X'011F') to be enabled in the active role.

The PKA Decrypt verb also requires the following commands to be enabled in the active role to disallow the unwrapping of an encrypted key that has been formatted:

Rule-array keyword	Offset	Command
PKCS-1.2	X'020A'	**PKA Decipher - Key Data Disallow PKCS-1.2**
ZERO-PAD	X'020B'	**PKA Decipher - Key Data Disallow ZERO-PAD**
PKCSOAEP	X'020C'	**PKA Decipher - Key Data Disallow PKCSOAEP**

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNDPKD.

The RSA private key must be enabled for key management functions.

The hardware configuration sets the limit on the modulus size of keys for key management; thus, this verb will fail if the RSA key modulus bit length exceeds this limit.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDPKDJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNDPKDJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  hikmNativeInteger rule_array_count,
  byte[]            rule_array,
  hikmNativeInteger enciphered_key_length,
  byte[]            enciphered_key,
  hikmNativeInteger data_struct_length,
  byte[]            data_struct,
  hikmNativeInteger RSA_private_key_length,
  byte[]            RSA_private_key,
  hikmNativeInteger key_value_length,
  byte[]            key_value);
```

PKA Encrypt (CSNDPKE)

This verb encrypts a supplied clear key value under an RSA public key.

The supplied key can be formatted using the **PKCS 1.2** or **ZERO-PAD** methods prior to encryption. Beginning with Release 4.4, the supplied key can also be formatted using the PKCSOAEP method. The *rule_array* keyword specifies the format of the key prior to encryption.

Format

The format of CSNDPKE.

```
CSNDPKE(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             keyvalue_length,
             keyvalue,
             data_structure_length,
             data_structure,
             PKA_key_identifier_length,
             PKA_key_identifier,
             PKA_enciphered_keyvalue_length,
             PKA_enciphered_keyvalue)
```

Parameters

The parameters for CSNDPKE.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1 or 2.

rule_array

Direction: Input
Type: String array

A keyword that provides control information to the verb. The keyword is left-aligned in an 8-byte field and padded on the right with blanks. The *rule_array* keywords are described in Table 76.

Table 76. Keywords for PKA Encrypt control information

Keyword	Description
Formatting Method (One, required). Specifies the method to use to format the key value prior to encryption.	
MRP	The key value will be padded on the left with binary zeros to the length of the PKA key modulus. The RSA public key can have an even or odd exponent.
PKCS-1.2	RSA DSI PKCS #1 block type 02 format will be used to format the supplied key value. In the RSA PKCS #1 v2.0 standard, RSA terminology describes this as the RSAES-PKCS1-v1_5 format.
PKCSOAEP	Specifies that the key is formatted as defined in the RSA PKCS #1 v2.0 standard for the RSAES-OAEP encryption/decryption scheme. See "PKCS #1 hash formats" on page 950.
ZERO-PAD	Places the clear_source_data variable in the low-order bit positions of a bit string of the same length as the modulus. The data is padded on the left as needed with binary zeros to the length of the PKA key modulus. The RSA public key must have an odd exponent.
Hash method (One required for **PKCSOAEP**, not allowed for any other recovery method).	
SHA-1	Specifies to use the SHA-1 hash method to calculate the OAEP message hash.

Table 76. Keywords for PKA Encrypt control information (continued)

Keyword	Description
SHA-256	Specifies to use the SHA-256 hash method to calculate the OAEP message hash.

keyvalue_length

Direction: Input
Type: Integer

The length of the *keyvalue* parameter. The maximum field size is 256 bytes. The actual maximum size depends on the modulus length of *PKA_key_identifier* and the formatting method you specify in the *rule_array* parameter. See "Usage notes" on page 286.

keyvalue

Direction: Input
Type: String

This field contains the supplied clear key value to be encrypted under the *PKA_key_identifier*.

data_structure_length

Direction: Input
Type: Integer

This value must be 0.

data_structure

Direction: Input
Type: String

This field is currently ignored.

PKA_key_identifier_length

Direction: Input
Type: Integer

The length of the *PKA_key_identifier* parameter. When the *PKA_key_identifier* is a key label, this field specifies the length of the label. The maximum size that you can specify is 2500 bytes.

PKA_key_identifier

Direction: Input
Type: String

The RSA public or private key token or the label of the RSA public or private key to be used to encrypt the supplied key value.

PKA_enciphered_keyvalue_length

Direction: Input/Output
Type: Integer

The length of the *PKA_enciphered_keyvalue* parameter in bytes. The maximum size that you can specify is 256 bytes. On return, this field is updated with the actual length of *PKA_enciphered_keyvalue*.

This length should be the same as the modulus length of the *PKA_key_identifier*.

PKA_enciphered_keyvalue

Direction: Output
Type: String

This field contains the key value protected under an RSA public key. This byte-length string is left-aligned within the *PKA_enciphered_keyvalue* parameter.

Restrictions

The restrictions for CSNDPKE.

- A message can be encrypted provided that it is smaller than the public key modulus.

  The term 'smaller' refers to the exact bit count, not the byte count of the modulus. For example, counting bits, the hexadecimal number X'FF' is several bits longer than the number X'1F', even though both numbers are one byte long as represented in computer memory.
- The exponent of the RSA public key must be odd unless the MRP keyword is supplied.
- The RSA public key modulus size (key size) is limited by the Function Control Vector to accommodate governmental export and import regulations.
- Rule array keywords PKCSOAEP, SHA-1, and SHA-256 are not supported in releases before Release 4.4.
- The **PKA Encipher - Clear Key Disallow MRP** (offset X'0208'), **PKA Encipher - Clear Key Disallow PKCS-1.2** (offset X'0206'), **PKA Encipher - Clear Key Disallow PKCSOAEP** (offset X'0209), and **PKA Encipher - Clear Key Disallow ZERO-PAD** (offset X'0207') are not defined in releases before Release 4.4.

Required commands

The required commands for CSNDPKE.

This verb requires the **PKA Encrypt** command (offset X'011E') to be enabled in the active role.

The PKA Encrypt verb also requires the following commands to be enabled in the active role to disallow the encryption of clear source data:

Rule-array keyword	Offset	Command
MRP	X'0208'	**PKA Encipher - Clear Key Disallow MRP**
PKCS-1.2	X'0206'	**PKA Encipher - Clear Key Disallow PKCS-1.2**
PKCSOAEP	X'0209'	**PKA Encipher - Clear Key Disallow PKCSOAEP**
ZERO-PAD	X'0207'	**PKA Encipher - Clear Key Disallow ZERO-PAD**

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNDPKE.

- For RSA DSI PKCS #1 formatting, the key value length must be a minimum of 11 bytes less than the modulus length of the RSA key.

- The hardware configuration sets the limit on the modulus size of keys for key management; thus, this service will fail if the RSA key modulus bit length exceeds this limit.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDPKEJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNDPKEJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  hikmNativeInteger rule_array_count,
  byte[]            rule_array,
  hikmNativeInteger key_value_length,
  byte[]            key_value,
  hikmNativeInteger data_struct_length,
  byte[]            data_struct,
  hikmNativeInteger RSA_public_key_length,
  byte[]            RSA_public_key,
  hikmNativeInteger RSA_encipher_length,
  byte[]            RSA_encipher);
```

Prohibit Export (CSNBPEX)

Use this verb to modify an operational key so that it cannot be exported.

Format

The format of CSNBPEX.

```
CSNBPEX(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              key_identifier)
```

Parameters

The parameters for CSNBPEX.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

key_identifier

Direction: Input/Output
Type: String

A 64-byte string variable containing the internal key token to be modified. The returned *key_identifier* will be encrypted under the current master key.

Restrictions

The restrictions for CSNBPEX.

None.

Required commands

The required commands for CSNBPEX.

This verb requires the **Prohibit Export** command (offset X'00CD') to be enabled in the active role.

The following access-control points are added beginning with Release 4.3:

- To disable the wrapping of a key with a weaker master key, the **Prohibit weak wrapping - Master keys** command (offset X'0333') must be enabled in the active role.
- To receive a warning when wrapping a key with a weaker master key, enable the **Warn when weak wrap - Master keys** command (offset X'0332') in the active role. The **Prohibit weak wrapping - Master keys** command overrides this command.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBPEX.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBPEXJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBPEXJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           byte[]            key_identifier);
```

Prohibit Export Extended (CSNBPEXX)

Use this verb to modify an exportable external CCA DES key-token so that its key can no longer be exported.

This verb performs the following functions:

- Multiply deciphers the source key under a key formed by the XOR of the source key’s control vector and the specified key-encrypting key (KEK).
- Turns from on to off the XPORT-OK bit in the source key’s control vector (bit 17).

- Multiply enciphers the key under a key formed by the XOR of the KEK key and the source key's modified control vector. The encrypted key and the modified control vector are stored in the source-key key token, and the TVV is updated.

Format

The format of CSNBPEXX.

```
CSNBPEXX(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              source_key_token,
              KEK_key_identifier)
```

Parameters

The parameters for CSNBPEXX.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

source_key_token

Direction: Input/Output
Type: String

A pointer to a string variable containing an external key-token.

KEK_key_identifier

Direction: Input
Type: String

A pointer to a string variable containing an internal key-encrypting token, or the key label of an internal key-encrypting token record.

Restrictions

The restrictions for CSNBPEXX.

This verb does not support version X'10' external DES key tokens (RKX key tokens).

Required commands

The required commands for CSNBPEXX.

This verb requires the **Prohibit Export Extended** command (offset X'0301') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBPEXX.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBPEXXJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBPEXXJ(
          hikmNativeInteger return_code,
          hikmNativeInteger reason_code,
          hikmNativeInteger exit_data_length,
          byte[]            exit_data,
          byte[]            source_key_token,
          byte[]            KEK_key_identifier);
```

Restrict Key Attribute (CSNBRKA)

Use the Restrict Key Attribute verb to modify an exportable internal or external variable-length symmetric key-token so that its key can no longer be exported.

Format

The format of CSNBRKA.

```
CSNBRKA (
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array
             key_identifier_length
             key_identifier
             key_encrypting_key_identifier_length
             key_encrypting_key_identifier
             opt_parameter1_length
             opt_parameter1
             opt_parameter2_length
             opt_parameter2)
```

Parameters

The parameters for CSNBRKA.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

The number of keywords you supplied in the *rule_array* parameter. This value must be 1 - 10.

rule_array

Direction: Input
Type: String array

The *rule_array* contains keywords that provide control information to the verb.

The keywords must be in contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. The *rule_array* keywords are described in Table 77.

Table 77. Keywords for Restrict Key Attribute control information

Token algorithm (one required)	
AES	Specifies to further restrict one or more attributes of a variable-length AES key-token. This keyword is not supported for a fixed-length AES key-token.
DES	Specifies to further restrict one or more attributes of a fixed-length DES key-token.
HMAC	Specifies to further restrict one or more attributes of a variable-length HMAC key-token.
Keywords for AES or HMAC variable-length key tokens (key token for clear key not allowed)	
General attribute restriction (one, optional). Ignored if attribute restriction keyword is specified.	
NOEXPORT	Prohibits the key from being exported by any verb. This keyword is equivalent to providing all of the export control attribute restrictions keywords (**NOEX-AES**, **NOEX-DES**, **NOEX-RSA**, **NOEX-RAW**, **NOEXSYM**, **NOEXAASY**, and **NOEXUASY**). This is the default when no attribute restrictions keywords (export control and key usage) are specified.
Export control attribute restrictions (one or more, optional).	
NOEX-AES	Prohibits the key from being exported using an AES key.
NOEX-DES	Prohibits the key from being exported using a DES key.
NOEX-RSA	Prohibits the key from being exported using an RSA key.
NOEX-RAW	Prohibits the key from being exported using a RAW key. Defined for future use. Currently ignored.
NOEX-SYM	Prohibits the key from being exported using a symmetric key.
NOEXAASY	Prohibits the key form being exported using an authenticated asymmetric key (for example, an RSA key in a trusted block).
NOEXUASY	Prohibits the key from being exported using an unauthenticated asymmetric key.
Ciphertext translation restriction (one, optional). Only valid for AES CIPHER keys.	
C-XLATE	Restricts the key to being used by the Cipher_Text_Translate2 (CSNBCTT2) verb.
KEK identifier rule (one, optional). Not allowed if KEK_identifier_length variable is 0. Required if KEK identifier is a key label and the key in key storage is an RSA KEK). Release 4.2 or later.	
IKEK-AES	The inbound key-encrypting key for the external key is an AES KEK. This is the default.
IKEK-PKA	The inbound key-encrypting key for the external key is an asymmetric key. Required if an RSA transport key is used (that is, the key being changed is wrapped using the PKOAEP2 method).
Keywords for DES fixed-length key tokens	
General attribute restriction (one, optional). Ignored if attribute restriction or key restriction keyword is specified.	
NOEXPORT	Prohibits the key from being exported by any verb. Use this keyword to set XPORT-OK off (CV bit 17 = B'0') and NOT31XPT (CV bit 27 = B'1'). This is the default if no attribute restriction or key restriction keywords are specified.
Attribute to restrict (one, optional). Keywords **CCAXPORT** and **NOT31XPT** in this group are defaults if no keyword is provided. Only valid for DES. Make multiple calls to this verb as needed to restrict more than one of these attributes.	
CCAXPORT	Prohibits the key from being exported by any verb. Use this keyword to set XPORT-OK off (CV bit 17 = B'0').

Table 77. Keywords for Restrict Key Attribute control information (continued)

DOUBLE-O	For double-length DES key tokens that do not have equal key halves (ignoring parity bits), sets CV bit 40 = B'1' to guarantee that the two 8-byte key values of a DES double-length key are unique, that is, the key halves are not replicated. Key type must be EXPORTER, IKEYXLAT, IMPORTER, or OKEYXLAT. **Note:** A double-length key with replicated key halves has an effective strength of a single-length key.
NOT31XPT (Release 4.2 or later)	Sets export control CV to NOT31XPT (bit 57 = B'1') to prohibit TR-31 export of the key.
KEK identifier rule (one, optional). Not allowed if KEK_identifier_length variable is 0. Release 4.2 or later.	
IKEK-DES	The inbound key-encrypting key for the external key is a DES KEK. This is the default.

key_identifier_length

Direction: Input
Type: Integer

The length of the **key_identifier** parameter in bytes. The maximum value is 725.

key_identifier

Direction: Input
Type: String

The key for which the export control is to be updated. The parameter contains an internal token or the 64-byte label of the key in key storage. If a label is specified, the key token will be updated in key storage and not returned by this verb.

key_encrypting_key_identifier_length

Direction: Input
Type: Integer

The byte length of the *key_encrypting_key_identifier* parameter. This value must be 0.

key_encrypting_key_identifier

Direction: Input
Type: String

This parameter is ignored.

opt_parameter1_length

Direction: Input
Type: Integer

The byte length of the *opt_parameter1* parameter. This value must be 0.

opt_parameter1

Direction: Input
Type: String

This parameter is ignored.

opt_parameter2_length

Direction: Input
Type: Integer

The byte length of the *opt_parameter2* parameter. This value must be 0.

opt_parameter2

Direction: Input
Type: String

This parameter is ignored.

Restrictions

The restrictions for CSNBRKA.

This verb was introduced with CCA 4.1.0.

Required commands

The required commands for CSNBRKA.

The Restrict Key Attribute verb requires the following commands to be enabled in the active role:

Rule-array keyword	Offset	Command
AES or **HMAC**	X'00E9'	**Restrict Key Attribute - Export Control**
DES	X'0154'	**Restrict Key Attribute - Permit setting the TR-31 export bit**

The following access-control points are added beginning with Release 4.3:

- To disable the wrapping of a key with a weaker master key, the **Prohibit weak wrapping - Master keys** command (offset X'0333') must be enabled in the active role.
- To receive a warning when wrapping a key with a weaker master key, enable the **Warn when weak wrap - Master keys** command (offset X'0332') in the active role. The **Prohibit weak wrapping - Master keys** command overrides this command.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBPEXX.

This verb is available starting with CCA 4.1.0.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBRKAJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBRKAJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger key_identifier_length,
           byte[]            key_identifier,
           hikmNativeInteger key_encrypting_key_identifier_length,
           byte[]            key_encrypting_key_identifier,
           hikmNativeInteger opt_parameter1_length,
           byte[]            opt_parameter1,
           hikmNativeInteger opt_parameter2_length,
           byte[]            opt_parameter2);
```

Random Number Generate (CSNBRNG)

This verb uses the cryptographic feature to generate a cryptographic-quality random number.

Format

The format of CSNBRNG.

```
CSNBRNG(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             form,
             random_number)
```

Parameters

The parameters for CSNBRNG.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

form

Direction: Input
Type: String

The 8-byte keyword that defines the characteristics of the random number should be left-aligned and padded on the right with blanks. The keywords are listed in Table 78.

Table 78. Keywords for Random Number Generate form parameter

Keyword	Description
EVEN	Generate a 64-bit random number with even parity in each byte.
ODD	Generate a 64-bit random number with odd parity in each byte.
RANDOM	Generate a 64-bit random number.

Parity is calculated on the seven high-order bits in each byte and is presented in the low-order bit in the byte.

random_number

Direction: Output
Type: String

The generated number returned by the verb in an 8-byte variable.

Restrictions

The restrictions for CSNBRNG.

None.

Required commands

The required commands for CSNBRNG.

This verb requires the **Key Generate - OP** command (offset X'008E') to be enabled in the active role.

Usage notes

The usage notes for CSNBRNG.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBRNGJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBRNGJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           byte[]            form,
           byte[]            random_number);
```

Random Number Generate Long (CSNBRNGL)

This verb uses the cryptographic feature to generate a cryptographic-quality random number from 1 - 8192 bytes in length.

Choose the parity of each generated random byte as even, odd, or random. This verb returns the random number in a string variable.

Because this verb uses cryptographic processes, the quality of the output is better than that which higher-level language compilers typically supply.

Format

The format of CSNBRNGL.

```
CSNBRNGL(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             seed_length,
             seed,
             random_number_length,
             random_number)
```

Parameters

The parameters for CSNBRNGL.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1.

rule_array

Direction: Input
Type: String array

A pointer to a string variable containing an array of keywords. The keywords are eight bytes in length, and must be left-aligned and padded on the right with space characters. The *rule_array* keywords are described in Table 79.

Table 79. Keywords for Random Number Generate Long control information

Keyword	Description
Parity adjust (One required)	
EVEN	Specifies that each generated random byte is adjusted for even parity.
ODD	Specifies that each generated random byte is adjusted for odd parity.
RANDOM	Specifies that each generated random byte is not adjusted for parity.

seed_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes in the *seed* variable. This value must be 0.

seed

Direction: Input
Type: String

This parameter is ignored.

random_number_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes in the *random_number* variable. On input, the minimum value is 1 and the maximum value is 8192.

Use this variable to specify the number of random bytes that the verb is to return. On output, this variable contains the number of bytes returned by the verb in the *random_number* variable.

random_number

Direction: Output
Type: String

A pointer to a string variable containing the random number generated.

Restrictions

The restrictions for CSNBRNGL.

None.

Required commands

The required commands for CSNBRNGL.

None.

Usage notes

The usage notes for CSNBRNGL.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBRNGLJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBRNGLJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger reserved_seed_length,
           byte[]            reserved_seed,
           hikmNativeInteger random_number_length,
           byte[]            random_number);
```

Symmetric Key Export (CSNDSYX)

Use the Symmetric Key Export verb to transfer an application-supplied AES DATA (version X'04'), DES DATA, or variable-length symmetric key token key from encryption under the AES or DES master key to encryption under an application-supplied RSA public key or AES EXPORTER key.

The application-supplied key must be an AES, DES or HMAC internal key token or the label of an AES or DES key token in the AES or DES key storage file.

Beginning with CCA 4.1.0, the verb can also export an HMAC key that is contained in an internal variable-length symmetric key-token. The exported key is returned in an external variable-length symmetric key-token.

Use the Symmetric Key Import verb to import a key exported using the AES or DES algorithm, and the Symmetric Key Import2 verb to import a key exported using the HMAC algorithm.

Different methods are supported for formatting the output key. Not all of these methods are available for each supported source key-token. The **AESKW** key-formatting method uses an AES EXPORTER key-encrypting key to wrap the output key before returning it in an external variable-length symmetric key-token. The other key formatting methods each use a different scheme to format the key before it is enciphered using an asymmetric RSA public-key. The formatted and enciphered key is returned as an opaque data buffer, and is not in a key token.

Format

The format of CSNDSYX.

```
CSNDSYX(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              rule_array_count,
              rule_array,
              source_key_identifier_length,
              source_key_identifier,
              transport_key_identifier_length,
              transport_key_identifier,
              enciphered_key_length,
              enciphered_key)
```

Parameters

The parameters for CSNDSYX.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1, 2, or 3.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. Each keyword is left-aligned in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage. The *rule_array* keywords are described in Table 80.

Table 80. Keywords for Symmetric Key Export control information

Keyword	Description
Algorithm (One, optional)	
AES	Export an AES key. If *source_key_identifier* is a variable-length symmetric key token or label, only the **PKOAEP2** and **AESKW** key formatting methods are supported.
DES	Export a DES key. This is the default.
HMAC	Export an HMAC key. Only the **PKOAEP2** and **AESKW** key formatting methods are supported.
Recovery method (One, required)	
AESKW	Specifies that the key is to be formatted using **AESKW** and placed in an external variable length CCA token. The *transport_key_identifier* must be an AES EXPORTER. This rule is not valid with the **DES** algorithm keyword or with AES DATA (version X'04') keys.
AESKWCV	Specifies to return the key formatted using the ANS X9.102 AESKW method creating a special variable-length symmetric key-token whose key type is DESUSECV. The AESKW payload that contains the DES key is encrypted by the AES EXPORTER key-encrypting key provided as the transport key, and returned in an external variable-length symmetric key-token with a token algorithm of DES. The DES control vector (with its key form bits masked to binary zeros to mask the key length) along with other significant key-token information is included in the associated data section of the variable-length symmetric key-token. Valid only with the DES algorithm, and only for a DES key in a fixed-length symmetric key-token.
PKCSOAEP	Specifies using the method found in RSA DSI PKCS #1V2 OAEP. See "PKCS #1 hash formats" on page 950. The default hash method is **SHA-1**. Use the **SHA-256** keyword for the SHA-256 hash method. Use the **SHA-384** keyword for the SHA-384 hash method. Use the **SHA-512** keyword for the SHA-512 hash method.
PKCS-1.2	Specifies using the method found in RSA DSI PKCS #1 block type 02 to recover the symmetric key. In the RSA PKCS #1 v2.0 standard, RSA terminology describes this as the RSAES-PKCS1-v1_5 format. See "PKCS #1 hash formats" on page 950.
PKOAEP2	Specifies that the key is formatted as defined in the RSA PKCS #1 v2.1 standard for the RSAES-OAEP encryption mechanism. Valid only with algorithm **HMAC**. This keyword was introduced with CCA 4.1.0. See "PKCS #1 hash formats" on page 950.
ZERO-PAD	The clear key is right-aligned in the field provided, and the field is padded to the left with zeros up to the size of the RSA encryption block (which is the modulus length).
Hash method (One, optional for **PKCSOAEP**, required for **PKOAEP2**. Not valid with any other key formatting method)	
SHA-1	Specifies to use the SHA-1 hash method to calculate the OAEP message hash. Valid only with key-formatting methods **PKCSOAEP** or **PKOAEP2**. This is the default for **PKCSOAEP**.
SHA-256	Specifies to use the SHA-256 hash method to calculate the OAEP message hash. Valid only with key-formatting methods **PKCSOAEP** or **PKOAEP2**.
SHA-384	Specifies to use the SHA-384 hash method to calculate the OAEP message hash. Valid only with key-formatting method **PKOAEP2**.
SHA-512	Specifies to use the SHA-512 hash method to calculate the OAEP message hash. Valid only with key-formatting method **PKOAEP2**.

source_key_identifier_length

Direction: Input
Type: Integer

The length of the *source_key_identifier* parameter. The minimum size is 64 bytes. The maximum size is 725 bytes.

source_key_identifier

Direction: Input
Type: String

The label or internal token of a secure AES DATA (version X'04'), DES DATA, or variable-length symmetric key token to encrypt under the supplied RSA public key or AES EXPORTER key. The key in the key identifier must match the algorithm in the *rule_array*. DES is the default algorithm.

transporter_key_identifier_length

Direction: Input
Type: Integer

The length of the *transporter_key_identifier* parameter. The maximum size is 3500 bytes for an RSA key token or 725 for an AES EXPORTER key token. The length must be 64 if *transporter_key_identifier* is a label.

transporter_key_identifier

Direction: Input
Type: String

A pointer to a string variable containing an RSA public key token, AES EXPORTER token, or label of the key to protect the exported symmetric key.

When the **AESKW** key formatting method is specified, this parameter must be an AES EXPORTER key token or label with the EXPORT bit on in the key-usage field. Otherwise, this parameter must be an RSA public key token or label.

enciphered_key_length

Direction: Input/Output
Type: Integer

The length of the *enciphered_key* parameter. This variable is updated with the actual length of the *enciphered_key* generated. The maximum size you can specify in this parameter is 900 bytes.

enciphered_key

Direction: Output
Type: String

A pointer to a string variable containing the key after it has been formatted and enciphered by the transport key. The enciphered key is returned either as an opaque data buffer or in an external variable-length symmetric key-token. For key-formatting method **PKOAEP2**, the key token has no key verification pattern.

Restrictions

The restrictions for CSNDSYX.

- The RSA public-key modulus size (key length) is limited by the Function Control Vector (FCV) to accommodate potential government export and import regulations.
- Retained keys are not supported.
- The maximum public exponent is 17 bits for any key that has a modulus greater than 2048 bits.

Required commands

The required commands for CSNDSYX.

This verb requires the following commands to be enabled in the active role based on the key-formatting method and the algorithm:

Key-formatting method	Algorithm	Offset	Command
AESKW	**AES**	X'0327'	**Symmetric Key Export - AESKW**
AESKWCV (Release 4.4 or later)	**DES**	X'02B3'	**Symmetric Key Export - AESKWCV**
PKOAEP2	**AES**	X'00FC'	**Symmetric Key Export - AES, PKOAEP2**
PKOAEP2	**HMAC**	X'00F5'	**Symmetric Key Export - HMAC, PKOAEP2**
PKCSOAEP or **PKCS-1.2**	**AES**	X'0130'	**Symmetric Key Export - AES, PKCSOAEP, PKCS-1.2**
PKCSOAEP or **PKCS-1.2**	**DES**	X'0105'	**Symmetric Key Export - DES, PKCS-1.2**
ZERO-PAD	**AES**	X'0131'	**Symmetric Key Export - AES, ZERO-PAD**
ZERO-PAD	**DES**	X'023E'	**Symmetric Key Export - DES, ZERO-PAD**

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

To disallow the wrapping of a key with a weaker key-encrypting key, enable the **Prohibit weak wrapping - Transport keys** command (offset X'0328') in the active role. This command affects multiple verbs. See Chapter 22, "Access control points and verbs," on page 953.

To receive a warning when wrapping a key with a weaker key-encrypting key, enable the **Warn when weak wrap - Transport keys** command (offset X'032C') in the active role. The **Prohibit weak wrapping - Transport keys** command (offset X'0328') overrides this command.

Usage notes

The usage notes for CSNDSYX.

The hardware configuration sets the limit on the modulus size of keys for key management; thus, this verb will fail if the RSA key modulus bit length exceeds this limit.

The strength of the exporter key expected by Symmetric Key Export depends on the attributes of the key being exported. The resulting return code and reason code

when using an exporter KEK that is weaker depends on the **Disallow Weak Key Wrap** command (offset X'0328') and the **Warn when Wrapping Weak Keys** command (offset X'032C'):

- If the **Disallow Weak Key Wrap** command (offset X'0328') is disabled (the default), the key strength requirement is not enforced. Using a weaker key results in return code 0 with a nonzero reason code if the **Warn when Wrapping Weak Keys** command (offset X'032C') is enabled. Otherwise, a reason code of zero is returned.
- If the **Disallow Weak Key Wrap** (offset X'0328') access control point is enabled (using TKE), the key strength requirement will be enforced, and attempting to use a weaker key results in return code 8.

For AES DATA and AES CIPHER keys, the AES EXPORTER key must be at least as long as the key being exported to be considered sufficient strength.

Note that wrapping an AES 192-bit key or an AES 256-bit key with any RSA key will always be considered a weak wrap.

For HMAC keys, the AES EXPORTER must be sufficient strength as described in Table 81.

Table 81. AES EXPORTER strength required for exporting an HMAC key under an AES EXPORTER

Key-usage field 2 in the HMAC key contains	Minimum strength of AES EXPORTER to adequately protect the HMAC key
SHA-256, SHA-384, SHA-512	256 bits
SHA-224	192 bits
SHA-1	128 bits

If an RSA public key is specified as the *transporter_key_identifier*, the RSA key used must have a modulus size greater than or equal to the total PKOAEP2 message bit length (key size plus total overhead), as described in Table 82.

Table 82. Minimum RSA modulus strength required to contain a PKOAEP2 block when exporting an AES key

AES key size	Total message sizes (and therefore minimum RSA key size) when the hash method is:			
	SHA-1	SHA-256	SHA-384	SHA-512
128 bits	736 bits	928 bits	1184 bits	1440 bits
192 bits	800 bits	992 bits	1248 bits	1504 bits
256 bits	800 bits	1056 bits	1312 bits	1568 bits

For AES keys, the AES EXPORTER must be sufficient strength as described in Table 83.

Table 83. Minimum RSA modulus length to adequately protect an AES key

AES key to be exported	Minimum strength of RSA wrapping key to adequately protect the AES key
AES 128	3072 bits
AES 192	7860 bits

Table 83. Minimum RSA modulus length to adequately protect an AES key (continued)

AES key to be exported	Minimum strength of RSA wrapping key to adequately protect the AES key
AES 256	15360 bits

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDSYXJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNDSYXJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger source_key_identifier_length,
           byte[]            source_key_identifier,
           hikmNativeInteger transporter_key_identifier_length,
           byte[]            transporter_key_identifier,
           hikmNativeInteger enciphered_key_length,
           byte[]            enciphered_key);
```

Symmetric Key Export with Data (CSNDSXD)

Use the Symmetric Key Export with Data verb to export a symmetric key, along with some application supplied data, encrypted using an RSA key.

The clear key data is copied into the provided data field at the offset specified with the **data_offset**. Then it is encrypted using the PKCS-1.5 block type 2 formatting algorithm.

Format

The format of CSNDSXD.

```
CSNDSXD(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            source_key_identifier_length,
            source_key_identifier,
            data_length,
            data_offset,
            data,
            RSA_public_key_identifier_length,
            RSA_public_key_identifier,
            RSA_enciphered_key_length,
            RSA_enciphered_key)
```

Parameters

The parameters for CSNDSXD.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. The value must be 2.

rule_array

Direction: Input
Type: String array

A pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The **rule_array** keywords are described in Table 84.

Table 84. Keywords for Symmetric Key Export with Data control information

Keyword	Description
Algorithm (One required)	
AES	The key specified in **source_key_identifier** is an AES key.
DES	The key specified in **source_key_identifier** is a DES key.
Key Formatting method (One required)	
PKCS–EXT	Copy the clear key data (length determined by the key length in the source key token) into the provided data field at the offset specified in the **data_offset** parameter. Then encrypt the key using the PKCS-1.5 block type 2 formatting algorithm.

source_key_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes in the **source_key_identifier** variable. This value is 64 when a label is supplied. When the key identifier is a key token, the value is the length of the token. For DES keys, the value must be 64. For AES keys, the maximum value is 725.

source_key_identifier

Direction: Input
Type: String

An internal key token, or the label of an operational symmetric key-token record in AES or DES key storage containing an operational AES or DES key token that is to be exported. If the key is a DES key, bit 17 of the control vector must be equal to '1'b (XPORT-OK). The key must have a control vector of DATAC or DKYGENKY with subtype DKYL0, unless the **Allow Symmetric Key Export with Data Special** access control point is enabled.

If the AES key is in a fixed length key token, no control vector checking is needed. If the AES key is in a variable length token, the key type must be CIPHER. If the key type is not CIPHER, an access control point **Allow Symmetric Key Export with Data Special** must be enabled. If the key is an AES key, the key management field in the key must allow export by RSA keys and by unauthenticated asymmetric keys.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

data_length

Direction: Input
Type: Integer

The length of the **data** parameter in bytes. The maximum value is the length of the modulus (in bytes) of the **RSA_public_key_identifier** minus 11. The overall maximum value is 501.

data_offset

Direction: Input
Type: Integer

The offset from the start of data where the clear DES or AES key is to be copied. The maximum value is **data_length** minus the key length of the clear source key.

data

Direction: Input
Type: String

The clear data. The deciphered key from parameter **source_key_identifier** is copied into this data at the specified offset, and then encrypted with the key from parameter **RSA_public_key_identifier**.

RSA_public_key_identifier_length

Direction: Input
Type: Integer

The length of the **RSA_public_key_identifier** field in bytes. This value is 64 when a label is supplied. When the key identifier is a key token, the value is the length of the token. The maximum value is 3500.

RSA_public_key_identifier

Direction: Input
Type: String

A PKA96 RSA internal or external key-token with the RSA public key of the remote node that imports the exported key.

RSA_enciphered_key_length

Direction: Input
Type: Integer

The length of the **RSA_enciphered_key** field in bytes. On output, the variable is updated with the actual length of the **RSA_enciphered_key** parameter. The maximum length is 512.

RSA_enciphered_key

Direction: Output
Type: String

The exported RSA-enciphered key.

Restrictions

The restrictions for CSNDSXD.

None.

Required commands

The required commands for CSNDSXD.

The Symmetric Key Export with Data verb requires the **Symmetric Key Export with Data** command (offset X'02B5') to be enabled in the active role.

In addition, the verb requires the following commands to be enabled in the active role based on the key-formatting method and the token algorithm:

Table 85. Required commands for the Symmetric Key Export with Data verb.

Key-formatting method	Algorithm	Offset	Command
PKCS-EXT	AES	X'0130'	**Symmetric Key Export - AES, PKCSOAEP, PKCS-1.2**
PKCS-EXT	DES	X'0105'	**Symmetric Key Export - DES, PKCS-1.2**

The **Symmetric Key Export with Data - Special** command (offset X'02B6') affects which key types are allowed for the source key token. When offset X'02B6' is enabled in the active role, any key type can be used. When it is not enabled in the active role, the following rules apply:

- Token algorithm AES:

  If the source AES key is in a fixed-length symmetric key-token, the key is always allowed. If the source AES key is in a variable-length symmetric key-token, the key type must be CIPHER.
- Token algorithm DES:

  The source DES key must be in a fixed-length symmetric key-token and have one of the following:
  - A control vector with bit 61 = B'1' (NOT-CCA)
  - A key type of DATAC
  - A key type of DKYGENKY with subtype DKYL0

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNDSXD.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDSXDJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNDSXDJ(
           hikmNativeInteger    return_code,
           hikmNativeInteger    reason_code,
           hikmNativeInteger    exit_data_length,
           byte[]               exit_data,
           hikmNativeInteger    rule_array_count,
           byte[]               rule_array,
           hikmNativeInteger    source_key_identifier_length,
           byte[]               source_key_identifier,
           hikmNativeInteger    data_length,
           hikmNativeInteger    data_offset,
           byte[]               data,
           hikmNativeInteger    RSA_public_key_identifier_length,
           byte[]               RSA_public_key_identifier,
           hikmNativeInteger    RSA_enciphered_key_length,
           byte[]               RSA_enciphered_key);
```

Symmetric Key Generate (CSNDSYG)

Use the Symmetric Key Generate verb to generate an AES or DES DATA key and return the key in two forms: enciphered under the master key and encrypted under an RSA public key.

You can import the RSA public key encrypted form by using the Symmetric Key Import or Symmetric Key Import2 verbs at the receiving node.

Also use the Symmetric Key Generate verb to generate any DES importer or exporter key-encrypting key encrypted under a RSA public key according to the PKA92 formatting structure. See "PKA92 key format and encryption process" on page 948 for more details about PKA92 formatting.

Format

The format of CSNDSYG.

```
CSNDSYG(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_encrypting_key_identifier,
             RSA_public_key_identifier_length,
             RSA_public_key_identifier,
             local_enciphered_key_identifier_length,
             local_enciphered_key_identifier,
             RSA_enciphered_key_length,
             RSA_enciphered_key)
```

Parameters

The parameters for CSNDSYG.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input

Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1 - 7.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. The recovery method is the method to use to recover the symmetric key. Each keyword is left-aligned in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage. The *rule_array* keywords are described in Table 86.

Table 86. Keywords for Symmetric Key Generate control information

Keyword	Description
Algorithm (One, optional)	
AES	Specifies to generate an AES key.
DES	Specifies to generate a DES key. This is the default.
Key-formatting method (One required)	
PKA92	Specifies the key-encrypting key is to be encrypted under a PKA96 RSA public key according to the PKA92 formatting structure.
PKCSOAEP	Specifies to use the method found in RSA DSI PKCS #1V2 OAEP. Supported by the DES and AES algorithms. The default hash method is **SHA-1**. Use the **SHA-256** keyword for the SHA-256 hash method.
PKCS-1.2	Specifies the method found in RSA DSI PKCS #1 block type 02. In the RSA PKCS #1 v2.0 standard, RSA terminology describes this as the RSAES-PKCS1-v1_5 format.
ZERO-PAD	The clear key is right-aligned in the field provided, and the field is padded to the left with zeros up to the size of the RSA encryption block (which is the modulus length).
Key length (One, optional use with **PKA92**)	
SINGLE-R	Generates a key-encrypting key that has equal left and right halves allowing it to perform as a single-length key. Valid only for the recovery method of **PKA92**.
Key length (One, optional use with **PKCSOAEP**, **PKCS-1.2**, or **ZERO-PAD**)	
SINGLE, **KEYLN8**	Generates a single-length DES key. This is the default for DES keys.
DOUBLE	Generates a double-length DES key. Valid only for DES keys.
KEYLN16	Generates a double-length DES DATA key. This is the default for AES keys.
KEYLN24	Generates a triple-length DES DATA key. Valid only for AES keys
KEYLN32	Generates a 32-byte AES key. Valid only for AES keys
Encipherment method for the local enciphered copy of the key (One, optional for use with **PKCSOAEP**, **PKCS-1.2**, and **ZERO-PAD**)	
EX	The DES enciphered key is enciphered by an EXPORTER key that is provided through the *key_encrypting_key_identifier* parameter.
IM	The DES enciphered key is enciphered by an IMPORTER key that is provided through the *key_encrypting_key_identifier* parameter.
OP	The DES enciphered key is enciphered by the master key. The *key_encrypting_key_identifier* parameter is ignored. This is the default.
Key-wrapping method (One, optional)	

Table 86. Keywords for Symmetric Key Generate control information (continued)

Keyword	Description
USECONFG	This is the default. Specifies to wrap the key using the configuration setting for the default wrapping method. The default wrapping method configuration setting may be changed using the TKE. This keyword is ignored for AES keys.
WRAP-ENH	Use enhanced key wrapping method, which is compliant with the ANSI X9.24 standard.
WRAP-ECB	Use original key wrapping method, which uses ECB wrapping for DES key tokens and CBC wrapping for AES key tokens.
Translation control (Optional) This is valid only with key-wrapping method **WRAP-ENH** or with **USECONFG** when the default wrapping method is **WRAP-ENH**. This option cannot be used on a key with a control vector valued to binary zeros.	
ENH-ONLY	Specifies to restrict the key from being wrapped with the legacy wrapping method after it has been wrapped with the enhanced wrapping method. Sets bit 56 (ENH-ONLY) of the control vector to B'1'. This keyword was introduced with CCA 4.1.0.
Hash method (Optional). Valid only with keyword **PKCSOAEP**.	
SHA-1	Specifies to use the SHA-1 hash method to calculate the OAEP message hash. This is the default.
SHA-256	Specifies to use the SHA-256 hash method to calculate the OAEP message hash.

key_encrypting_key_identifier

Direction: Input/Output
Type: String

The label or internal token of a key-encrypting key. If the *rule_array* specifies IM, this DES key must be an IMPORTER. If the *rule_array* specifies EX, this DES key must be an EXPORTER.

RSA_public_key_identifier_length

Direction: Input
Type: Integer

The length of the *RSA_public_key_identifier* parameter. If the *RSA_public_key_identifier* parameter is a label, this parameter specifies the length of the label. The maximum size is 3500 bytes.

RSA_public_key_identifier

Direction: Input
Type: String

The token, or label, of the RSA public key to be used for protecting the generated symmetric key.

local_enciphered_key_identifier_length

Direction: Input/Output
Type: Integer

The length of the *local_enciphered_key_identifier*. This field is updated with the actual length of the *local_enciphered_key_identifier* that is generated. The maximum length is 3500 bytes. However, this value should be 64 as in current CCA practice a DES key-token or a key label is always a 64-byte structure.

local_enciphered_key_identifier

Direction: Input/Output
Type: String

A pointer to a string variable containing either a key name or a key token. The control vector for the local key is taken from the identified key token. On output, the generated key is inserted into the identified key token.

On input, you must specify a token type consistent with your choice of local-key encryption. If you specify IM or EX, you must specify an external key-token. Otherwise, specify an internal key-token or a null key-token.

When **PKCSOAEP**, **PKCS-1.2**, or **ZERO-PAD** is specified, a null key-token can be specified. In this case, an AES DATA or DES DATA key is returned. For an internal key (**OP**), a default AES DATA or DATA control-vector is returned in the key token. For an external key (**IM** or **EX**), the control vector is set to null.

RSA_enciphered_key_length

Direction: Input/Output
Type: Integer

The length of the *RSA_enciphered_key* parameter. This verb updates this with the actual length of the *RSA_enciphered_key* it generates. The maximum size is 3500 bytes.

RSA_enciphered_key

Direction: Input/Output
Type: String

A pointer to a string variable containing the generated RSA-enciphered key returned by the verb. If you specify **PKCSOAEP**, **PKCS-1.2**, or **ZERO-PAD**, on input specify a null key token. If you specify **PKA92**, on input specify an internal (operational) CCA DES key-token.

Restrictions

The restrictions for CSNDSYG.

None.

Required commands

The required commands for CSNDSYG.

This verb requires the following commands to be enabled in the active role based on the key-formatting method and the algorithm:

Key-formatting method	Algorithm	Offset	Command
PKCSOAEP or **PKCS-1.2**	**AES**	X'012C'	**Symmetric Key Generate - AES_ PKCSOAEP_ PKCS-1.2**
PKCSOAEP or **PKCS-1.2**	**DES**	X'023F'	**Symmetric Key Generate - DES_ PKCS-1.2**
ZERO-PAD	**AES**	X'012D'	**Symmetric Key Generate - AES_ ZERO-PAD**
ZERO-PAD	**DES**	X'023C'	**Symmetric Key Generate - DES_ ZERO-PAD**
PKA92	**DES**	X'010D'	**Symmetric Key Generate - DES_ PKA92**

The use of the **WRAP-ECB** or **WRAP-ENH** key-wrapping method keywords requires the **Symmetric Key Generate - Allow wrapping override keywords** command (offset X'013E') to be enabled.

The following access-control points are added beginning with Release 4.3:

- To disable the wrapping of a key with a weaker master key, the **Prohibit weak wrapping - Master keys** command (offset X'0333') must be enabled in the active role.
- To receive a warning when wrapping a key with a weaker master key, enable the **Warn when weak wrap - Master keys** command (offset X'0332') in the active role. The **Prohibit weak wrapping - Master keys** command overrides this command.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNDSYG.

The hardware configuration sets the limit on the modulus size of keys for key management; thus, this verb will fail if the RSA key modulus bit length exceeds this limit.

Specification of **PKA92** with an input NOCV key-encrypting key token is not supported.

Use the **PKA92** key-formatting method to generate a key-encrypting key. The verb enciphers one key copy using the key encipherment technique employed in the IBM Transaction Security System (TSS) 4753, 4755, and AS/400 cryptographic product PKA92 implementations (see "PKA92 key format and encryption process" on page 948). The control vector for the RSA-enciphered copy of the key is taken from an internal (operational) DES key token that must be present on input in the *RSA_enciphered_key* variable.

Only key-encrypting keys that conform to the rules for an OPEX case under the Key Generate verb are permitted. The control vector for the local key is taken from a DES key token that must be present on input in the *local_enciphered_key_identifier* variable. The control vector for one key copy must be from the EXPORTER class, while the control vector for the other key copy must be from the IMPORTER class.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDSYGJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNDSYGJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            key_encrypting_key_identifier,
           hikmNativeInteger RSA_public_key_identifier_length,
           byte[]            RSA_public_key_identifier,
           hikmNativeInteger local_enciphered_key_identifier_length,
           byte[]            local_enciphered_key_identifier,
           hikmNativeInteger RSA_enciphered_key_token_length,
           byte[]            RSA_enciphered_key_token);
```

Symmetric Key Import (CSNDSYI)

Use the Symmetric Key Import verb to import a symmetric AES DATA or DES DATA key enciphered under an RSA public key. The verb returns the key in operational form, enciphered under the master key.

This verb also supports import of a PKA92-formatted DES key-encrypting key under a PKA96 RSA public key.

Format

The format of CSNDSYI.

```
CSNDSYI(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             RSA_enciphered_key_length,
             RSA_enciphered_key,
             RSA_private_key_identifier_length,
             RSA_private_key_identifier,
             target_key_identifier_length,
             target_key_identifier)
```

Parameters

The parameters for CSNDSYI.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1 - 5.

rule_array

Direction: Input
Type: String array

The keyword that provides control information to the verb. The recovery method is the method to use to recover the symmetric key. The keyword is left-aligned in an 8-byte field and padded on the right with blanks. The *rule_array* keywords are described in Table 87.

Table 87. Keywords for Symmetric Key Import control information

Keyword	Description
Algorithm (One, optional)	
AES	Export an AES key.
DES	Export a DES key. This is the default.
Recovery method (One required)	

Table 87. Keywords for Symmetric Key Import control information (continued)

Keyword	Description
PKA92	Specifies the key-encrypting key is encrypted under a PKA96 RSA public key according to the PKA92 formatting structure.
PKCSOAEP	Specifies to use the method found in RSA DSI PKCS #1V2 OAEP. Supported by the DES and AES algorithms. The default hash method is SHA-1. Use the **SHA-256** keyword for the SHA-256 hash method.
PKCS-1.2	Specifies the method found in RSA DSI PKCS #1 block type 02. In the RSA PKCS #1 v2.0 standard, RSA terminology describes this as the RSAES-PKCS1-v1_5 format.
ZERO-PAD	The clear key is right-aligned in the field provided, and the field is padded to the left with zeros up to the size of the RSA encryption block (which is the modulus length).
Key-wrapping method (One, optional)	
USECONFG	This is the default. Specifies to wrap the key using the configuration setting for the default wrapping method. The default wrapping method configuration setting may be changed using the TKE. This keyword is ignored for AES keys.
WRAP-ENH	Specifies to wrap the key using the legacy wrapping method. This keyword is ignored for AES keys. This keyword was introduced with CCA 4.1.0.
WRAP-ECB	Specifies to wrap the key using the enhanced wrapping method. Valid only for DES keys. This keyword was introduced with CCA 4.1.0.
Translation control (Optional) This is valid only with key-wrapping method **WRAP-ENH** or with **USECONFG** when the default wrapping method is **WRAP-ENH**. This option cannot be used on a key with a control vector valued to binary zeros.	
ENH-ONLY	Specifies to restrict the key from being wrapped with the legacy wrapping method after it has been wrapped with the enhanced wrapping method. Sets bit 56 (ENH-ONLY) of the control vector to B'1'. This keyword was introduced with CCA 4.1.0.
Hash method (Optional). Valid only with keyword **PKCSOAEP**.	
SHA-1	Specifies to use the SHA-1 hash method to calculate the OAEP message hash. This is the default.
SHA-256	Specifies to use the SHA-256 hash method to calculate the OAEP message hash.

RSA_enciphered_key_length

Direction: Input
Type: Integer

The length of the *RSA_enciphered_key* parameter. The maximum size is 3500 bytes.

RSA_enciphered_key

Direction: Input
Type: String

The key to import, protected under an RSA public key. The encrypted key is in the low-order bits (right-aligned) of a string whose length is the minimum number of bytes that can contain the encrypted key. This string is left-aligned within the *RSA_enciphered_key* parameter.

RSA_private_key_identifier_length

Direction: Input
Type: Integer

The length of the *RSA_private_key_identifier* parameter. When the *RSA_private_key_identifier* parameter is a key label, this field specifies the length

of the label. The maximum size is 3500 bytes.

RSA_private_key_identifier

Direction: Input
Type: String

An internal RSA private key token or label whose corresponding public key protects the symmetric key.

target_key_identifier_length

Direction: Input/Output
Type: Integer

The length of the *target_key_identifier* parameter. This field is updated with the actual length of the *target_key_identifier* that is generated. The maximum length is 3500 bytes.

target_key_identifier

Direction: Input/Output
Type: String

This field contains the internal token of the imported symmetric key.

Except for **PKA92** processing, this verb produces a DATA key token with a key of the same length as that contained in the imported token.

Restrictions

The restrictions for CSNDSYI.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Required commands

The required commands for CSNDSYI.

This verb requires the following commands to be enabled in the active role based on the key-formatting method and the algorithm:

Key-formatting method	Algorithm	Offset	Command
PKA92 and DATA, MAC, MACVER, KEYGENKY, EXPORTER, or OKEYXLAT key	**DES**	X'0235'	**Symmetric Key Import - DES, PKA92 KEK**
PKCSOAEP or **PKCS-1.2**	**AES** **DES**	X'012E' X'0106'	**Symmetric Key Import - AES, PKCSOAEP, PKCS-1.2** **Symmetric Key Import - DES, PKCS-1.2**
ZERO-PAD	**AES** **DES**	X'012F' X'023D'	**Symmetric Key Import - AES, ZERO-PAD** **Symmetric Key Import - DES, ZERO-PAD**
WRAP-ECB or **WRAP-ENH**, when the default key-wrapping method setting does not match the keyword	**DES**	X'0144'	**Symmetric Key Import - Allow wrapping override keywords**

The following access control points control the use of weak transport keys (Release 4.2 or later):

- To disallow the import of a key wrapped with a weaker transport key, the **Symmetric Key Import2 - disallow weak import** command (offset X'032B') must be enabled in the active role.
- To disable the wrapping of a key with a weaker transport key, the **Prohibit weak wrapping - Transport keys** command (offset X'0328') must be enabled in the active role.
- To receive an informational message when wrapping a key with a weaker key-encrypting key, enable the **Warn when weak wrap - Transport keys** command (offset X'032C') in the active role. The **Prohibit weak wrapping - Transport keys** command overrides this command.

The following access control points control the use of weak master keys (Release 4.3 or later):

- To disable the wrapping of a key with a weaker master key, the **Prohibit weak wrapping - Master keys** command (offset X'0333') must be enabled in the active role.
- To receive a warning when wrapping a key with a weaker master key, enable the **Warn when weak wrap - Master keys** command (offset X'0332') in the active role. The **Prohibit weak wrapping - Master keys** command overrides this command.

Usage notes

The usage notes for CSNDSYI.

The hardware configuration sets the limit on the modulus size of keys for key management; thus, this verb will fail if the RSA key modulus bit length exceeds this limit.

Use of **PKA92** with an input NOCV key-encrypting key token is not supported.

During initialization of a CEX*C, an Environment Identifier (EID) of zero will be set in the coprocessor. This will be interpreted by the Symmetric Key Import verb to mean that environment identification checking is to be bypassed. Thus it is possible on a Linux on z Systems system for a key-encrypting key RSA-enciphered at a node (EID) to be imported at the same node.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDSYIJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNDSYIJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger RSA_enciphered_key_length,
           byte[]            RSA_enciphered_key,
           hikmNativeInteger RSA_private_key_identifier_length,
           byte[]            RSA_private_key_identifier,
           hikmNativeInteger target_key_identifier_length,
           byte[]            target_key_identifier);
```

Symmetric Key Import2 (CSNDSYI2)

Use the Symmetric Key Import2 verb to import a symmetric key that has been exported by the Symmetric Key Export verb into a variable-length symmetric key-token. The verb imports the key into an internal variable-length symmetric key-token, or, beginning with Release 4.4, into an external fixed-length DES key-token.

The enciphered input key to be imported can be one of the following, depending on what release is used:

- In all releases, the input key cant be an **HMAC** key that has been previously formatted using key-formatting method **PKOAEP2**.
- Beginning with Release 4.2, AES keys are supported, along with support for the AES token algorithm. With this support, the input key can also be an HMAC key that has been previously formatted using key-formatting method **AESKW**, provided that the operational AES key-encrypting key used to encipher the key is provided. Likewise, an AES key can either be in an external AES variable-length symmetric key-token enciphered under an AES key-encrypting key (**AESKW**), or an RSA public-key (**PKOAEP2**).
- Beginning with Release 4.4, DES keys are supported, along with support for the DES token algorithm. With this support, the input key must have a key type of DESUSECV. A DESUSECV key contains the control vector and other information necessary to recreate the original internal fixed-length DES key-token.

  When importing a DES key, the verb must decide whether to use the legacy ECB mode or the enhanced CBC mode when wrapping the key in the target key-token. New optional key-wrapping method keywords are added to select which key-wrapping method to use.

  Also when importing a DES key, a new optional translation control keyword allows the target key to be restricted to being wrapped only with the enhanced CBC method once it has been wrapped with the enhanced method.

Before importing a DES key (Release 4.4 or later), the verb must determine whether to wrap the target key in legacy ECB mode or in enhanced CBC mode. These factors influence the key-wrapping method used for the imported target key-token:

1. The first is the default internal key-token key-wrapping preference of the receiving system where the target key token will be created. The receiving system can be set to a preference to wrap internal key tokens in either ECB or Enhanced modes.
2. The second is the key-wrapping method used to wrap the original key which was exported from the originating system.
3. The third is the key-wrapping method used by the verb which is specified by an optional key-wrapping method keyword or by default.

Table 88 and Table 89 on page 317 show how the key-wrapping method will be determined for the target key, based on the previously explained factors.

Table 88. Symmetric Key Import2 key-wrapping method of target key when system default is ECB (Legacy)

Wrap method of original key that was exported	Key-wrapping method keyword	Key-wrapping method used for the imported target key

Table 88. Symmetric Key Import2 key-wrapping method of target key when system default is ECB (Legacy) (continued)

ECB (Legacy)	**USECONFG**	ECB (Legacy).
	WRAP-ECB	Under control of command **Symmetric Key Import2 - Allow wrapping override keywords** (offset X'02B9'): • If X'02B9' is enabled, ECB (Legacy). • If X'02B9' is not enabled, not authorized error is returned.
	WRAP-ENH	Under control of offset X'02B9': • If X'02B9' is enabled, Enhanced. In addition, if **ENH-ONLY** keyword, CV bit 56 is set to B'1' (ENH-ONLY). • If X'02B9' is not enabled, not authorized error is returned.
Enhanced with CV bit 56 = B'0' (not ENH-ONLY)	**USECONFG**	ECB (Legacy).
	WRAP-ECB	Under control of command offset X'02B9': • If X'02B9' is enabled, ECB (Legacy). • If X'02B9' is not enabled, not authorized error is returned.
	WRAP-ENH	Under control of offset X'02B9': • If X'02B9' is enabled, Enhanced. In addition, if **ENH-ONLY** keyword, CV bit 56 is set to B'1' (ENH-ONLY). • If X'02B9' is not enabled, not authorized error is returned.
Enhanced with CV bit 56 = B'1' (ENH-ONLY)	**USECONFG**	Control information in the key token conflicts with that in the rule array error is returned.
	WRAP-ECB	Under control of offset X'02B9': • If X'02B9' is enabled, control information in the key token conflicts with that in the rule array error is returned. • If X'02B9' is not enabled, not authorized error is returned.
	WRAP-ENH	Under control of offset X'02B9': • If X'02B9' is enabled, Enhanced and CV bit 56 is set to B'1' (ENH-ONLY). • If X'02B9' is not enabled, error not authorized is returned.
Note: Conversion of an original key-token wrapped in Enhanced mode to an imported target key-token wrapped in ECB mode reduces security for that key. In this case, a command to override the system key-wrapping default is required.		

Table 89. Symmetric Key Import2 key-wrapping method of target key when system default is CBC (Enhanced)

Wrap method of original key that was exported	**Key-wrapping method keyword**	**Key-wrapping method used for the imported target key**
ECB (Legacy)	**USECONFG**	Enhanced. In addition, if **ENH-ONLY** keyword, CV bit 56 is set to B'1' (ENH-ONLY).
	WRAP-ECB	Under control of offset X'02B9': • If X'02B9' is enabled, ECB (Legacy). • If X'02B9' is not enabled, not authorized error is returned.
	WRAP-ENH	Enhanced. In addition, if **ENH-ONLY** keyword, CV bit 56 is set to B'1' (ENH-ONLY).

Table 89. Symmetric Key Import2 key-wrapping method of target key when system default is CBC (Enhanced) (continued)

Enhanced with CV bit 56 = B'0' (not ENH-ONLY)	**USECONFG**	Enhanced. If **ENH-ONLY** keyword, CV bit 56 is set to B''1' (ENH-ONLY).
	WRAP-ECB	Under control of offset X'02B9': • If X'02B9' is enabled, ECB (Legacy). • If X'02B9' is not enabled, not authorized error is returned.
	WRAP-ENH	Enhanced. In addition, if **ENH-ONLY** keyword, CV bit 56 is set to B'1' (ENH-ONLY).
Enhanced with CV bit 56 = B'1' (ENH-ONLY)	**USECONFG**	Enhanced and CV bit 56 is set to B'1' (ENH-ONLY).
	WRAP-ECB	Under control of offset X'02B9': • If X'02B9' is enabled, control information in the key token conflicts with that in the rule array error is returned. • If X'02B9' is not enabled, not authorized error is returned.
	WRAP-ENH	Under control of offset X'02B9': • If X'02B9' is enabled, enhanced and CV bit 56 is set to B'1' (ENH-ONLY). • If X'02B9' is not enabled, not authorized error is returned.
Note: Conversion of an original key-token wrapped in ECB (Legacy) mode to an imported target key-token wrapped in Enhanced (CBC) mode improves security for that key.		

Format

The format of CSNDSYI2.

```
CSNDSYI2(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             enciphered_key_length,
             enciphered_key,
             transport_key_identifier_length,
             transport_key_identifier,
             key_name_length,
             key_name,
             target_key_identifier_length,
             target_key_identifier)
```

Parameters

The parameters for CSNDSYI2.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

The number of keywords you supplied in the *rule_array* parameter. This value must be 2.

rule_array

Direction: Input
Type: String array

The keywords that provide control information to the verb. The following table provides a list. The recovery method is the method to use to recover the symmetric key. The keywords must be 8 bytes of contiguous storage with the keyword left-aligned in its 8-byte location and padded on the right with blanks. The *rule_array* keywords are described in Table 90.

Table 90. Keywords for Symmetric Key Import2 control information

Keyword	Description
Algorithm (One, required)	
AES	The key being imported is an AES key.
DES	The key being imported is a DES key.
HMAC	The key being imported is an HMAC key. Only the **PKOAEP2** recovery method is supported.
Recovery method (One, required)	
AESKW	Specifies the enciphered key has been wrapped with the AESKW formatting method.
AESKWCV	Specifies that the key is to be formatted using AESKW and placed in a symmetric variable length CCA token of type DESUSECV. The **transport_key_identifier** must be an AES EXPORTER key. The DES control vector and other significant token information is in the associated data section of the variable length key token. Only valid with the DES token algorithm.
PKOAEP2	Specifies to format the key according to the method found in RSA DSI PKCS #1 v2.1 RSAES-OAEP documentation.
Key wrapping method (Optional, valid only for DES algorithm. The access control point **Symmetric Key Import2 – Allow wrapping override keywords** must be enabled to specify these keywords)	
USECONFG	This is the default. Specifies to wrap the key using the configuration setting for the default wrapping method. The default wrapping method configuration setting may be changed using the TKE. This keyword is ignored for AES keys.
WRAP-ENH	Specifies that the new enhanced wrapping method is to be used to wrap the key.
WRAP-ECB	Specifies that the original wrapping method is to be used.
Translation Control (Optional, valid only for enhanced wrapping)	
ENH-ONLY	Specify this keyword to indicate that the key once wrapped with the enhanced method cannot be wrapped with the original method. This restricts translation to the original method.

Note: There is no need for a hash method keyword, because the hash method is encoded in the external key-token carrying the encoded and encrypted payload.

enciphered_key_length

Direction: Input
Type: Integer

The length of the *enciphered_key* parameter. The maximum size is 900 bytes.

enciphered_key

Direction: Input
Type: String

The key to import, protected under either an RSA public key or an AES KEK. If the recovery method is **PKOAEP2**, the encrypted key is in the low-order bits

(right-aligned) of a string whose length is the minimum number of bytes that can contain the encrypted key. If the recovery method is **AESKW**, the encrypted key is an AES key or HMAC key in the external variable length key token.

transport_key_identifier_length

Direction: Input
Type: Integer

The length of the *transport_key_identifier* parameter. When the *transport_key_identifier* parameter is a key label, this field must be 64. The maximum size is 3500 bytes for an RSA private key, or 725 bytes for an AES IMPORTER KEK.

transport_key_identifier

Direction: Input
Type: String

An internal RSA private key token, internal AES IMPORTER KEK, or the 64-byte label of a key token whose corresponding key protects the symmetric key.

When the **AESKW** key formatting method is specified, this parameter must be an AES IMPORTER key with the IMPORT bit on in the key-usage field. Otherwise, this parameter must be an RSA private key.

key_name_length

Direction: Input
Type: Integer

The length of the *key_name* parameter for *target_key_identifier*. Valid values are 0 and 64.

key_name

Direction: Input
Type: String

A 64-byte key store label to be stored in the associated data structure of *target_key_identifier*.

target_key_identifier_length

Direction: Input/Output
Type: Integer

On input, the length in bytes of the buffer for the *target_key_identifier* parameter. The buffer must be large enough to receive the target key token. The maximum value is 725 bytes.

On output, the parameter will hold the actual length of the target key token.

target_key_identifier

Direction: Output
Type: String

This parameter contains the internal token of the imported symmetric key.

Restrictions

The restrictions for CSNDSYI2.

The exponent of the RSA public key must be odd.

Required commands

The required commands for CSNDSYI2.

The Symmetric Key Import2 verb requires the following commands to be enabled in the active role:

Key-formatting method keyword	Token algorithm keyword	Offset	Command
AESKW (Rel. 4.2 or later)	**AES** or **HMAC**	X'0329'	**Symmetric Key Import2 - AESKW**
AESKWCV (Release 4.4 or later)	**DES**	X'02B4'	**Symmetric Key Import2 - AESKWCV**
WRAP-ECB or **WRAP-ENH**, when the default key-wrapping method setting does not match the keyword	**DES**	X'02B9'	**Symmetric Key Import2 - Allow wrapping override keywords**
PKOAEP2	**AES** (Rel. 4.2 or later)	X'00FD'	**Symmetric Key Import2 - AES,PKOAEP2**
PKOAEP2	**HMAC**	X'00F4'	**Symmetric Key Import2 - HMAC,PKOAEP2**

To disallow the import of a key wrapped with a weaker transport key, the **Symmetric Key Import2 - disallow weak import** command (offset X'032B') must be enabled in the active role. This command affects multiple verbs. See Chapter 22, "Access control points and verbs," on page 953.

To receive a warning against the wrapping of a stronger key with a weaker key, the **Warn when weak wrap - Transport keys** command (offset X'032C') must be enabled in the active role. The **Symmetric Key Import2 - disallow weak import** command overrides this command.

To disable the wrapping of a key with a weaker master key, the **Prohibit weak wrapping - Master keys** command (offset X'0333') must be enabled in the active role.

To receive a warning when wrapping a key with a weaker master key, enable the **Warn when weak wrap - Master keys** command (offset X'0332') in the active role. The **Prohibit weak wrapping - Master keys** command overrides this command.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNDSYI2.

This is the message layout used to encode the key material exported with the PKOAEP2 formatting method.

Table 91. PKCS#1 OAEP encoded message layout (PKOAEP2)

Field	Size	Value
Hash field	32 bytes	SHA-256 hash of associated data section in the source key identifier
Key bit length	2 bytes	Variable
Key material	length in bytes of the key material (rounded up to the nearest byte)	Variable

Hash field
The associated data for the HMAC variable length token is hashed using SHA-256.

Key bit length
A 2-byte key bit length field.

Key material
The key material is padded to the nearest byte with '0' bits.

The hardware configuration sets the limit on the modulus size of keys for key management; thus, this verb will fail if the RSA key modulus bit length exceeds this limit.

Specification of **PKA92** with an input NOCV key-encrypting key token is not supported.

During initialization of a CEX*C, an Environment Identifier (EID) of zero is set in the coprocessor. This is interpreted by the Symmetric Key Import2 verb to mean that environment identification checking is to be bypassed. Thus it is possible on a Linux on z Systems platform for a key-encrypting key RSA-enciphered at a node (EID) to be imported at the same node.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDSYI2J.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNDSYI2J(
          hikmNativeInteger return_code,
          hikmNativeInteger reason_code,
          hikmNativeInteger exit_data_length,
          byte[]            exit_data,
          hikmNativeInteger rule_array_count,
          byte[]            rule_array,
          hikmNativeInteger enciphered_key_length,
          byte[]            enciphered_key,
          hikmNativeInteger transport_key_identifier_length,
          byte[]            transport_key_identifier,
          hikmNativeInteger key_name_length,
          byte[]            key_name,
          hikmNativeInteger target_key_identifier_length,
          byte[]            target_key_identifier);
```

Unique Key Derive (CSNBUKD)

The Unique Key Derive verb performs the key derivation process as defined in ANSI X9.24 Part 1.

The process derives keys from two values: the base derivation key and the derivation data:

- The base derivation key is the key from which the others are derived. This must be a KEYGENKY with the UKPT bit (bit 18) set to 1 in the Control Vector.
- The derivation data is used to make the derived key specific to a particular device and to a specific transaction from that device. The derivation data, called the Current Key Serial Number (CKSN), is the 80-bit concatenation of the device's 59-bit Initial Key Serial Number value and the 21-bit value of the current encryption counter which the device increments for each new transaction.

The Initial Pin Encryption Key (IPEK) is derived from the base derivation key and the initial derivation data. Specify the K3IPEK rule array keyword to return the IPEK.

Rule array keywords determine the types and number of keys derived on a particular call. See the Rule Array parameter description for more information.

Output keys are wrapped using the mode configured as the default wrapping mode, either enhanced wrapping mode (WRAP-ENH) or original ECB wrapping mode (WRAP-ECB).

Format

The format of CSNBUKD.

```
CSNBUKD(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             base_derivation_key_identifier_length,
             base_derivation_key_identifier,
             derivation_data_length,
             derivation_data,
             generated_key_identifier1_length,
             generated_key_identifier1,
             generated_key_identifier2_length,
             generated_key_identifier2,
             generated_key_identifier3_length,
             generated_key_identifier3,
             transport_key_identifier_length,
             transport_key_identifier,
             reserved1_length,
             reserved1,
             reserved2_length,
             reserved2,
             reserved3_length,
             reserved3,
             reserved4_length,
             reserved4,
             reserved5_length,
             reserved5)
```

Parameters

The parameter definitions for CSNBUKD.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. Values are in the range 1 - 6.

rule_array

Direction: Input
Type: String array

An array of 8-byte keywords providing the processing control information to the verb. The keywords must be in contiguous storage with each of the keywords left-justified in its own 8-byte location and padded on the right with blanks. The **rule_array** keywords are described in Table 92.

Table 92. Keywords for Unique Key Derive control information

Keyword	Description
Algorithm (One, optional. The default is DES.)	
DES	Specifies that the keys to be generated are DES (Triple DES) keys. All input skeleton tokens must be DES tokens and all generated output tokens are DES tokens.
Token output type (One, required for K3IPEK.)	
TDES-TOK	Specifies that the output IPEK should be wrapped by the TDES transport key and returned in an external TDES token.
TR31-TOK	Specifies that the output IPEK should be wrapped by the TDES transport key and returned in a TR-31 key block.
Key wrapping method (One, optional. The default is USECONFG.) The access control point **Unique Key Derive – Override Default Wrapping Method** must be enabled to specify these keywords.	
USECONFG	Specifies to wrap the key using the configuration setting for the default wrapping method. The default wrapping method configuration setting may be changed using the TKE. This keyword is ignored for AES keys.
WRAP-ECB	Specifies to wrap the key using the original wrapping method.
WRAP-ENH	Specifies to wrap the key using the enhanced wrapping method.
Output key selection keywords (One required, up to 3 can be specified.) Neither the PIN-DATA nor the K3IPEK keyword can be specified with any other Output Key Selection keywords. Any combination of the other keywords (K1DATA, K2MAC, and K3PIN) can be specified, enabling a program to produce up to 3 different output keys with one call.	
K1DATA	The returned key type for this keyword is a DATA ENCRYPTION key. This is the output key selection keyword for the **generated_key_identifier1_length** and **generated_key_identifier1** parameters. The output value **generated_key_identifier1** is created and is a data encryption key. The skeleton token provided in that parameter on input must be one of the permitted data encryption key types for this callable service. For valid values see Table 93 on page 328.
K2MAC	The returned key type for this keyword is a MAC key. This is the output key selection keyword for the **generated_key_identifier2_length** and **generated_key_identifier2** parameters. The output value **generated_key_identifier2** is created and is a MAC key. The skeleton token provided in that parameter on input must be one of the permitted MAC key types for this callable service. For valid values, see Table 93 on page 328.

Table 92. Keywords for Unique Key Derive control information (continued)

Keyword	Description
K3PIN	The returned key type for this keyword is a PIN key. This is an output key selection keyword for the generated_key_identifier3_length and generated_key_identifier3 parameters. The output value **generated_key_identifier3** is created and is a PIN key. The skeleton token provided in that parameter on input must be one of the permitted PIN key types for this callable service. For valid values see Table 93 on page 328.
K3IPEK	The returned key for this keyword is the IPEK. This is an output key selection keyword for the generated_key_identifier3_length and generated_key_identifier3 parameters. The output value generated_key_identifier3 will be created and will be the initial PIN encryption key wrapped by the TDES transport key and returned in an external symmetric token or TR-31 key block as indicated by the token output type keyword. The skeleton token provided in that parameter on input must be one of the permitted PIN key types for this callable service. For valid values see Table 93 on page 328. This keyword may not be combined with any other output key selection keyword.
PIN-DATA	The returned key type for this keyword is a PIN key, which is returned in a DATA key token. This is an output key selection keyword for the **generated_key_identifier3_length** and **generated_key_identifier3** parameters. The output value **generated_key_identifier3** is created and will be a DATA key. The skeleton token provided in that parameter on input must be one of the permitted "PIN key with rule keyword PIN-DATA" key types for this callable service. For valid values, see Table 93 on page 328. To use this option: • Control Vector bit 61 (Not-CCA) is set to a B'1'. • Access control point **Unique Key Derive — Allow PIN-DATA processing** must be enabled.

base_derivation_key_identifier_length

Direction: Input
Type: Integer

Length of the **base_derivation_key_identifier** parameter in bytes. This value must be 64.

base_derivation_key_identifier

Direction: Input/Output
Type: String

The base derivation key is the key from which the operational keys are derived using the DUKPT algorithms defined in ANSI X9.24 Part 1. The base derivation key must be an internal key token or the label of an internal key token containing a double-length KEYGENKY key with the UKPT bit (bit 18) set to B'1' in the control vector.

derivation_data_length

Direction: Input
Type: Integer

Length of the **derivation_data** parameter in bytes. This value must be 10.

derivation_data

Direction: Input
Type: String

The derivation data is an 80-bit (10-byte) string that contains the Current Key Serial Number (CKSN) of the device concatenated with the 21-bit value of the current Encryption Counter which the device increments for each new transaction.

generated_key_identifier1_length

Direction: Input/Output
Type: Integer

Length of the **generated_key_identifier1** parameter in bytes. Values are 0 and 64.

generated_key_identifier1

Direction: Input/Output
Type: String

On input, this parameter must be a DES Data encryption key token or a skeleton token of a DES Data encryption key, with one of the Data encryption control vectors as shown in Table 93 on page 328.

On output, **generated_key_identifier1** contains the data encryption token with the derived data encryption key.

generated_key_identifier2_length

Direction: Input/Output
Type: Integer

Length of the **generated_key_identifier2** parameter in bytes. Values are 0 and 64.

generated_key_identifier2

Direction: Input/Output
Type: String

On input, this must be a DES MAC key token or a skeleton token of a DES MAC key, with one of the MAC control vectors as shown in Table 93 on page 328.

On output, **generated_key_identifier2** contains the MAC token with the derived MAC key.

generated_key_identifier3_length

Direction: Input/Output
Type: Integer

Length of the **generated_key_identifier3** parameter in bytes. When the rule array keyword is K3IPEK, the length must be at least 64 bytes. Otherwise, values are 0 and 64.

generated_key_identifier3

Direction: Input/Output
Type: String

The input and output values for this parameter depends on the keyword specified in the rule_array parameter. The rule_array keyword for the

generation_key_identifier3 parameter can be either PIN-DATA or K3PIN.

- When rule array keyword is PIN-DATA, input must be a data key token or skeleton token of a data key with one of the *PIN key with rule keyword PIN-DATA*-control-vectors as shown in Table 93 on page 328. On output, this parameter contains the data token with the derived PIN key.
- When rule array keyword is K3PIN, input must be a DES PIN key token or a skeleton token of a DES PIN key, with one of the PIN control vectors as shown in Table 93 on page 328. On output, this parameter contains the PIN token with the derived PIN key.
- When rule array keyword is K3IPEK, input must be a DES PIN key token or a skeleton token of a DES PIN key left-justified in the field, with one of the PIN control vectors as shown in Table 93 on page 328. On output, this parameter containd the TDES wrapped IPEK in an external symmetric key token or TR-31 key block.

transport_key_identifier_length

Direction: Input
Type: Integer

Length of the **transport_key_identifier** parameter in bytes. If the transport key identifier is not used, the length must be 0. Otherwise, the length must be 64.

transport_key_identifier

Direction: Input/Output
Type: String

If the K3IPEK keyword is specified, the **transport_key_identifier** contains the label or key token for the key encrypting key to be used to wrap the IPEK. The transport key must be a DES EXPORTER KEK. Otherwise this field is ignored.

reserved1_length

Direction: Input
Type: Integer

This parameter must be zero.

reserved1

Direction: Ignored
Type: String

This parameter is ignored.

reserved2_length

Direction: Input
Type: Integer

This parameter must be zero.

reserved2

Direction: Ignored
Type: String

This parameter is ignored.

reserved3_length

Direction: Input
Type: Integer

This parameter must be zero.

reserved3

Direction: Ignored
Type: String

This parameter is ignored.

reserved4_length

Direction: Input
Type: Integer

This parameter must be zero.

reserved4

Direction: Ignored
Type: String

This parameter is ignored.

reserved5_length

Direction: Input
Type: Integer

This parameter must be zero.

reserved5

Direction: Ignored
Type: String

This parameter is ignored.

Restrictions

The restrictions for CSNBUKD.

Table 93 shows the valid skeleton tokens depending on the key type to be derived.

Table 93. Valid Control Vectors for Derived Keys.

Key to be derived	Supported key types in the skeleton token		
Data encryption key	CIPHER	00 03 71 00 03 41 00 00	00 03 71 00 03 21 00 00
	ENCIPHER	00 03 60 00 03 41 00 00	00 03 60 00 03 21 00 00
	DECIPHER	00 03 50 00 03 41 00 00	00 03 50 00 03 21 00 00
Message authentication code(MAC)	MAC	00 05 4D 00 03 41 00 00	00 05 4D 00 03 21 00 00
	MACVER	00 05 44 00 03 41 00 00	00 05 44 00 03 21 00 00
PIN key	IPINENC	00 21 5F 00 03 41 00 00	00 21 5F 00 03 21 00 00
	OPINENC	00 24 77 00 03 41 00 00	00 24 77 00 03 21 00 00

Table 93. Valid Control Vectors for Derived Keys (continued).

Key to be derived	Supported key types in the skeleton token		
PIN key with rule keyword PIN-DATA	DATA PIN	00 00 7D 00 03 41 00 00	00 00 7D 00 03 21 00 00

Note that the following bits of the control vector are not checked and may have a value of either 0 or 1:

- Bit 17 - Export control
- Bit 56 – Enhanced wrapping control
- Bit 57 – TR-31 export control
- Bits 4 and 5 – UDX

Additional control vector bit that is not checked for PIN key with rule keyword PIN-DATA:

- Bit 61 - Not-CCA

Required commands

The required commands for CSNBUKD.

The Unique Key Derive verb requires the **Unique Key Derive** command (offset X'01C8') to be enabled in the active role.

In addition, these commands are required to be enabled in the active role, depending on the rule-array keyword or keywords:

Table 94. Required commands for the Symmetric Key Export with Data verb

Rule-array keyword	Offset	Command
K3IPEK	X'0335'	**Unique Key Derive - K3IPEK**
PIN-DATA	X'01C9'	**Unique Key Derive - Allow PIN-DATA processing**
WRAP-ECB or **WRAP-ENH** and default key-wrapping method setting does not match keyword	X'01CA'	**Unique Key Derive - Override default wrapping**

The following access control points control the use of weak transport keys:

- To disallow the import of a key wrapped with a weaker transport key, the **Symmetric Key Import2 - disallow weak import** command (offset X'032B') must be enabled in the active role.
- To receive a warning against the wrapping of a key with a weaker key, the **Warn when weak wrap - Transport keys** command (offset X'032C') must be enabled in the active role. The **Symmetric Key Import2 - disallow weak import** command overrides this command.

The following access control points control the use of weak master keys:

- To disable the wrapping of a key with a weaker master key, the **Prohibit weak wrapping - Master keys** command (offset X'0333') must be enabled in the active role.

- To receive a warning when wrapping a key with a weaker master key, enable the **Warn when weak wrap - Master Keys** command (offset X'0332') in the active role. The **Prohibit weak wrapping - Master keys** command overrides this command.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBUKD.

Table 95 indicates the variants used for each output key type to be derived.

Table 95. Derivation variants

Key type	DUKPT derivation variant	DUKPT key usage description
IPINENC or OPINENC PIN key (using PIN-DATA rule array keyword)	00000000000000FF 00000000000000FF	PIN Encryption
MAC	000000000000FF00 000000000000FF00	MAC, request or both ways
MACVER	00000000FF000000 00000000FF000000	MAC, response only
CIPHER ENCIPHER	0000000000FF0000 0000000000FF0000	Data Encryption, request or both ways
DECIPHER	000000FF00000000 000000FF00000000	Data Encryption, response only

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBUKDJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBUKDJ(
          hikmNativeInteger    return_code,
          hikmNativeInteger    reason_code,
          hikmNativeInteger    exit_data_length,
          byte[]               exit_data,
          hikmNativeInteger    rule_array_count,
          byte[]               rule_array,
          hikmNativeInteger    base_derivation_key_identifier_length,
          byte[]               base_derivation_key_identifier,
          hikmNativeInteger    derivation_data_length,
          byte[]               derivation_data,
          hikmNativeInteger    generated_key_identifier1_length,
          byte[]               generated_key_identifier1,
          hikmNativeInteger    generated_key_identifer2_length,
          byte[]               generated_key_identifier2,
          hikmNativeInteger    generated_key_identifier3_length,
          byte[]               generated_key_identifier3,
          hikmNativeInteger    transport_key_identifier_length,
          byte[]               transport_key_identifier,
          hikmNativeInteger    reserved1_length,
          byte[]               reserved1,
```

```
|                                hikmNativeInteger    reserved2_length,
|                                byte[]               reserved2,
|                                hikmNativeInteger    reserved3_length,
|                                byte[]               reserved3,
|                                hikmNativeInteger    reserved4_length,
|                                byte[]               reserved4,
|                                hikmNativeInteger    reserved5_length,
|                                byte[]               reserved5);

|
```

Chapter 7. Protecting data

Use CCA to protect sensitive data stored on your system, sent between systems, or stored off your system on magnetic tape.

To protect data, encipher it under a key. When you want to read the data, decipher it from ciphertext to plaintext form.

CCA provides Encipher and Decipher verbs to perform these functions. If you use a key to encipher data, you must use the same key to decipher the data. The Encipher and Decipher verbs use encrypted keys as input. You can also use clear keys, indirectly, by first using the Clear Key Import verb and then using the Encipher and Decipher verbs.

This topic describes the following verbs used for protecting data using DES or AES:

- "Decipher (CSNBDEC)" on page 335
- "Encipher (CSNBENC)" on page 339
- "Symmetric Algorithm Decipher (CSNBSAD)" on page 344
- "Symmetric Algorithm Encipher (CSNBSAE)" on page 350
- "Cipher Text Translate2 (CSNBCTT2)" on page 356

Modes of operation

Different algorithms are used to encipher or decipher DES data or keys and AES data or keys.

To encipher or decipher DES data or keys, CCA uses the U.S. National Institute of Standards and Technology (NIST) Data Encryption Standard (DES) algorithm, with single-length, double-length, or triple-length keys.

To encipher or decipher AES data or keys, CCA uses the U.S. National Institute of Standards and Technology (NIST) Advanced Encryption Standard (AES) algorithm, with 16-byte, 24-byte or 32-byte keys.

The Encipher and Decipher verbs operate in DES CBC (Cipher Block Chaining) mode.

Cipher Block Chaining (CBC) mode

The CBC mode uses an initial chaining vector (ICV) in its processing.

The CBC mode processes blocks of data only in exact multiples of the blocksize. The ICV is exclusive ORed with the first block of plaintext prior to the encryption step. The block of ciphertext just produced is exclusive-ORed with the next block of plaintext, and so on. You must use the same ICV to decipher the data. This disguises any pattern that may exist in the plaintext. CBC mode is the default for encrypting and decrypting data using the Encipher and Decipher verbs. "Ciphering methods" on page 932 describes the cipher processing rules in detail.

Electronic Code Book (ECB) mode

In ECB mode, each block of plaintext is separately enciphered and each block of the ciphertext is separately deciphered.

In other words, the encipherment or decipherment of a block is totally independent of other blocks.

Processing rules

You can use different types of processing rules for block chaining.

"Ciphering methods" on page 932 describes the cipher processing rules in detail.

CCA handles chaining for each block of data, from the first block until the last complete block of data in each Encipher or Symmetric Algorithm Encipher call. There are different types of *processing rules* you can choose for block chaining:

ANSI X9.23
Data is not necessarily in exact multiples of the block size. This processing rule pads the plaintext so the ciphertext produced is in exact multiples of the block size.

Cipher block chaining (CBC)
Data must be an exact multiple of the block size, and output will have the same length.

Cryptographic Unit Support Program (CUSP)
CBC mode (cipher block chaining) that is compatible with IBM's CUSP and PCF products. The data need not be in exact multiples of the block size. The ciphertext is the same length as the plaintext.

Electronic Code Book (ECB)
The data length must be a multiple of the block size. See "Electronic Code Book (ECB) mode."

Information Protection System (IPS)
CBC mode that is compatible with IBM's IPS product. The data need not be in exact multiples of the block size. The ciphertext is the same length as the plaintext.

PKCS-PAD
The data is padded on the right with between one and 16 bytes of pad characters, making ciphertext a multiple of the block size.

The resulting chaining value (except for ECB mode), after an Encipher or Symmetric Algorithm Encipher call, is known as an *output chaining vector (OCV).* When there are multiple cipher requests, the application can pass the OCV from the previous Encipher or Symmetric Algorithm Encipher call, as the input chaining vector (ICV) in the next Encipher or Symmetric Algorithm Encipher call. This produces chaining between successive calls, which is known as *record chaining*. CCA provides the ICV selection keyword CONTINUE in the *rule_array* parameter used to select record chaining with the CBC processing rule.

Triple DES encryption

Triple DES encryption uses a triple-length DATA key comprised of three 8-byte DES keys to encipher eight bytes of data.

To encipher the data it uses the following method:

- Encipher the data using the first key
- Decipher the result using the second key
- Encipher the second result using the third key

The procedure is reversed to decipher data that has been triple-DES enciphered:

- Decipher the data using the third key
- Encipher the result using the second key
- Decipher the second result using the first key

A variation of the triple-DES algorithm supports the use of a double-length DATA key comprised of two 8-byte DATA keys. In this method, the first 8-byte key is reused in the last encipherment step.

Due to export regulations, triple-DES encryption might not be available on your processor.

Decipher (CSNBDEC)

Use the Decipher verb to decipher data using the DES cipher block chaining mode.

CCA supports the following processing rules to decipher data. You choose the type of processing rule that the Decipher verb should use for block chaining.

ANSI X9.23
For cipher block chaining. The cipher text must be an exact multiple of eight bytes, but the plain text will be between 1 and 8 bytes shorter than the cipher text. The *text_length* will also be reduced to show the original length of the plain text.

Cipher Block Chaining (CBC)
The cipher text must be an exact multiple of eight bytes and the plain text will have the same length.

Cryptographic Unit Support Program (CUSP)
CBC mode (cipher block chaining) that is compatible with IBM's CUSP and PCF products. The data need not be in exact multiples of eight bytes. The cipher text is the same length as the plain text.

Information Protection System (IPS)
CBC mode (cipher block chaining) that is compatible with IBM's IPS product. The data need not be in exact multiples of eight bytes. The cipher text is the same length as the plain text.

The cipher block chaining (CBC) mode uses an initial chaining value (ICV) in its processing. The first eight bytes of cipher text is deciphered and then the ICV is XORed with the resulting eight bytes of data to form the first 8-byte block of plain text. Thereafter, the 8-byte block of cipher text is deciphered and XORed with the previous 8-byte block of cipher text until all the cipher text is deciphered.

The selection between single-DES decryption mode and triple-DES decryption mode is controlled by the length of the key supplied in the *key_identifier* parameter. If a single-length key is supplied, single-DES decryption is performed. If a double-length or triple-length key is supplied, triple-DES decryption is performed.

A different ICV could be passed on each call to the Decipher verb. However, the same ICV that was used in the corresponding Encipher verb must be passed.

Short blocks are text lengths of between one and seven bytes. A short block can be the only block. Trailing short blocks are blocks of between one and seven bytes that follow an exact multiple of eight bytes. For example, if the text length is 21, there are two 8-byte blocks and a trailing short block of five bytes. Because the DES processes text only in exact multiples of eight bytes, some special processing is required to decipher such short blocks.

These methods of treating short blocks and trailing short blocks do not increase the length of the cipher text compared to the length of the plain text. If the plain text was *padded* during encipherment, the length of the cipher text will always be an exact multiple of eight bytes.

CCA supports the ANSI X9.23 padding method.

Host CPU acceleration: CPACF

Only keys with a key type of DATA can be used successfully with the CPACF exploitation layer through this verb.

Specifically, a DATA key has a CV (Control Vector) of all X'00' bytes for all active bytes of the CV (eight bytes for 8-byte DES keys, 16 bytes for 16-byte DES keys, and 16 bytes for 24-byte DES keys).

For details about CPACF, see "CPACF support" on page 12.

Format

The format of CSNBDEC.

```
CSNBDEC(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             key_identifier,
             text_length,
             cipher_text,
             initialization_vector,
             rule_array_count,
             rule_array,
             chaining_vector,
             clear_text)
```

Parameters

The parameters for CSNBDEC.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

key_identifier

Direction: Input/Output
Type: String

A 64-byte string that is the internal key token containing the data-encrypting key or the label of a DES key storage record containing a data-encrypting key to be used for deciphering the data. If the key token or key label contains a single-length key, single-DES decryption is performed. If the key token or key

label contains a double-length or triple-length key, triple-DES decryption is performed.

Double length CIPHER and DECIPHER keys are also supported.

text_length

Direction: Input/Output
Type: Integer

On entry, you supply the length of the ciphertext. The maximum length of text is 214,783,647 bytes. A zero value for the *text_length* parameter is not valid. If the returned deciphered text (*clear_text* parameter) is a different length because of the removal of padding bytes, the value is updated to the length of the plain text.

The application program passes the length of the ciphertext to the verb. The verb returns the length of the plain text to your application program.

cipher_text

Direction: Input
Type: String

The text to be deciphered.

initialization_vector

Direction: Input
Type: String

The 8-byte supplied string for the cipher block chaining. The first block of the ciphertext is deciphered and XORed with the initial chaining vector (ICV) to get the first block of clear text. The input block is the next ICV. To decipher the data, you must use the same ICV used when you enciphered the data.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1, 2, or 3.

rule_array

Direction: Input
Type: String array

An array of 8-byte keywords providing the processing control information. The array is positional. The first keyword in the array is the processing rule. You choose the processing rule you want the verb to use for deciphering the data. The second keyword is the ICV selection keyword. The third keyword (or the second if the ICV selection keyword is allowed to default) is the encryption algorithm to use. The *rule_array* keywords are described in Table 96.

Table 96. Keywords for Decipher control information

Keyword	Description
Processing Rule (One, required)	

Table 96. Keywords for Decipher control information (continued)

Keyword	Description
CBC	Performs ANSI X3.102 cipher block chaining. The data must be a multiple of eight bytes. An OCV is produced and placed in the *chaining_vector* parameter. If the ICV selection keyword **CONTINUE** is specified, the CBC OCV from the previous call is used as the ICV for this call.
CUSP	Performs Cryptographic Unit Support Program (CUSP) cipher block chaining.
IPS	Performs Information Protection System (IPS) cipher block chaining.
X9.23	Deciphers with cipher block chaining and text length reduced to the original value. This is compatible with the requirements in ANSI standard X9.23. The ciphertext length must be an exact multiple of eight bytes. Padding is removed from the plaintext.
ICV Selection (One, optional)	
CONTINUE	This specifies taking the initialization vector from the output chaining vector (OCV) contained in the work area to which the *chaining_vector* parameter points. **CONTINUE** is valid only for the CBC processing rule.
INITIAL	This specifies taking the initialization vector from the *initialization_vector* parameter. **INITIAL** is the default value.
Encryption algorithm (Optional)	
DES	This specifies using the data encryption standard and ignoring the token marking.

"Ciphering methods" on page 932 describes the cipher processing rules in detail.

chaining_vector

Direction: Input/Output
Type: String

An 18-byte field CCA uses as a system work area. Your application program must not change the data in this string. The chaining vector holds the output chaining vector (OCV) from the caller. The OCV is the first eight bytes in the 18-byte string.

The direction is Output if the ICV selection keyword of the *rule_array* parameter is **INITIAL**. The direction is Input/Output if the ICV selection keyword of the *rule_array* parameter is **CONTINUE**.

clear_text

Direction: Output
Type: String

The field where the verb returns the deciphered text.

Restrictions

The restrictions for CSNBDEC.

This verb fails if the key token contains double or triple-length keys and triple-DES is not enabled.

Required commands

The required commands for CSNBDEC.

This verb requires the **Decipher - DES** command (offset X'000F') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBDEC.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBDECJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBDECJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           byte[]            key_identifier,
           hikmNativeInteger text_length,
           byte[]            ciphertext,
           byte[]            initialization_vector,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            chaining_vector,
           byte[]            plaintext);
```

Encipher (CSNBENC)

Use the Encipher verb to encipher data using the DES cipher block chaining mode.

CCA supports the following processing rules to encipher data. You choose the type of processing rule that the Encipher verb should use for the block chaining.

Cipher block chaining (CBC)
In exact multiples of eight bytes.

Cryptographic Unit Support Program (CUSP)
CBC mode (cipher block chaining) that is compatible with IBM's CUSP and PCF products. The data need not be in exact multiples of eight bytes. The cipher text is the same length as the plain text.

Information Protection System (IPS)
CBC mode (cipher block chaining) that is compatible with IBM's IPS product. The data need not be in exact multiples of eight bytes. The cipher text is the same length as the plain text.

ANSI X9.23
For block chaining not necessarily in exact multiples of eight bytes. This process rule pads the plain text so that cipher text produced is an exact multiple of eight bytes.

For more information about the processing rules, see Table 97 on page 342 and "Ciphering methods" on page 932.

The cipher block chaining (CBC) mode of operation uses an initial chaining vector (ICV) in its processing. The ICV is XORed with the first eight bytes of plain text before the encryption step and thereafter, the 8-byte block of cipher text just

produced is XORed with the next 8-byte block of plain text and so on. This disguises any pattern that might exist in the plain text.

The selection between single-DES encryption mode and triple-DES encryption mode is controlled by the length of the key supplied in the **key_identifier** parameter. If a single-length key is supplied, single-DES encryption is performed. If a double-length or triple-length key is supplied, triple-DES encryption is performed.

To nullify the CBC effect on the first 8-byte block, supply eight bytes of zero. However, the ICV might require zeros.

Cipher block chaining also produces a resulting chaining value called the output chaining vector (OCV). The application can pass the OCV as the ICV in the next encipher call. This results in *record chaining*.

Note that the OCV that results is the same, whether an Encipher or a Decipher verb was invoked, assuming the same text, ICV, and key were used.

Short blocks are text lengths of between one and seven bytes. A short block can be the only block. Trailing short blocks are blocks of between one and seven bytes that follow an exact multiple of eight bytes. For example, if the text length is 21, there are two 8-byte blocks, and a trailing short block of five bytes.

An alternative method is to pad the plain text and produce a cipher text that is longer than the plain text. The plain text can be padded with up to eight bytes using one of several padding methods. This padding produces a cipher text that is an exact multiple of eight bytes in length.

If the clear text is already a multiple of eight, the cipher text can be created using any processing rule.

Because of padding, the returned cipher text length is longer than the provided plain text. Therefore, the **text_length** parameter is modified. The returned cipher text field should be eight bytes longer than the length of the plain text to accommodate the maximum amount of padding.

Attention: If you lose the data-encrypting key under which the data (plain text) is enciphered, the data enciphered under that key (cipher text) cannot be recovered.

Host CPU acceleration: CPACF

Only keys with a key type of DATA can be used successfully with the CPACF exploitation layer through this verb.

Specifically, a DATA key has a CV (Control Vector) of all X'00' bytes for all active bytes of the CV (eight bytes for 8-byte DES keys, 16 bytes for 16-byte DES keys, and 16 bytes for 24-byte DES keys).

For details about CPACF, see “CPACF support” on page 12.

Format

The format of CSNBENC.

```
CSNBENC(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              key_identifier,
              text_length,
              clear_text,
              initialization_vector,
              rule_array_count,
              rule_array,
              pad_character,
              chaining_vector,
              cipher_text)
```

Parameters

The parameters for CSNBENC.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

key_identifier

Direction: Input/Output
Type: String

A 64-byte string that is the internal key token containing the data-encrypting key or the label of a DES key storage record containing the data-encrypting key, to be used for encrypting the data. If the key token or key label contains a single-length key, single-DES encryption is performed. If the key token or key label contains a double-length or triple-length key, triple-DES encryption is performed.

Single and double-length CIPHER and ENCIPHER keys are also supported.

text_length

Direction: Input/Output
Type: Integer

On entry, the length of the plain text (*clear_text* parameter) you supply. The maximum length of text is 214,783,647 bytes. A zero value for the *text_length* parameter is not valid. If the returned enciphered text (*cipher_text* parameter) is a different length because of the addition of padding bytes, the value is updated to the length of the ciphertext.

The application program passes the length of the plain text to the verb. This verb returns the length of the ciphertext to the application program.

clear_text

Direction: Input
Type: String

The text that is to be enciphered.

initialization_vector

Direction: Input

Type: String

The 8-byte supplied string for the cipher block chaining. The first eight bytes (or less) block of the data is XORed with the ICV and then enciphered. The input block is enciphered and the next ICV is created. You must use the same ICV to decipher the data.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1, 2, or 3.

rule_array

Direction: Input
Type: String array

An array of 8-byte keywords providing the processing control information. The array is positional. The first keyword in the array is the processing rule. You choose the processing rule you want the verb to use for enciphering the data. The second keyword is the ICV selection keyword. The third keyword (or the second if the ICV selection keyword is allowed to default to INITIAL) is the encryption algorithm to use. The *rule_array* keywords are described in Table 97.

Table 97. Keywords for Encipher control information

Keyword	Description
Processing Rule (1, required)	
CBC	Performs ANSI X3.102 cipher block chaining. The data must be a multiple of 8 bytes. An OCV is produced and placed in the *chaining_vector* parameter. If the ICV selection keyword CONTINUE is specified, the CBC OCV from the previous call is used as the ICV for this call.
CUSP	Performs Cryptographic Unit Support Program (CUSP) cipher block chaining.
IPS	Performs Information Protection System (IPS) cipher block chaining.
X9.23	Performs cipher block chaining with 1 - 8 bytes of padding. This is compatible with the requirements in ANSI X9.23. If the data is not in exact multiples of eight bytes, X9.23 pads the plaintext so the ciphertext produced is an exact multiple of 8 bytes. The plaintext is padded to the next multiple eight bytes, even if 8 bytes are added. An OCV is produced.
ICV Selection (1, optional)	
CONTINUE	Specifies to take the initialization vector from the output chaining vector (OCV) contained in the work area to which the *chaining_vector* parameter points. **CONTINUE** is valid only for the CBC processing rule.
INITIAL	Specifies to take the initialization vector from the *initialization_vector* parameter. **INITIAL** is the default value.
Encryption Algorithm (Optional)	
DES	Specifies to use the data encryption standard and to ignore the token marking.

"Ciphering methods" on page 932 describes the cipher processing rules in detail.

pad_character

Direction: Input
Type: Integer

An integer between 0 and 255 that is used as a padding character for the X9.23 process rule (*rule_array* parameter).

chaining_vector

Direction: Input/Output
Type: String

An 18-byte field CCA uses as a system work area. Your application program must not change the data in this string. The chaining vector holds the output chaining vector (OCV) from the caller. The OCV is the first eight bytes in the 18-byte string.

The direction is Output if the ICV selection keyword of the *rule_array* parameter is **INITIAL**.

The direction is Input/Output if the ICV selection keyword of the *rule_array* parameter is **CONTINUE**.

cipher_text

Direction: Output
Type: String

The enciphered text the verb returns. The length of the ciphertext is returned in the *text_length* parameter. The *cipher_text* could be eight bytes longer than the length of the *clear_text* field because of the padding that is required for some processing rules.

Restrictions

The restrictions for CSNBENC.

This verb will fail if the key token contains double-length or triple-length keys and triple-DES is not enabled.

Required commands

The required commands for CSNBENC.

This verb requires the **Encipher - DES** command (offset X'000E') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBENC.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBENCJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBENCJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
```

```
hikmNativeInteger exit_data_length,
byte[]            exit_data,
byte[]            key_identifier,
hikmNativeInteger text_length,
byte[]            plaintext,
byte[]            initialization_vector,
hikmNativeInteger rule_array_count,
byte[]            rule_array,
hikmNativeInteger pad_character,
byte[]            chaining_vector,
byte[]            ciphertext);
```

Symmetric Algorithm Decipher (CSNBSAD)

Use the Symmetric Algorithm Decipher verb to decipher data with an AES algorithm.

CCA supports the following processing rules to decipher data. You choose the type of processing rule that the verb should use for block chaining.

Cipher Block Chaining (CBC)
The plain text must be an exact multiple of eight bytes, and the cipher text will have the same length.

Electronic Code Book (ECB)
The plain text length must be a multiple of the block size.

PKCS-PAD
The plain text was padded on the right with 1 - 16 bytes of pad characters, making the padded text a multiple of the block size.

The AES key used to decipher the data can either be 16, 24, or 32 bytes (128, 192, or 256 bits) in length. The key can be supplied to the verb in any of three forms:

1. A clear text key consisting of only the key bytes, not contained in a key token.
2. A clear text key contained in an internal fixed or variable length AES key-token.
3. An encrypted key contained in an internal fixed or variable length AES key-token, where the key is wrapped (encrypted) with the AES master key.

To use this verb, specify:

- The *rule_array*:
  1. The algorithm identifier keyword **AES**, which is the only symmetric algorithm currently supported.
  2. An optional processing rule using keyword **CBC** (the default), **ECB**, or **PKCS-PAD**, which selects the decryption mode.
  3. An optional key rule using the keyword **KEY-CLR** (the default) or **KEYIDENT**, which selects whether the **key_identifier** parameter points to a 16-byte, 24-byte, or 32-byte clear key, or a key contained in a 64-byte AES key-token, either in application storage or a key label of such a key in key storage.
  4. An optional initial chaining value (ICV) selection using the keyword **INITIAL** (the default) or **CONTINUE**, which indicates whether it is the first or a subsequent request, and which parameter points to the initialization vector.
- For a key rule of **KEY-CLR**, a key identifier containing a 16-byte, 24-byte, or 32-byte clear key. For a key rule of **KEYIDENT**, a fixed-length or variable-length

internal AES key-token or the key label of such a key in AES key-storage. The key token can contain either a clear or enciphered key.

A variable-length AES key-token must have a key that can be used for decryption (key-usage field 1 high-order byte is B'x1xx xxxx'). Also, for processing rule **CBC** or **PKCS-PAD**, key usage must allow the key to be used for Cipher Block Chaining (KUF2 high-order byte is X'00'). For processing rule **ECB**, key usage must allow the key to be used for Electronic Code Book (KUF2 high-order byte is X'01').

- A block size of 16 for the cryptographic algorithm.
- For cipher block chaining, either one of these:
  1. For an ICV selection of **INITIAL**, a 16-byte initialization vector of your choosing and a 32-byte chain data buffer.
  2. For an ICV selection of **CONTINUE**, no initialization vector and the 32-byte chain data buffer from the output of the previous chained call. The electronic code book algorithm does not use an initialization vector or a chain data buffer.
- The cipher text to be deciphered.
- A clear text buffer large enough to receive the deciphered output.

This verb does the following when it deciphers the data:

1. Verifies the AES key-token for keyword **KEYIDENT**.
2. Verifies that the ciphertext length is a multiple of the block size.
3. Deciphers the input AES key if the key is encrypted (MKVP was present in token).
4. Deciphers the ciphertext with the AES clear key according to the encryption mode specified.
5. Removes from 1 - 16 pad characters from the right of the clear data for keyword **PKCS-PAD**.
6. Returns the cleartext data and its length.
7. Returns the chain data and its length if keyword **ECB** is not specified.

CPACF exploitation for CSNBSAD requires fixed-length AES key-tokens with a key type of **DATA** or variable-length AES key-tokens of type **CIPHER** with DECRYPT key-usage enabled. A fixed-length **AES DATA** key has a control vector (CV) of all X'00' bytes for all active bytes of the CV. Note that as of CCA Release 3.30 (the first release of AES function on the CEX2C 4764 adapter) to Release 4.1.0, AES keys were only available as DATA keys in fixed-length tokens. Beginning with Release 4.2.0 AES keys are available in variable-length symmetric key-tokens. For details about CPACF, see “CPACF support” on page 12.

Format

The format of CSNBSAD.

```
CSNBSAD(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_identifier_length,
             key_identifier,
             key_parms_length,
             key_parms,
             block_size,
             initialization_vector_length,
             initialization_vector,
             chain_data_length,
             chain_data,
             ciphertext_length,
             cipher_text,
             cleartext_length,
             cleartext,
             optional_data_length,
             optional_data)
```

Parameters

The parameters for CSNBSAD.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1, 2, 3, or 4.

rule_array

Direction: Input
Type: String array

An array of 8-byte keywords providing the processing control information. The keywords must be left-aligned and padded on the right with space characters. The *rule_array* keywords are described in Table 98.

Table 98. Keywords for Symmetric Algorithm Decipher control information

Keyword	Description
Decryption algorithm (One required)	
AES	Specifies use of the Advanced Encryption Standard (AES) as the deciphering algorithm. The block size for AES is 16 bytes, and the key length is 16, 24, or 32 bytes. AES is the only algorithm currently supported by this verb.
Processing rule (One, optional)	
CBC	Performs ANSI X3.102 cipher block chaining. The data must be a multiple of eight bytes. An OCV is produced and placed in the *chaining_vector* parameter. If the ICV selection keyword **CONTINUE** is specified, the CBC OCV from the previous call is used as the ICV for this call.

Table 98. Keywords for Symmetric Algorithm Decipher control information (continued)

Keyword	Description
ECB	Specifies deciphering in Electronic Code Book mode. The ciphertext length must be a multiple of the block size.
PKCS-PAD	Specifies that the cleartext was padded on the right with 1 - 16 bytes of pad characters, making the padded text a multiple of the block size, before the data was enciphered. Each pad character is valued to the number of pad characters added. The output cleartext is stripped of any pad characters and the cleartext length is 1 - 16 bytes less than the ciphertext length.
Key rule (One, optional)	
KEY-CLR	Specifies that the *key_identifier* parameter points to a cleartext AES key. Only the key value is allowed; the key is not contained in a key token. This is the default value.
KEYIDENT	Specifies that the *key_identifier* parameter points to an internal AES key-token or the label of an internal key-token in AES key-storage.
ICV selection (One, optional)	
CONTINUE	This specifies taking the initialization vector from the output chaining vector (OCV) contained in the work area to which the *chaining_vector* parameter points. **CONTINUE** is valid only for the **CBC** processing rule.
INITIAL	This specifies taking the initialization vector from the *initialization_vector* parameter. INITIAL is the default value.

"Ciphering methods" on page 932 describes the cipher processing rules in detail.

key_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *key_identifier* variable. This value must be 16, 24, 32, or ≥ 64.

key_identifier

Direction: Input
Type: String

A pointer to a string variable containing either a cleartext AES key or the internal key-token or a label for an internal key-token record in AES key-storage. This is the key used to decipher the data pointed to by the *ciphertext* parameter.

For *rule_array* keyword **KEY-CLR**, a 16-byte, 24-byte, or 32-byte clear AES key is required. For *rule_array* keyword **KEYIDENT**, a 64-byte internal key-token or key label for an internal key-token record in AES key-storage is required.

key_parms_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *key_parms* parameter. This value must be 0.

key_parms

Direction: Input

Type: String

A pointer to a string variable for key-related parameters. It is currently unused.

block_size

Direction: Input
Type: Integer

A pointer to an integer variable containing the block size used by the cryptographic algorithm. This value must be 16.

initialization_vector_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *initialization_vector* variable. For cipher block chaining (**CBC** or **PKCS-PAD**) with an **INITIAL** ICV selection, this value must be 16. For processing rule **ECB** or ICV selection **CONTINUE**, this value should be 0.

initialization_vector

Direction: Input
Type: String

A pointer to a string variable containing the initialization vector for the **INITIAL** call to **CBC** mode decryption. It is not used if the process rule is **ECB** or the ICV selection is **CONTINUE**. The same initialization vector must be used when deciphering the data.

chain_data_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *chain_data* variable. On input, this variable contains the length of the buffer provided and should have a value of 32 or greater for CBC mode encryption, or 0 for ECB mode encryption.

On output, the variable is updated with the length of the data returned in the *chain_data* variable. The *chain_data_length* parameter must not be changed by the calling application until chained operations are complete.

chain_data

Direction: Input/Output
Type: String

A pointer to a string variable used as a work area for **CBC** decipher requests. This work area is not used for **ECB** mode decryption. When the verb performs a CBC decipher operation and the ICV selection is **INITIA**L, the *chain_data* variable is an output-only buffer that receives data used as input for deciphering the next part of the input data, if any. When the ICV selection is **CONTINUE**, the *chain_data* variable is both an input and output buffer. The application must not change any intermediate data in this string.

ciphertext_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *ciphertext* variable. The *ciphertext_length* value must be a multiple of the block size. This value must not be 0. If **PKCS-PAD** is specified, the output *cleartext_length* variable will be 1 - 16 bytes less than the *ciphertext_length* value.

ciphertext

Direction: Input
Type: String

A pointer to a string variable containing the data to be deciphered, including any pad bytes.

cleartext_length

Direction: Input/Output
Type: Integer

On input, this parameter is a pointer to an integer variable containing the number of bytes of data in the cleartext variable. On output, this variable is updated to contain the actual length of text output in the cleartext variable. If **PKCS-PAD** is specified, the *cleartext* value is updated with 1 - 16 bytes of data less than the *ciphertext_length* value.

cleartext

Direction: Input/Output
Type: String

A pointer to a string variable used to contain the data to be deciphered, excluding any pad bytes.

optional_data_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *optional_data* variable. This value should be 0.

optional_data

Direction: Input
Type: String

A pointer to a string variable containing optional data for the decryption. It is currently not used.

Restrictions

The restrictions for CSNBSAD.

None.

Required commands

The required commands for CSNBSAD.

This verb requires the **Symmetric Algorithm Decipher - secure AES keys** command (offset X'012B') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBSAD.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBSADJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBSADJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger key_length,
           byte[]            key_identifier,
           hikmNativeInteger key_parms_length,
           byte[]            key_parms,
           hikmNativeInteger block_size,
           hikmNativeInteger iv_length,
           byte[]            iv,
           hikmNativeInteger chain_data_length,
           byte[]            chain_data,
           hikmNativeInteger cipher_text_length,
           byte[]            cipher_text,
           hikmNativeInteger clear_text_length,
           byte[]            clear_text,
           hikmNativeInteger optional_data_length,
           byte[]            optional_data);
```

Symmetric Algorithm Encipher (CSNBSAE)

Use the Symmetric Algorithm Encipher verb to encipher data using the AES algorithm.

CCA supports the following processing rules to encipher data. You choose the type of processing rule that the verb should use for block chaining.

Cipher Block Chaining (CBC)
The plaintext must be an exact multiple of eight bytes, and the ciphertext will have the same length.

Electronic Code Book (ECB)
The plaintext length must be a multiple of the block size.

PKCS-PAD
The plaintext was padded on the right with 1 - 16 bytes of pad characters, making the padded text a multiple of the block size.

The AES key used to encipher the data can either be 16, 24, or 32 bytes (128, 192, or 256 bits) in length. The key can be supplied to the verb in any of three forms:

1. A cleartext key consisting of only the key bytes, not contained in a key token.
2. A cleartext key contained in an internal fixed or variable length AES key-token.
3. An encrypted key contained in an internal fixed or variable length AES key-token, where the key is wrapped (encrypted) with the AES master key.

To use this verb, specify:

- The *rule_array*:
  1. The algorithm identifier keyword **AES**, which is the only symmetric algorithm currently supported.
  2. An optional processing rule using keyword **CBC** (the default), **ECB**, or **PKCS-PAD**, which selects the encryption mode.
  3. An optional key rule using the keyword **KEY-CLR** (the default) or **KEYIDENT**, which selects whether the *key_identifier* parameter points to a 16-byte, 24-byte, or 32-byte clear key, or a key contained in a 64-byte AES key-token in application storage or a key label of such a key in AES key-storage
  4. An optional ICV (initial chaining value) selection using the keyword **INITIAL** (the default) or **CONTINUE**, which indicates whether it is the first or a subsequent request, and which parameter points to the initialization vector.
- For a key rule of **KEY-CLR**, a key identifier containing a 16-byte, 24-byte, or 32-byte clear key. For a key rule of **KEYIDENT**, a fixed-length or variable-length internal AES key-token or the key label of such a key in AES key-storage. The key token can contain either a clear or enciphered key.

  A variable-length AES key-token must have a key that can be used for decryption (key-usage field 1 high-order byte is B'x1xx xxxx'). Also, for processing rule **CBC** or **PKCS-PAD**, key usage must allow the key to be used for Cipher Block Chaining (KUF2 high-order byte is X'00'). For processing rule **ECB**, key usage must allow the key to be used for Electronic Code Book (KUF2 high-order byte is X'01').
- A block size of 16 for the cryptographic algorithm.
- For cipher block chaining, either one of these:
  1. For an ICV selection of **INITIAL**, a 16-byte initialization vector of your choosing and a 32-byte chain data buffer.
  2. For an ICV selection of **CONTINUE**, no initialization vector and the 32-byte chain data buffer from the output of the previous chained call. The electronic code book algorithm does not use an initialization vector or a chain data buffer.
- The cleartext to be enciphered.
- A ciphertext buffer large enough to receive the enciphered output.

This verb does the following when it enciphers the data:

1. Verifies the AES key-token for keyword **KEYIDENT**.
2. Deciphers the input AES key if the key is encrypted (MKVP was present in token).
3. Pads the cleartext data with 1 - 16 bytes on the right for keyword **PKCS-PAD**, otherwise verifies that the cleartext length is a multiple of the block size.
4. Enciphers the cleartext, including any pad characters, with the AES clear key according to the encryption mode specified.
5. Returns the ciphertext data and its length.
6. Returns the chain data and its length if keyword **ECB** is not specified.

CPACF exploitation for CSNBSAE requires fixed-length AES key-tokens with a key type of **DATA** or variable-length AES key-tokens of type **CIPHER** with ENCRYPT key-usage enabled. A fixed-length **AES DATA** key has a control vector (CV) of all X'00' bytes for all active bytes of the CV. Note that as of CCA Release 3.30 (the first

release of AES function on the CEX2C 4764 adapter) to Release 4.1.0, AES keys were only available as **DATA** keys in fixed-length tokens. Beginning with Release 4.2.0 AES keys are available in variable-length symmetric key-tokens. For details about CPACF, see "CPACF support" on page 12.

Format

The format of CSNBSAE.

```
CSNBSAE(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_identifier_length,
             key_identifier,
             key_parms_length,
             key_parms,
             block_size,
             initialization_vector_length,
             initialization_vector,
             chain_data_length,
             chain_data,
             cleartext_length,
             cleartext,
             ciphertext_length,
             cipher_text,
             optional_data_length,
             optional_data)
```

Parameters

The parameters for CSNBSAE.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1, 2, 3, or 4.

rule_array

Direction: Input
Type: String array

A pointer to a string variable containing an array of keywords. The keywords are eight bytes in length, and must be left-aligned and padded on the right with space characters. The *rule_array* keywords are described in Table 99.

Table 99. Keywords for Symmetric Algorithm Encipher control information

Keyword	Description
Encryption algorithm (Required)	

Table 99. Keywords for Symmetric Algorithm Encipher control information (continued)

Keyword	Description
AES	Specifies use of the Advanced Encryption Standard (AES) as the encryption algorithm. The block size for **AES** is 16 bytes, and the key length is 16, 24, or 32 bytes. **AES** is the only algorithm currently supported by this verb.
Processing rule (One, optional)	
CBC	Performs ANSI X3.102 cipher block chaining. The data must be a multiple of eight bytes. An OCV is produced and placed in the *chaining_vector* parameter. If the ICV selection keyword **CONTINUE** is specified, the CBC OCV from the previous call is used as the ICV for this call.
ECB	Specifies enciphering in Electronic Code Book mode. The cleartext length must be a multiple of the block size.
PKCS-PAD	Specifies padding of the cleartext on the right with 1 - 16 bytes of pad characters, making the padded text a multiple of the block size. Each pad character is valued to the number of pad characters added. The ciphertext length must be large enough to include the added pad characters. The ciphertext length must be large enough to include the added pad characters. The padded cleartext is enciphered in Cipher Block Chaining mode.
Key rule (One, optional)	
KEY-CLR	Specifies that the *key_identifier* parameter points to a cleartext AES key. Only the key value is allowed; the key is not contained in a key token. This is the default value.
KEYIDENT	Specifies that the *key_identifier* parameter points to an internal AES key-token or the label of an internal key-token in AES key-storage.
ICV selection (One, optional)	
CONTINUE	Specifies taking the initialization vector from the output chaining vector (OCV) contained in the work area to which the *chaining_vector* parameter points. **CONTINUE** is valid only for the CBC processing rule.
INITIAL	Specifies taking the initialization vector from the *initialization_vector* parameter. **INITIAL** is the default value.

"Ciphering methods" on page 932 describes the cipher processing rules in detail.

key_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *key_identifier* variable. This value must be 16, 24, 32, or ≥ 64.

key_identifier

Direction: Input
Type: String

A pointer to a string variable containing either a cleartext AES key or the internal key-token or a label for an internal key-token record in AES key-storage. This is the key used to encipher the data pointed to by the cleartext parameter. For *rule_array* keyword **KEY-CLR**, a 16-byte, 24-byte, or 32-byte clear AES key is required. For *rule_array* keyword **KEYIDENT**, a 64-byte fixed or variable-length internal AES key-token or key label for such a key in AES key-storage is required.

key_parms_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *key_parms* parameter. This value must be 0.

key_parms

Direction: Input
Type: String

A pointer to a string variable for key-related parameters. It is currently unused.

block_size

Direction: Input
Type: Integer

A pointer to an integer variable containing the block size used by the cryptographic algorithm. This value must be 16.

initialization_vector_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *initialization_vector* variable. For cipher block chaining (**CBC** or **PKCS-PAD**) with an **INITIAL** ICV selection, this value must be 16. For processing rule **ECB** or ICV selection **CONTINUE**, this value should be 0.

initialization_vector

Direction: Input
Type: String

A pointer to a string variable containing the initialization vector for the **INITIAL** call to **CBC** mode encryption. It is not used if the process rule is **ECB** or the ICV selection is **CONTINUE**. The same initialization vector must be used when deciphering the data.

chain_data_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *chain_data* variable. On input, this variable contains the length of the buffer provided and should have a value of 32 or greater for **CBC** mode encryption, or 0 for **ECB** mode encryption.

On output, the variable is updated with the length of the data returned in the *chain_data* variable. The *chain_data_length* parameter must not be changed by the calling application until chained operations are complete.

chain_data

Direction: Input/Output
Type: String

A pointer to a string variable used as a work area for **CBC** encipher requests. This work area is not used for **ECB** mode encryption. When the verb performs a **CBC** encipher operation and the ICV selection is **INITIAL**, the *chain_data* variable is an output-only buffer that receives data used as input for enciphering the next part of the input data, if any. When the ICV selection is **CONTINUE**, the *chain_data* variable is both an input and output buffer. The

application must not change any intermediate data in this string.

cleartext_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *cleartext* variable. This length must be a multiple of the *block_size* variable unless processing rule **PKCS-PAD** is specified. A value of zero is not permitted.

cleartext

Direction: Input
Type: String

A pointer to a string variable used to contain the data to be enciphered, excluding any pad bytes.

ciphertext_length

Direction: Input/Output
Type: Integer

On input, the *ciphertext_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *ciphertext* variable. On output, the *ciphertext_length* variable is updated to contain the actual length of text output in the *ciphertext* variable. If **PKCS-PAD** is specified, the *ciphertext_length* value must be greater than or equal to the next higher multiple of 16 as the *cleartext_length* value (from 1 - 16 bytes longer). Otherwise, the *ciphertext_length* value must be greater than or equal to the *cleartext_length* variable.

ciphertext

Direction: Input/Output
Type: String

A pointer to a string variable used as an output buffer where the verb returns the enciphered data. If **PKCS-PAD** is specified, on output the *ciphertext* buffer contains 1 - 16 bytes of data more than the *cleartext* input buffer contains.

optional_data_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *optional_data* variable. This value should be 0.

optional_data

Direction: Input
Type: String

A pointer to a string variable containing optional data for the encryption. It is currently not used.

Restrictions

The restrictions for CSNBSAE.

None.

Required commands

The required commands for CSNBSAE.

This verb requires the **Symmetric Algorithm Encipher - secure AES keys** command (offset X'012A') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBSAE.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBSAEJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBSAEJ(
          hikmNativeInteger return_code,
          hikmNativeInteger reason_code,
          hikmNativeInteger exit_data_length,
          byte[]            exit_data,
          hikmNativeInteger rule_array_count,
          byte[]            rule_array,
          hikmNativeInteger key_length,
          byte[]            key_identifier,
          hikmNativeInteger key_parms_length,
          byte[]            key_parms,
          hikmNativeInteger block_size,
          hikmNativeInteger iv_length,
          byte[]            iv,
          hikmNativeInteger chain_data_length,
          byte[]            chain_data,
          hikmNativeInteger clear_text_length,
          byte[]            clear_text,
          hikmNativeInteger cipher_text_length,
          byte[]            cipher_text,
          hikmNativeInteger optional_data_length,
          byte[]            optional_data);
```

Cipher Text Translate2 (CSNBCTT2)

This callable service deciphers encrypted data (ciphertext) under one ciphertext translation key and re-enciphers it under another ciphertext translation key without having the data appear in the clear outside the cryptographic coprocessor. Such a function is useful in a multiple node network, where sensitive data is passed through multiple nodes prior to it reaching its final destination.

Use the Cipher Text Translate2 verb to decipher text under an input key and then to encipher the text under an output key. Both AES and DES algorithms are supported. Translation between AES and DES is allowed with restrictions controlled by access control points.

The encryption modes supported are:

- DES – CBC, CUSP and IPS
- AES – CBC and ECB

The padding methods supported are:

- DES – X9.23
- AES – PKCSPAD

Scenario for using the CSNBCTT2 verb

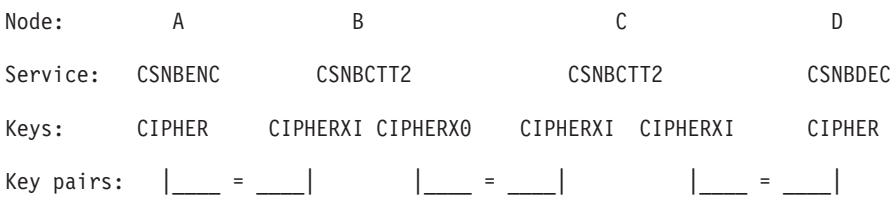
This scenario uses the Encipher (CSNBENC), Cipher Text Translate2 (CSNBCTT2), and Decipher (CSNBDEC) callable services with four network nodes: A, B, C, and D. You want to send data from your network node A to a destination node D. You cannot communicate directly with node D, because nodes B and C are situated between A and D. You do not want nodes B and C to decipher your data.

At node A, you use the CSNBENC service. Node D uses the CSNBDEC service. Node B and C will use the CSNBCTT2 service.

Consider the keys that are needed to support this process:

1. At your node, generate one key in two forms: OPEX CIPHER CIPHERXI.
2. Send the exportable CIPHERXI key to node B.
3. Node B and C need to share a key, so generate a different key in two forms: EXEX CIPHERX0 CIPHERXI.
4. Send the exportable CIPHERX0 key to node B.
5. Send the exportable CIPHERXI key to node C.
6. Node C and node D need to share a CIPHERX0 key and a CIPHER key. Node D can generate one key in two forms: OPEX CIPHERX0 CIPHERXI.
7. Node D sends the exportable CIPHERX0 key to node C.

The communication process is shown as:



Therefore, you need three keys, each in two different forms. You can generate two of the keys at node A, and node D can generate the third key. Note that the key used in the decipher callable service at node D is not the same key used in the encipher callable service at node A.

Format

The format of CSNBCTT2.

```
CSNBCTT2(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_identifier_in_length,
             key_identifier_in,
             init_vector_in_length,
             init_vector_in,
             cipher_text_in_length,
             cipher_text_in,
             chaining_vector_length,
             chaining_vector,
             key_identifier_out_length,
             key_identifier_out,
             init_vector_out_length,
             init_vector_out,
             cipher_text_out_length,
             cipher_text_out,
             reserved1_length,
             reserved1,
             reserved2_length,
             reserved2)
```

Parameters

The parameter definitions for CSNBCTT2.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. The value must be 4 or 5.

rule_array

Direction: Input
Type: String array

The keyword that provides control information to the verb. The processing method is the algorithm used to create the generated key. The keyword is left-aligned and padded on the right with blanks. The *rule_array* keywords are described in Table 100.

Table 100. Keywords for Cipher Text Translate2 control information

Keyword	Description
Inbound Processing Rule (One required)	
I-CBC	Specifies encryption using CBC mode for the inbound ciphertext. The text length must be a multiple of the block size. The DES block size is 8 bytes. The AES block size is 16 bytes.

Table 100. Keywords for Cipher Text Translate2 control information (continued)

Keyword	Description
I-CUSP	Specifies that CBC with CUSP processing for the inbound ciphertext. The ciphertext may be any length. The ciphertext is the same length as the plain text. This keyword is only valid with DES.
I-ECB	Specifies encryption using ECB mode for the inbound ciphertext. The text must be a multiple of the block size. This keyword is only valid for AES encryption.
I-IPS	Specifies that CBC with IPS processing has been used for the inbound ciphertext. The cipher text may be any length. The ciphertext is the same length as the plain text. This keyword is only valid with DES.
IPKCSPAD	Specifies that CBC with PKCS padding was used for the inbound ciphertext. The text was padded on the right with 1 - 16 bytes of pad characters, making the padded text a multiple of the AES block size, before the data was enciphered. Each pad character is valued to the number of pad characters added. This keyword is only valid for AES encryption.
I-X923	Specifies that CBC with X9.24 padding was used for the inbound ciphertext. This is compatible with the requirements in ANSI Standard X9.23. This keyword is only valid for DES encryption.
Outbound processing rule (One required)	
O-CBC	Specifies that encryption in CBC mode is used for the outbound ciphertext. The text length must be a multiple of the block size. The DES block size is 8 bytes. The AES block size is 16 bytes.
O-CUSP	Specifies that CBC with CUSP processing is used for the outbound text. The outbound ciphertext has the same length as the plain text. This keyword is only valid with DES.
O-ECB	Specifies that encryption using ECB mode is used for the outbound ciphertext. The text must be a multiple of the block size. This keyword is only valid for AES encryption.
O-IPS	Specifies that CBC with IPS processing is used for the outbound text. The outbound ciphertext has the same length as the plain text. This keyword is only valid with DES.
OPKCSPAD	Specifies that CBC with PKCS padding is used for the outbound text. The outbound text is padded on the right with 1 - 16 bytes of pad characters, making the padded text a multiple of the AES block size, before the data was enciphered. Each pad character is valued to the number of pad characters added. This keyword is only valid for AES encryption.
O-X923	Specifies that CBC with X9.24 padding is used for the outbound text. This is compatible with the requirements in ANSI Standard X9.23. This keyword option is only valid for DES encryption.
Segmenting control (One optional)	
CONTINUE	Specifies the initialization vectors are taken from the chaining vector. The chaining vector is updated and must not be modified between calls. This keyword is ignored for I-ECB and O-ECB processing rules. The CONTINUE keyword is not valid with the I-X923 or O-X923 keywords.
INITIAL	Specifies that the initialization vectors are taken from the **init_vector_in** and **init_vector_out** parameters. This is the default. This keyword is ignored for I-ECB and O-ECB processing rules.
Inbound key identifier (One required)	
IKEY-DES	Specifies that the inbound key identifier is a DES key.
IKEY-AES	Specifies that the inbound key identifier is an AES key.
Outbound key identifier (One required)	
OKEY-DES	Specifies that the outbound key identifier is a DES key.
OKEY-AES	Specifies that the outbound key identifier is an AES key.

key_identifier_in_length

Direction: Input
Type: Integer

Length of the **key_identifier_in** in bytes. The value is 64 when a label is

supplied. When the key identifier is a key token, the value is the length of the token. The maximum value is 725.

key_identifier_in

Direction: Input/Output
Type: String

An internal key token or the label of an AES or DES key storage record containing the cipher translation key for the inbound ciphertext.

Acceptable DES key types are DATA, CIPHER, CIPHERXI, CIPHERXL, and DECIPHER. The keys must have bit 19 for DECIPHER set on in the control vector. The key may be a single-, double-, or triple-length key. If the **Cipher Text translate2 - Allow only ciphertext translate types** access control point is enabled, only CIPHERXI and CIPHERXL are allowed.

Acceptable AES key types include the 64-byte AES DATA key and the variable length token CIPHER key with the DECRYPT bit on in the key usage field. The C-XLATE bit can optionally be on. If the **Cipher Text translate2 - Allow only ciphertext translate types** access control point is enabled, the C-XLATE bit must be turned on in the key usage field.

init_vector_in_length

Direction: Input
Type: Integer

Length of the **init_vector_in** field in bytes. For AES keys, the length is 16. For DES keys, the length is 8. When the initialization vector is not required (segmenting rule CONTINUE, processing rule I-ECB), the value must be 0.

init_vector_in

Direction: Input
Type: String

The initialization vector that is used to decipher the input data. This parameter is the initialization vector used at the previous cryptographic node. This parameter is required for segmenting rule INITIAL.

cipher_text_in_length

Direction: Input
Type: Integer

The length of the ciphertext to be processed. See the table of ciphertext length restrictions in “Usage notes” on page 363.

cipher_text_in

Direction: Input
Type: String

The text that is to be translated. The text is enciphered under the cipher key specified in the **key_identifier_in** parameter.

chaining_vector_length

Direction: Input
Type: Integer

The length of the **chaining_vector** parameter in bytes. The **chaining_vector** field must be 128 bytes long.

chaining_vector

Direction: Input/Output
Type: String

The **chaining_vector** parameter is a work area used by the service to carry segmented data between procedure calls. This area must not be modified between calls to the service.

key_identifier_out_length

Direction: Input
Type: Integer

The length of the **key_identifier_out** parameter in bytes. This value is 64 when a label is supplied. When the key identifier is a key token, the value is the length of the token. The maximum value is 725.

key_identifier_out

Direction: Input/Output
Type: String

An internal key token or the label of an AES or DES key storage record containing the cipher translation key for the outbound ciphertext.

Acceptable DES key types are DATA, CIPHER, CIPHERXL, CIPHERXO, and ENCIPHER. The key may be a double- or triple-length key. If the **Cipher Text translate2 – Allow only ciphertext translate types** access control point is enabled, only CIPHERXO and CIPHERXL are allowed. Acceptable DES key types are DATA, CIPHER, CIPHERXL, CIPHERXO, and ENCIPHER. The keys must have bit 18 for ENCIPHER set on in the control vector. The key may be a double- or triple-length key. If the **Cipher Text translate2 - Allow only ciphertext translate types** access control point is enabled, only CIPHERXO and CIPHERXL are allowed.

Acceptable AES key types include the 64-byte AES DATA key and the variable length token CIPHER key with the ENCRYPT bit on in the key usage field.

The C-XLATE bit can optionally be on. If the **Cipher Text translate2 – Allow only ciphertext translate types** access control point is enabled, the C-XLATE bit must be turned on in the key usage field.

init_vector_out_length

Direction: Input
Type: Integer

The length of the **init_vector_out** parameter in bytes. For AES keys, the length is 16. For DES keys, the length is 8. When the initialization vector is not required (segmenting rule CONTINUE, processing rule O-ECB), the value must be 0.

init_vector_out

Direction: Input
Type: String

The initialization vector that is used to encipher the input data. This is the new initialization vector used when the callable service enciphers the plain text. This parameter is required for segmenting rule INITIAL.

cipher_text_out_length

Direction: Input/Output
Type: Integer

The length of the **cipher_text_out** parameter in bytes. This parameter is updated with the actual length of the data in the **cipher_text_out** parameter. Note that padding may require this value to be larger than the **cipher_text_in_length** parameter (see Table 101 on page 363).

cipher_text_out

Direction: Output
Type: String

The field where the callable service returns the translated text.

reserved1_length

Direction: Input
Type: Integer

The length of the **reserved1** parameter in bytes. The value must be zero.

reserved1

Direction: Input
Type: String

This parameter is ignored.

reserved2_length

Direction: Input
Type: Integer

The length of the **reserved2** parameter in bytes. The value must be zero.

reserved2

Direction: Input
Type: String

This parameter is ignored.

Restrictions

The restrictions for CSNBCTT2.

None.

Required commands

The CSNBCTT2 required commands.

This verb requires the following commands to be enabled in the active role:

Command	Offset	Description
Cipher Text Translate2	X'01C0'	Enable the Ciphertext Translate2 service
Cipher Text Translate2 – Allow translate from AES to TDES	X'01C1'	Allow translation from an AES key to 2 or 3 key triple DES key.

Command	Offset	Description
Cipher Text Translate2 – Allow translate to weaker AES	X'01C2'	Allow translation from a stronger to weaker AES key. (For example, IN key AES256 and OUT key AES128.)
Cipher Text Translate2 – Allow translate to weaker DES	X'01C3'	Allow translation from a triple-length DES key to a weaker double-length DES key.
Cipher Text Translate2 – Allow only ciphertext translate types	X'01C4'	When enabled, the verb only accepts these key-translation-only key types, which cannot be used in normal cipher or MAC data operations: • AES key tokens with key type CIPHER and key usage set to allow data translate (C-XLATE) only • DES key tokens with a key type CIPHERXI, CIPHERXO, or CIPHERXL In other words, with offset X'01C4' enabled in the active role, AES key type AESDATA and DES key types CIPHER, DATA, DATAC, and ENCIPHER, are not allowed. This restricts the verb from using keys that can be used in data operations other than translation.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

Usage notes for CSNBCTT2.

Restriction: The Cipher Text Translate2 callable service is only available on the IBM zEnterprise EC12 and later servers.

The initialization vectors must have already been established between the communicating applications or must be passed with the data.

Table 101 outlines the restrictions for the **cipher_text_in_length** and **cipher_text_out_length** parameters. The DES blocks referred to in this table are 8 bytes. The AES blocks referred to in this table are 16 bytes.

Table 101. Restrictions for **cipher_text_in_length** *and* **cipher_text_out_length**.

Input cipher method	Output cipher method	Input cipher text length restriction(s)	Output cipher text length restriction(s)
DES CBC	DES CBC X9.23	Input cipher text must be a multiple of a DES block.	Output cipher text length must be greater than or equal to the sum of the length of the input cipher text and a DES block.
DES CBC	AES CBC PKCSPAD	Input cipher text must be a multiple of a DES block.	If the input cipher text is not a multiple of an AES block, then the output cipher text length must be greater than or equal to the sum of the input cipher text length and a DES block. If the input cipher text is a multiple of an AES block, then the output cipher text length must be greater than or equal to the sum of the input cipher text length and an AES block.

Table 101. Restrictions for **cipher_text_in_length** *and* **cipher_text_out_length** *(continued).*

Input cipher method	Output cipher method	Input cipher text length restriction(s)	Output cipher text length restriction(s)
DES CBC	DES CUSP or IPS	Input cipher text must be a multiple of a DES block.	Output cipher text length must be greater than or equal to the input cipher text length.
DES CBC	DES CBC	Input cipher text must be a multiple of a DES block.	Output cipher text length must be greater than or equal to the input cipher text length.
DES CBC	AES CBC	Input cipher text must be a multiple of an AES block.	Output cipher text length must be greater than or equal to the input cipher text length.
DES CBC	AES CBC	Input cipher text must be a multiple of an AES block.	Output cipher text length must be greater than or equal to the input cipher text length.
DES CBC CUSP or IPS	DES CBC CUSP or IPS	No restrictions	Output cipher text length must be greater than or equal to the input cipher text length.
DES CBC CUSP or IPS	DES CBC	Input cipher text must be a multiple of a DES block.	Output cipher text length must be greater than or equal to the input cipher text length.
DES CBC CUSP or IPS	AES CBC or ECB	Input cipher text must be a multiple of an AES block.	Output cipher text length must be greater than or equal to the input cipher text length.
DES CBC CUSP or IPS	DES CBC X9.23	No restrictions	Output cipher text length must be greater than or equal to the sum of the input cipher text length and a DES block.
DES CBC CUSP or IPS	AES CBC PKCSPAD	No restrictions	Output cipher text length must be greater than or equal to the sum of the input cipher text length and an AES block.
DES CBC X9.23	DES CBC X9.23	Input cipher text must be a multiple of a DES block.	Output cipher text length must be greater than or equal to the input cipher text length.
DES CBC X9.23	AES CBC PKCSPAD	Input cipher text must be a multiple of a DES block.	Output cipher text length must be greater than or equal to the sum of the input cipher text length and a DES bock.
DES CBC X9.23	DES CBC CUSP or IPS	Input cipher text must be a multiple of a DES block.	Output cipher text length must be greater than or equal to the input cipher text length.
DES CBC X9.23	DES CBC	Input cipher text must be a multiple of a DES block.	Output cipher text length must be greater than or equal to the input cipher text length. **Note:** This operation is not possible if the padding is determined by the adapter to be from 1-7 bytes.
DES CBC X9.23	AES CBC	Input cipher text must be a multiple of a DES block but must not be a multiple of an AES block.	Output cipher text length must be greater than or equal to the input cipher text length. **Note:** This operation is not possible if the padding is determined by the adapter to be from 1-7 bytes.
DES CBC X9.23	AES ECB	Input cipher text must be a multiple of a DES block but must not be a multiple of an AES block.	Output cipher text length must be greater than or equal to the input cipher text length. **Note:** This operation is not possible if the padding is determined by the adapter to be from 1-7 bytes.
AES CBC or ECB	DES CBC X9.23	Input cipher text must be a multiple of an AES block.	Output cipher text length must be greater than or equal to the sum of the input cipher text length and a DES bock.
AES CBC or ECB	AES CBC PKCSPAD	Input cipher text must be a multiple of an AES block.	Output cipher text length must be greater than or equal to the sum of the input cipher text length and an AES bock.
AES CBC or ECB	DES CBC CUSP or IPS	Input cipher text must be a multiple of an AES block.	Output cipher text length must be greater than or equal to the input cipher text length.

Table 101. Restrictions for **cipher_text_in_length** *and* **cipher_text_out_length** *(continued).*

Input cipher method	Output cipher method	Input cipher text length restriction(s)	Output cipher text length restriction(s)
AES CBC or ECB	DES CBC	Input cipher text must be a multiple of an AES block.	Output cipher text length must be greater than or equal to the input cipher text length.
AES CBC or ECB	AES CBC	Input cipher text must be a multiple of an AES block.	Output cipher text length must be greater than or equal to the input cipher text length.
AES CBC or ECB	AES ECB	Input cipher text must be a multiple of an AES block.	Output cipher text length must be greater than or equal to the input cipher text length.
AES CBC PKCSPAD	DES CBC X9.23	Input cipher text must be a multiple of an AES block.	Output cipher text length must be greater than or equal to the input cipher text length.
AES CBC PKCSPAD	AES CBC PKCSPAD	Input cipher text must be a multiple of an AES block.	Output cipher text length must be greater than or equal to the input cipher text length.
AES CBC PKCSPAD	DES CBC CUSP or IPS	Input cipher text must be a multiple of an AES block.	Output cipher text length must be greater than or equal to the input cipher text length minus 1.
AES CBC PKCSPAD	DES CBC	Input cipher text must be a multiple of an AES block. Output cipher text length must be greater than or equal	Output cipher text length must be greater than or equal to the input cipher text length minus the length of a DES block. **Note:** This operation is not possible if the padding is determined by the adapter to be from 1-7 bytes or 9-15 bytes.
AES CBC PKCSPAD	AES CBC	Input cipher text must be a multiple of an AES block.	Output cipher text length must be greater than or equal to the input cipher text length minus the length of a AES block. **Note:** This operation is not possible if the padding is determined by the adapter to be from 1-15 bytes.
AES CBC PKCSPAD	AES ECB	Input cipher text must be a multiple of an AES block.	Output cipher text length must be greater than or equal to the input cipher text length minus the length of an AES block. **Note:** This operation is not possible if the padding is determined by the adapter to be from 1-15 bytes.

There are requirements for the keys for the **key_identifier_in** and **key_identifier_out** parameters. The **key_identifier_in** key must be able to decipher text. The **key_identifier_out** key must be able to encipher text.

Table 102 on page 366 table shows the valid key types which are allowed for the **key_identifier_in** and **key_identifier_out** parameters. In the table, a variable length key token cipher key is denoted by vCIPHER. vCIPHER is the default which has the ENCRYPT and DECRYPT bits on in the usage field. vCIPHERe has only the ENCRYPT bit on in the usage field. vCIPHERd has only the DECRYPT bit on in the usage field. Adding x to either of the preceding names means the TRANSLAT bit is on in the usage field for that key. For example, vCIPHERex means a variable length token with the ENCRYPT and TRANSLAT bits turned on.

AESDATA is the 64-byte AES DATA key type.

Table 102. Cipher Text Translate2 key usage.

key_identifier_in (DEC bit except DATA and AESDATA)	**key_identifier_out (ENC bit except DATA and AESDATA)**
DATA CIPHER DECIPHER CIPHERXI CIPHERXL	DATA CIPHER ENCIPHER CIPHERXO CIPHERXL AESDATA vCIPHER vCIPHERe vCIPHERex vCIPHERedx
AESDATA vCIPHER vCIPHERd vCIPHERdx vCIPHERdex	DATA (must be at least double-length key with ACP) CIPHER (requires ACP to be enabled) ENCIPHER (requires ACP to be enabled) CIPHERXO (requires ACP to be enabled) CIPHERXL (requires ACP to be enabled) AESDATA vCIPHER vCIPHERe vCIPHERex vCIPHERedx

Note:

1. Translation from stronger encryption to single-key DES is not allowed.
2. Translation from a triple-length DES key to a double-length DES key requires the **Cipher Text Translate2 - Allow translate to weaker DES** access control point (offset X'01C3') to be enabled.
3. When the **Cipher Text Translate2 - Allow only ciphertext translate types** access control point (offset X'01C4') is enabled, only CIPHERXI, CIPHERXL, and CIPHERXO DES key types are allowed and AES key tokens with key type CIPHER must be set to allow data translate (C-XLATE).

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBCTT2.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBCTT2J(
  hikmNativeInteger   return_code,
  hikmNativeInteger   reason_code,
  hikmNativeInteger   exit_data_length,
  byte[]              exit_data,
  hikmNativeInteger   rule_array_count,
  byte[]              rule_array,
  hikmNativeInteger   key_identifier_in_len,
  byte[]              key_identifier_in,
  hikmNativeInteger   init_vector_in_length,
  byte[]              init_vector_in,
  hikmNativeInteger   cipher_text_in_length,
  byte[]              cipher_text_in,
```

```
| hikmNativeInteger   chaining_vector_length,
| byte[]              chaining_vector,
| hikmNativeInteger   key_identifier_out_length,
| byte[]              key_identifier_out,
| hikmNativeInteger   init_vector_out_length,
| byte[]              init_vector_out,
| hikmNativeInteger   cipher_text_out_length,
| byte[]              cipher_text_out,
| hikmNativeInteger   reserved1_length,
| byte[]              reserved1,
| hikmNativeInteger   reserved2_length,
| byte[]              reserved2);
```

Chapter 8. Verifying data integrity and authenticating messages

CCA provides methods to verify the integrity of transmitted messages and stored data

The methods provided are:

- Message authentication code (MAC)
- Hash functions, including Modification Detection Code (MDC) processing and one-way hash generation

Note: You can also use digital signatures (see Chapter 12, "Using digital signatures," on page 625) to authenticate messages.

The choice of verb depends on the security requirements of the environment in which you are operating. If you need to ensure the authenticity of the sender as well as the integrity of the data and both the sender and receiver can share a secret key, consider Message Authentication Code processing. If you need to ensure the integrity of transmitted data in an environment where it is not possible for the sender and the receiver to share a secret cryptographic key, consider hashing functions.

The verbs described in this topic include:

How MACs are used

When a message is sent, an application program can generate an authentication code for it using the MAC Generate verb.

CCA supports the ANSI X9.9-1 basic procedure and both the ANSI X9.19 basic procedure and optional double key MAC procedure. The MAC Generate verb computes the text of the Message Authentication Code using the algorithm and a key. The ANSI X9.9-1 or ANSI X9.19 basic procedures accept either a single-length MAC generation (MAC) key or a data-encrypting (DATA) key, and the message text. The ANSI X9.19 optional double key MAC procedure accepts a double-length MAC key and the message text. The originator of the message sends the MAC with the message text.

When the receiver gets the message, an application program calls the MAC Verify verb. The MAC Generate verb generates a MAC using the same algorithm as the sender and either the single-length or double-length MAC verification key, the single-length or double-length MAC generation key, or DATA key, and the message

text. The MAC Verify verb compares the MAC it generates with the one sent with the message and issues a return code that indicates whether the MACs match. If the return code indicates that the MACs match, the receiver can accept the message as genuine and unaltered. If the return code indicates that the MACs do not match, the receiver can assume the message is either fraudulent or has been altered. The newly computed MAC is not revealed outside the cryptographic coprocessor.

In a similar manner, MACs can be used to ensure the integrity of data stored on the system or on removable media, such as tape.

Secure use of the MAC Generate and MAC Verify verbs requires the use of MAC and MACVER keys in these verbs, respectively. To accomplish this, the originator of the message generates a MAC/MACVER key pair, uses the MAC key in the MAC Generate verb, and exports the MACVER key to the receiver. The originator of the message enforces key separation on the link by encrypting the MACVER key under a transport key that is not an NOCV key before exporting the key to the receiver. With this type of key separation enforced, the receiver can receive only a MACVER key and can use only this key in the MAC Verify verb. This ensures that the receiver cannot alter the message and produce a valid MAC with the altered message. These security features are not present if DATA keys are used in the MAC Generate verb or if DATA or MAC keys are used in the MAC Verify verb.

By using MACs you get the following benefits:

- **For data transmitted over a network,** you can validate the authenticity of the message as well as ensure the data has not been altered during transmission. For example, an active eavesdropper can tap into a transmission line and interject fraudulent messages or alter sensitive data being transmitted. If the data is accompanied by a MAC, the recipient can use a verb to detect whether the data has been altered. Because both the sender and receiver share a secret key, the receiver can use a verb that calculates a MAC on the received message and compares it to the MAC transmitted with the message. If the comparison is equal, the message could be accepted as unaltered. Furthermore, because the shared key is secret, when a MAC is verified it can be assumed that the sender was, in fact, the other person who knew the secret key.
- **For data stored on tape or DASD,** you can ensure that the data read back onto the system was the same as the data written onto the tape or DASD. For example, someone might be able to bypass access controls. Such an access might escape the notice of auditors. However, if a MAC is stored with the data, and verified when the data is read, you can detect alterations to the data.

How hashing functions and MDCs are used

Hashing functions include the MDC and one-way hash.

You need to hash text before submitting it to the Digital Signature Generate and Digital Signature Verify verbs (see Chapter 12, “Using digital signatures,” on page 625). CCA supports the SHA-1, MD5, and RIPEMD-160 hashing functions.

When a message is sent, an application program can generate a hash or a Modification Detection Code (MDC) for it using the One-Way Hash verb. This verb computes the hash or MDC, a short, fixed-length value, using a one-way cryptographic function and the message text. The originator of the message

ensures the hash or MDC is transmitted with integrity to the intended receiver of the message. For example, the value could be published in a reliable source of public information.

When the receiver gets the message, an application program calls the One-Way Hash verb to generate a new hash or MDC using the same function and message text that were used by the sender. The application program can compare the new value with the one generated by the originator of the message. If the two values match, the receiver knows the message was not altered.

In a similar manner, hashes and MDCs can be used to ensure the integrity of data stored on the system or on removable media, such as tape.

By using hashes and MDCs, you get the following benefits:

- **For data transmitted over a network between locations that do not share a secret key,** you can ensure the data has not been altered during transmission. It is easy to compute a hash or MDC for specific data, yet hard to find data that will result in a given hash or MDC. In effect, the problem of ensuring the integrity of a large file is reduced to ensuring the integrity of a short, fixed-length value.
- **For data stored on tape or DASD,** you can ensure that the data read back onto the system was the same as the data written onto the tape or DASD. After a hash has been established for a file, the One-Way Hash verb can be run at any later time on the file. The resulting value can be compared with the stored value to detect deliberate or inadvertent modification.

For more information, see "Modification Detection Code calculation" on page 930.

HMAC Generate (CSNBHMG)

Use the HMAC Generate verb to generate a keyed hash Message Authentication Code (HMAC) for the message string provided as input.

An HMAC key that can be used for generate is required to calculate the HMAC.

Format

The format of CSNBHMG.

```
CSNBHMG(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_identifier_length,
             key_identifier,
             text_length,
             text,
             chaining_vector_length,
             chaining_vector,
             mac_length,
             mac)
```

Parameters

The parameters for CSNBHMG.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

The number of keywords you supplied in the *rule_array* parameter. This value must be 2 or 3.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. The following table lists the keywords. Each keyword is left-aligned in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage. The *rule_array* keywords are described in Table 103.

Table 103. Keywords for HMAC Generate control information

Keyword	Description
Token algorithm (One required)	
HMAC	Specifies the HMAC algorithm to be used to generate the MAC.
Hash method (One required).	
SHA-1	Specifies the FIPS-198 HMAC procedure using the SHA-1 hash method, a symmetric key and text to produce a 20-byte (160-bit) MAC.
SHA-224	Specifies the FIPS-198 HMAC procedure using the SHA-224 hash method, a symmetric key and text to produce a 28-byte (224-bit) MAC.
SHA-256	Specifies the FIPS-198 HMAC procedure using the SHA-256 hash method, a symmetric key and text to produce a 32-byte (256-bit) MAC.
SHA-384	Specifies the FIPS-198 HMAC procedure using the SHA-384 hash method, a symmetric key and text to produce a 48-byte (384-bit) MAC.
SHA-512	Specifies the FIPS-198 HMAC procedure using the SHA-512 hash method, a symmetric key and text to produce a 64-byte (512-bit) MAC.
Segmenting control (One optional)	
FIRST	First call, this is the first segment of data from the application program.
LAST	Last call; this is the last data segment.
MIDDLE	Middle call; this is an intermediate data segment.
ONLY	Only call; segmenting is not employed by the application program. This is the default value.

key_identifier_length

Direction: Input
Type: Integer

The length of the *key_identifier* parameter. The maximum value is 725.

key_identifier

Direction: Input
Type: String

The 64-byte label or internal token of an encrypted HMAC key.

text_length

Direction: Input
Type: Integer

The length of the text you supply in the *text* parameter. The maximum length of *text* is 214783647 bytes. For **FIRST** and **MIDDLE** calls, the *text_length* must be a multiple of 64 for **SHA-1**, **SHA-224** and **SHA-256** hash methods, and a multiple of 128 for **SHA-384** and **SHA-512** hash methods.

text

Direction: Input
Type: String

The application-supplied text for which the MAC is generated.

chaining_vector_length

Direction: Input/Output
Type: Integer

The length of the *chaining_vector* in bytes. This value must be 128.

chaining_vector

Direction: Input/Output
Type: String

An 128-byte string used as a system work area. Your application program must not change the data in this string. The chaining vector permits data to be chained from one invocation call to another.

On the first call, initialize this parameter as binary zeros.

mac_length

Direction: Input/Output
Type: Integer

The length of the *mac* parameter in bytes. This parameter is updated to the actual length of the *mac* parameter on output. The minimum value is 4, and the maximum value is 64.

mac

Direction: Output
Type: String

The field in which the verb returns the MAC value if the segmenting rule is **ONLY** or **LAST**.

Restrictions

The restrictions for CSNBHMG.

This verb was introduced with CCA 4.1.0.

Required commands

The required commands for CSNBHMG.

This verb requires the commands shown in the following table to be enabled in the active role:

Rule-array keyword	Offset	Command
SHA-1	X'00E4'	**HMAC Generate - SHA-1**
SHA-224	X'00E5'	**HMAC Generate - SHA-224**
SHA-256	X'00E6'	**HMAC Generate - SHA-256**
SHA-384	X'00E7'	**HMAC Generate - SHA-384**
SHA-512	X'00E8'	**HMAC Generate - SHA-512**

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBHMG.

None.

Related information

More information about CSNBHMG is provided in other topics.

The HMAC Verify verb is described in "HMAC Verify (CSNBHMV)."

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBHMGJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBHMGJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger key_identifier_length
           byte[]            key_identifier,
           hikmNativeInteger text_length,
           byte[]            text,
           hikmNativeInteger chaining_vector_length,
           byte[]            chaining_vector,
           hikmNativeInteger mac_length,
           byte[]            MAC);
```

HMAC Verify (CSNBHMV)

Use the HMAC Verify verb to verify a keyed hash Message Authentication Code (HMAC) for the message string provided as input.

A MAC key contained in an internal variable-length symmetric key-token is required to verify the HMAC. The key must have the same value as the key used to generate the HMAC.

Format

The format of CSNBHMV.

```
CSNBHMV(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              rule_array_count,
              rule_array,
              key_identifier_length,
              key_identifier,
              text_length,
              text,
              chaining_vector_length,
              chaining_vector,
              mac_length,
              mac)
```

Parameters

The parameters for CSNBHMV.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

The number of keywords you supplied in the *rule_array* parameter. This value must be 2 or 3.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. The following table lists the keywords. Each keyword is left-aligned in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage. The *rule_array* keywords are described in Table 104.

Table 104. Keywords for HMAC Verify control information

Keyword	Description
Token algorithm (One required)	
HMAC	Specifies that the HMAC algorithm is to be used to verify the MAC.
Hash method (One required)	
SHA-1	Specifies the FIPS-198 HMAC procedure using the SHA-1 hash method, a symmetric key and text to produce a 20-byte (160-bit) MAC.
SHA-224	Specifies the FIPS-198 HMAC procedure using the SHA-224 hash method, a symmetric key and text to produce a 28-byte (224-bit) MAC.
SHA-256	Specifies the FIPS-198 HMAC procedure using the SHA-256 hash method, a symmetric key and text to produce a 32-byte (256-bit) MAC.
SHA-384	Specifies the FIPS-198 HMAC procedure using the SHA-384 hash method, a symmetric key and text to produce a 48-byte (384-bit) MAC.

Table 104. Keywords for HMAC Verify control information (continued)

Keyword	Description
SHA-512	Specifies the FIPS-198 HMAC procedure using the SHA-512 hash method, a symmetric key and text to produce a 64-byte (512-bit) MAC.
Segmenting control (Optional)	
FIRST	First call, this is the first segment of data from the application program.
LAST	Last call; this is the last data segment.
MIDDLE	Middle call; this is an intermediate data segment.
ONLY	Only call; segmenting is not employed by the application program. This is the default value.

key_identifier_length

Direction: Input
Type: Integer

The length of the *key_identifier* parameter. The maximum value is 725.

key_identifier

Direction: Input/Output
Type: String

The 64-byte label or internal token of an encrypted HMAC or HMACVER key.

text_length

Direction: Input
Type: Integer

The length of the text you supply in the *text* parameter. The maximum length of *text* is 214783647 bytes. For **FIRST** and **MIDDLE** calls, the *text_length* must be a multiple of 64 for **SHA-1**, **SHA-224**, and **SHA-256** hash methods, and a multiple of 128 for **SHA-384** and **SHA-512** hash methods.

text

Direction: Input
Type: String

The application-supplied text for which the HMAC is to be verified.

chaining_vector_length

Direction: Input/Output
Type: Integer

The length of the *chaining_vector* in bytes. This value must be 128.

chaining_vector

Direction: Input/Output
Type: String

An 128-byte string used as a system work area. Your application program must not change the data in this string. The chaining vector permits data to be chained from one invocation call to another.

On the first call, initialize this parameter as binary zeros.

mac_length

Direction: Input
Type: Integer

The length of the *mac* parameter in bytes. The maximum value is 64.

mac

Direction: Input
Type: String

The field that contains the MAC value you want to verify.

Restrictions

The restrictions for CSNBHMV.

This verb was introduced with CCA 4.1.0.

Required commands

The required commands for CSNBHMV.

This verb requires the commands shown in the following table to be enabled in the active role:

Rule-array keyword	Offset	Command
SHA-1	X'00F7'	**HMAC Verify - SHA-1**
SHA-224	X'00F8'	**HMAC Verify - SHA-224**
SHA-256	X'00F9'	**HMAC Verify - SHA-256**
SHA-384	X'00FA'	**HMAC Verify - SHA-384**
SHA-512	X'00FB'	**HMAC Verify - SHA-512**

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBHMV.

None.

Related information

Additional information about CSNBHMV,

The HMAC Generate verb is described in "HMAC Generate (CSNBHMG)" on page 371.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBHMVJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBHMVJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger key_identifier_length
           byte[]            key_identifier,
           hikmNativeInteger text_length,
           byte[]            text,
           hikmNativeInteger chaining_vector_length,
           byte[]            chaining_vector,
           hikmNativeInteger mac_length,
           byte[]            MAC);
```

MAC Generate (CSNBMGN)

When a message is sent, an application program can generate an authentication code for it using the MAC Generate verb.

This verb generates a 4-byte, 6-byte, or 8-byte Message Authentication Code (MAC) for an application-supplied text string.

This verb computes the Message Authentication Code using one of the following methods:

- Using the ANSI X9.9-1 single key algorithm, a single-length MAC generation key or data-encrypting key, and the message text.
- Using the ANSI X9.19 optional double key algorithm, a double-length MAC generation key and the message text.
- Using the Europay, MasterCard and Visa (EMV) padding rules.

The MAC can be the leftmost 32 or 48 bits of the last block of the ciphertext or the entire last block (64 bits) of the ciphertext. The originator of the message sends the Message Authentication Code with the message text.

Host CPU acceleration: CPACF

Only keys with a key type of **DATA** can be used successfully with the CPACF exploitation layer through this verb.

Specifically, a **DATA** key has a CV (Control Vector) of all X'00' bytes for all active bytes of the CV (eight bytes for 8-byte DES keys, 16 bytes for 16-byte DES keys, and 16 bytes for 24-byte DES keys).

For details about CPACF, see “CPACF support” on page 12.

Format

The format of CSNBMGN.

```
CSNBMGN(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             key_identifier,
             text_length,
             text,
             rule_array_count,
             rule_array,
             chaining_vector,
             mac)
```

Parameters

The parameters for CSNBMGN.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

key_identifier

Direction: Input/Output
Type: String

The 64-byte key label or internal key token that identifies a single-length or double-length MAC generate key or a single-length **DATA** or **DATAM** key. The type of key depends on the MAC process rule in the *rule_array* parameter.

text_length

Direction: Input
Type: Integer

The length of the text you supply in the *text* parameter. If the *text_length* is not a multiple of eight bytes and if the **ONLY** or **LAST** keyword of the *rule_array* parameter is called, the text is padded in accordance with the processing rule specified.

text

Direction: Input
Type: String

The application-supplied text for which the MAC is generated.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0, 1, 2, or 3.

rule_array

Direction: Input
Type: String array

Zero to three keywords that provide control information to the verb. The

keywords are described in Table 105. The keywords must be in 24 bytes of contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. For example,

```
'X9.9-1  MIDDLE  MACLEN4 '
```

The order of the *rule_array* keywords is not fixed.

You can specify one of the MAC processing rules and then choose one of the segmenting control keywords and one of the MAC length keywords. The *rule_array* keywords are described in Table 105.

Table 105. Keywords for MAC Generate control information

Keyword	Description
MAC process rules (One, optional)	
EMVMAC	EMV padding rule with a single-length MAC key. The *key_identifier* parameter must identify a single-length MAC or a single-length **DATA** key. The text is always padded with 1 - 8 bytes so the resulting text length is a multiple of eight bytes. The first pad character is X'80'. The remaining pad characters are X'00'.
EMVMACD	EMV padding rule with a double-length MAC key. The *key_identifier* parameter must identify a double-length MAC key. The padding rules are the same as for keyword **EMVMAC**.
TDES-MAC	ANSI X9.9-1 procedure using ISO 16609 CBC mode triple-DES (TDES) encryption of the data. Uses a double-length key.
X9.19OPT	ANSI X9.19 optional double key MAC procedure. The *key_identifier* parameter must identify a double-length MAC key. The padding rules are the same as for keyword X9.9-1.
X9.9-1	ANSI X9.9-1 and X9.19 basic procedure. The *key_identifier* parameter must identify a single-length MAC or a single-length **DATA** key. X9.9-1 causes the MAC to be computed from all of the data. The text is padded only if the text length is not a multiple of eight bytes. If padding is required, the pad character X'00' is used. This is the default value.
Segmenting control (One, optional)	
FIRST	First call; this is the first segment of data from the application program.
LAST	Last call; this is the last data segment.
MIDDLE	Middle call; this is an intermediate data segment.
ONLY	Only call; segmenting is not employed by the application program. This is the default value.
MAC length and presentation (One, optional)	
HEX-8	Generates a 4-byte MAC value and presents it as 8 hexadecimal characters.
HEX-9	Generates a 4-byte MAC value and presents it as two groups of 4 hexadecimal characters with a space between the groups.
MACLEN4	Generates a 4-byte MAC value. This is the default value.
MACLEN6	Generates a 6-byte MAC value.
MACLEN8	Generates an 8-byte MAC value.

chaining_vector

Direction: Input/Output
Type: String

An 18-byte string that CCA uses as a system work area. Your application program must not change the data in this string. The chaining vector permits data to be chained from one invocation call to another.

On the first call, initialize this parameter as binary zeros.

mac

Direction: Output
Type: String

The 8-byte or 9-byte field in which the verb returns the MAC value if the segmenting rule is **ONLY** or **LAST**. Allocate an 8-byte field for MAC values of four bytes, six bytes, eight bytes, or **HEX-8**. Allocate a 9-byte MAC field if you specify **HEX-9** in the *rule_array* parameter.

Restrictions

The restrictions for CSNBMGN.

It might seem intuitive that a **DATAM** key should also be usable for the MAC Generate verb, and a **DATAMV** key for the MAC Verify verb, with the CPACF exploitation layer. However, this would violate the security restrictions imposed by the user when the user creates a key of type **DATAM** or **DATAMV**. A DES key that has been translated for use with the CPACF (see "CPACF support" on page 12) can be used with CPACF DES encrypt and decrypt operations, an operation that is by definition not allowed for a **DATAM** or **DATAMV** key type. Also note that by definition both through z/OS CCA-ICSF and in this S390 Linux CCA access layer, a **DATA** key of 16 bytes or 24 bytes in length is restricted from use with the **X9.19OPT** and **EMVMACD** *rule_array* keyword specified MAC algorithms. The only available MAC algorithm for a 16-byte or 24-byte **DATA** key is the TDES-MAC algorithm.

Also note that the CPACF exploitation layer is activated only for MAC Generate or MAC Verify calls that specify the **ONLY** *rule_array* keyword for segmenting control (this is the default segmenting control if no segmenting control *rule_array* keyword is specified). The reason for this is that the intermediate MAC context for normal CEX*C calls to MAC Generate and MAC Verify is protected by the adapter Master Key. Because the same security cannot be provided for intermediate results from the host-based CPACF exploitation layer (they are returned in the clear by the CPACF) the **FIRST**, **MIDDLE**, and **LAST** segmenting control keywords will direct operations to the CEX*C.

Required commands

The required commands for CSNBMGN.

This verb requires the **MAC Generate** command (offset X'0010') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBMGN.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBMGNJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBMGNJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           byte[]            key_identifier,
           hikmNativeInteger text_length,
           byte[]            text,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            chaining_vector,
           byte[]            mac);
```

MAC Generate2 (CSNBMGN2)

Use the MAC Generate2 service to generate a keyed hash message authentication code (HMAC) or a ciphered message authentication code (CMAC) for the message string provided as input. A MAC key with key usage that can be used for generate is required to calculate the MAC.

The MAC generate key must be in a variable-length HMAC key token for HMAC and an AES MAC token for CMAC.

Format

The format of CSNBMGN2.

```
CSNBMGN2(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_identifier_length,
             key_identifier,
             message_text_length,
             message_text,
             chaining_vector_length,
             chaining_vector,
             MAC_length,
             MAC)
```

Parameters

The parameters for CSNBMGN2.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. This value must be 0, 1, 2, or 3.

rule_array

Direction: Input
Type: String array

The **rule_array** contains keywords that provide control information to the callable service. The keywords must be in contiguous storage with each of the keywords left-justified in its own 8-byte location and padded on the right with blanks. The **rule_array** keywords are described in Table 106.

Table 106. Keywords for MAC Generate2 control information

Keyword	Description
Token algorithm (One, required)	
AES	Specifies the use of the AES CMAC algorithm to generate a MAC.
HMAC	Specifies the use of the HMAC algorithm to generate a MAC.
Hash method (One, required for HMAC only)	
SHA-1	Specifies the use of the SHA-1 hash method.
SHA-224	Specifies the use of the SHA-224 hash method.
SHA-256	Specifies the use of the SHA-256 hash method.
SHA-384	Specifies the use of the SHA-384 hash method.
SHA-512	Specifies the use of the SHA-512 hash method.
Segmenting Control (One, optional)	
ONLY	Only call. Segmenting is not employed by the application program. This is the default value.
FIRST	First call. This is the first segment of data from the application program.
MIDDLE	Middle call. This is an intermediate data segment.
LAST	Last call. This is the last data segment.

key_identifier_length

Direction: Input/Output
Type: String

The length in bytes of the **key_identifier** parameter. If the **key_identifier** parameter contains a label, the value must be 64. Otherwise, the value must be at least the actual token length, up to 725.

key_identifier

Direction: Input/Output
Type: String

The identifier of the key to generate the MAC. The key identifier is an operational token or the key label of an operational token in key storage.

For the HMAC algorithm, the key algorithm must be HMAC and the key usage fields must indicate GENONLY or GENERATE and the hash method selected. For the AES algorithm, the key algorithm must be AES, the key type must be MAC, and the key usage fields must indicate GENONLY or GENERATE and must indicate CMAC.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

message_text_length

Direction: Input
Type: Integer

The length of the text you supply in the **message_text** parameter. The maximum length of **text** is 214783647 bytes. For FIRST and MIDDLE calls, the **message_text_length** must be:

- a multiple of 64 for the SHA-1, SHA-224, and SHA-256 hash methods
- a multiple of 128 for the SHA-384 and SHA-512 hash methods
- a multiple of 16 for the AES CMAC method.

message_text

Direction: Input
Type: String

The application-supplied text for which the MAC is generated.

chaining_vector_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable specifying the length in bytes of the **chaining_vector** parameter. The value must be 128.

chaining_vector

Direction: Input/Output
Type: String

A pointer to a string variable containing a work area that the security server uses to carry segmented data between procedure calls. When the segmenting control is **FIRST** or **ONLY**, this value is ignored, but must be declared.

MAC_length

Direction: Input/Output
Type: Integer

The length of the **MAC** parameter in bytes. This parameter is updated to the actual length of the **MAC** parameter on output. For HMAC, the minimum value is 4 and the maximum value is 64. For AES, the value must be 16.

MAC

Direction: Output
Type: String

The field in which the callable service returns the MAC value if the segmenting rule is ONLY or LAST.

Restrictions

The restrictions for CSNBMGN2.

None.

Required commands

The required commands for CSNBMGN2.

The MAC Generate2 verb requires the commands listed in Table 107 to be enabled in the active role based on the keyword specified for the process rule:

Table 107. CSNBMGN2 access control points.

Hash method / Rule-array keyword	Offset	Command
AES CMAC	X'0336'	**MAC Generate2 - AES CMAC**
SHA-1	X'00E4'[1)]	**HMAC Generate - SHA-1**
SHA-224	X'00E5'[1)]	**HMAC Generate - SHA-224**
SHA-256	X'00E6'[1)]	**HMAC Generate - SHA-256**
SHA-384	X'00E7'[1)]	**HMAC Generate - SHA-384**
SHA-512	X'00E8'[1)]	**HMAC Generate - SHA-512**
[1)] A role with this offset enabled can also use the HMAC Generate verb.		

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBMGN2.

None.

Related information

Additional information about CSNBMGN2.

The MAC Verify2 verb is described in "MAC Verify2 (CSNBMVR2)" on page 390.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBMGN2J.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBMGN2J(
           hikmNativeInteger    return_code,
           hikmNativeInteger    reason_code,
           hikmNativeInteger    exit_data_length,
           byte[]               exit_data,
           hikmNativeInteger    rule_array_count,
           byte[]               rule_array,
           hikmNativeInteger    key_identifier_length,
           byte[]               key_identifier,
           hikmNativeInteger    message_text_length,
           byte[]               message_text,
           hikmNativeInteger    chaining_vector_length,
           byte[]               chaining_vector,
           hikmNativeInteger    MAC_length,
           byte[]               MAC);
```

MAC Verify (CSNBMVR)

When the receiver gets a message, an application program calls the MAC Verify verb.

This verb verifies a 4-byte, 6-byte, or 8-byte Message Authentication Code (MAC) for an application-supplied text string. This verb verifies a MAC by generating another MAC and comparing it with the MAC received with the message. This process takes place entirely within the secure module on the coprocessor. If the two codes are the same, the message sent was the same one received. A return code indicates whether the MACs are the same. The generated MAC never appears in storage and is not revealed outside the cryptographic feature.

The MAC Verify verb can use any of the following methods to generate the MAC for authentication:

- The ANSI X9.9-1 single key algorithm, a single-length MAC verification or MAC generation key (or a data-encrypting key), and the message text.
- The ANSI X9.19 optional double key algorithm, a double-length MAC verification or MAC generation key and the message text.
- Using the Europay, MasterCard and Visa (EMV) padding rules.

The method used to verify the MAC should correspond with the method used to generate the MAC.

Host CPU acceleration: CPACF

Only keys with a key type of **DATA** can be used successfully with the CPACF exploitation layer through this verb.

Specifically, a **DATA** key has a CV (Control Vector) of all X'00' bytes for all active bytes of the CV (eight bytes for 8-byte DES keys, 16 bytes for 16-byte DES keys, and 16 bytes for 24-byte DES keys).

For details about CPACF, see “CPACF support” on page 12.

Format

The format of CSNBMVR.

```
CSNBMVR(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             key_identifier,
             text_length,
             text,
             rule_array_count,
             rule_array,
             chaining_vector,
             mac)
```

Parameters

The parameters for CSNBMVR.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters common to all verbs” on page 20.

key_identifier

Direction: Input/Output
Type: String

The 64-byte key label or internal key token that identifies a single-length or double-length MAC verify key, a single-length or double-length MAC generation key, or a single-length **DATA** key. The type of key depends on the MAC process rule in the *rule_array* parameter.

text_length

Direction: Input
Type: Integer

The length of the clear text you supply in the *text* parameter. If the *text_length* parameter is not a multiple of eight bytes and if the **ONLY** or **LAST** keyword of the *rule_array* parameter is called, the text is padded in accordance with the processing rule specified.

text

Direction: Input
Type: String

The application-supplied text for which the MAC is verified.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0, 1, 2, or 3.

rule_array

Direction: Input
Type: String array

Zero to three keywords that provide control information to the verb. The keywords are described in Table 108. The keywords must be in 24 bytes of contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. For example,

```
'X9.9-1  MIDDLE  MACLEN4 '
```

The order of the *rule_array* keywords is not fixed.

You can specify one of the MAC processing rules, and then choose one of the segmenting control keywords and one of the MAC length keywords. The *rule_array* keywords are described in Table 108.

Table 108. Keywords for MAC Verify control information

Keyword	Description
MAC process rules (One, optional)	
EMVMAC	EMV padding rule with a single-length MAC key. The *key_identifier* parameter must identify a single-length **MAC**, **MACVER**, or **DATA** key. The text is always padded with 1 - 8 bytes, so that the resulting text length is a multiple of eight bytes. The first pad character is X'80'. The remaining pad characters are X'00'.

Table 108. Keywords for MAC Verify control information (continued)

Keyword	Description
EMVMACD	EMV padding rule with a double-length MAC key. The *key_identifier* parameter must identify a double-length **MAC** or **MACVER** key. The padding rules are the same as for **EMVMAC**.
TDES-MAC	ANSI X9.9-1 procedure using ISO 16609 CBC mode triple-DES (TDES) encryption of the data. Uses a double-length key.
X9.9-1	ANSI X9.9-1 and X9.19 basic procedure. The *key_identifier* parameter must identify a single-length **MAC**, **MACVER**, or **DATA** key. **X9.9-1** causes the MAC to be computed from all the data. The text is padded only if the text length is not a multiple of eight bytes. If padding is required, the pad character X'00' is used. This is the default value.
X9.19OPT	ANSI X9.19 optional double-length MAC procedure. The *key_identifier* parameter must identify a double-length **MAC** or **MACVER** key. The padding rules are the same as for **X9.9-1**.
Segmenting control (Optional)	
FIRST	First call. This is the first segment of data from the application program.
LAST	Last call. This is the last data segment.
MIDDLE	Middle call. This is an intermediate data segment.
ONLY	Only call. The application program does not employ segmenting. This is the default value.
MAC length and presentation (Optional)	
HEX-8	Verifies a 4-byte MAC value represented as 8 hexadecimal characters.
HEX-9	Verifies a 4-byte MAC value represented as two groups of 4 hexadecimal characters with a space character between the groups.
MACLEN4	Verifies a 4-byte MAC value. This is the default value.
MACLEN6	Verifies a 6-byte MAC value.
MACLEN8	Verifies an 8-byte MAC value.

chaining_vector

Direction: Input/Output
Type: String

An 18-byte string CCA uses as a system work area. Your application program must not change the data in this string. The chaining vector permits data to be chained from one invocation call to another.

On the first call, initialize this parameter to binary zeros.

mac

Direction: Input
Type: String

The 8-byte or 9-byte field that contains the MAC value you want to verify. The value in the field must be left-aligned and padded with zeros. If you specified the **HEX-9** keyword in the *rule_array* parameter, the input MAC is nine bytes in length.

Restrictions

The restrictions for CSNBMVR.

It might seem intuitive that a **DATAM** key should also be usable for the MAC Generate verb, and a **DATAMV** key for the MAC Verify verb, with the CPACF exploitation layer. However, this would violate the security restrictions imposed by the user when the user creates a key of type **DATAM** or **DATAMV**. A DES key

that has been translated for use with the CPACF (see "CPACF support" on page 12) can be used with CPACF DES encrypt and decrypt operations, an operation that is by definition not allowed for a **DATAM** or **DATAMV** key type. Also note that by definition both through z/OS CCA-ICSF and in this S390 Linux CCA access layer, a **DATA** key of 16 bytes or 24 bytes in length is restricted from use with the **X9.19OPT** and **EMVMACD** *rule_array* keyword specified MAC algorithms. The only available MAC algorithm for a 16-byte or 24-byte **DATA** key is the **TDES-MAC** algorithm.

Also note that the CPACF exploitation layer is activated only for MAC Generate or MAC Verify calls that specify the **ONLY** *rule_array* keyword for segmenting control (this is the default segmenting control if no segmenting control *rule_array* keyword is specified). The reason for this is that the intermediate MAC context for normal CEX*C calls to MAC Generate and MAC Verify is protected by the adapter Master Key. Because the same security cannot be provided for intermediate results from the host-based CPACF exploitation layer (they are returned in the clear by the CPACF) the **FIRST**, **MIDDLE**, and **LAST** segmenting control keywords will direct operations to the CEX*C.

Required commands

The required commands for CSNBMVR.

This verb requires the **MAC Verify** command (offset X'0011') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBMVR.

To verify a MAC in one call, specify the **ONLY** keyword on the segmenting rule keyword for the *rule_array* parameter. For two or more calls, specify the **FIRST** keyword for the first input block, **MIDDLE** for intermediate blocks (if any), and **LAST** for the last block.

For a given text string, the MAC resulting from the verification process is the same regardless of how the text is segmented or how it was segmented when the original MAC was generated.

Related information

Additional information for CSNBMVR.

The MAC Generate verb is described in "MAC Generate (CSNBMGN)" on page 378.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBMVRJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBMVRJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[]            exit_data,
    byte[]            key_identifier,
    hikmNativeInteger text_length,
    byte[]            text,
    hikmNativeInteger rule_array_count,
    byte[]            rule_array,
    byte[]            chaining_vector,
    byte[]            MAC);
```

MAC Verify2 (CSNBMVR2)

Use the MAC Verify2 verb to verify a keyed hash message authentication code (HMAC) or a ciphered message authentication code (CMAC) for the message text provided as input. A MAC key with key usage that can be used for verify is required to verify the MAC.

The MAC verify key must be in a variable-length HMAC key token for HMAC and an AES MAC token for CMAC.

Format

The format of CSNBMVR2.

```
CSNBMVR2(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_identifier_length,
             key_identifier,
             message_text_length,
             message_text,
             chaining_vector_length,
             chaining_vector,
             MAC_length,
             MAC)
```

Parameters

The parameters for CSNBMVR2.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. This value must be 1, 2, or 3.

rule_array

Direction: Input
Type: String array

The **rule_array** contains keywords that provide control information to the MAC Verify2 callable service. The keywords are described in Table 109.

Table 109. Keywords for MAC Verify2 control information

Keyword	Description
Token algorithm (One, required)	
AES	Specifies the use of the AES CMAC algorithm to generate a MAC.
HMAC	Specifies the use of the HMAC algorithm to generate a MAC.
Hash method (One required for HMAC only)	
SHA-1	Specifies the use of the SHA-1 hash method.
SHA-224	Specifies the use of the SHA-224 hash method.
SHA-256	Specifies the use of the SHA-256 hash method.
SHA-384	Specifies the use of the SHA-384 hash method.
SHA-512	Specifies the use of the SHA-512 hash method.
Segmenting Control (One optional)	
FIRST	First call. This is the first segment of data from the application program.
LAST	Last call. This is the last data segment.
MIDDLE	Middle call. This is an intermediate data segment.
ONLY	Only call. Segmenting is not employed by the application program. This is the default value.

key_identifier_length

Direction: Input
Type: Integer

Length of the **key_identifier** parameter in bytes. If the key_identifier parameter contains a label, the value must be 64. Otherwise, the value must be at least the actual token length, up to 725.

key_identifier

Direction: Input/Output
Type: String

The identifier of the key to verify the MAC. The key identifier is an operational token or the key label of an operational token in key storage.

For the HMAC algorithm, the key algorithm must be HMAC and the key usage fields must indicate GENERATE or VERIFY and the hash method selected. For the AES algorithm, the key algorithm must be AES, the key type must be MAC, and the key usage fields must indicate GENERATE or VERIFY and must indicate CMAC.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

message_text_length

Direction: Input
Type: Integer

The length of the clear text you supply in the **message_text** parameter. The maximum length of text is 214783647 bytes. For FIRST and MIDDLE calls, the **message_text_length** must be:

- a multiple of 64 for the SHA-1, SHA-224, and SHA-256 hash methods,
- a multiple of 128 for the SHA-384 and SHA-512 hash methods,
- a multiple of 16 for the AES CMAC method.

message_text

Direction: Input
Type: String

The application-supplied text for which the MAC is generated.

chaining_vector_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **chaining_vector** parameter. The value must be 128.

chaining_vector

Direction: Input/Output
Type: String

A pointer to a string variable containing a work area that the security server uses to carry segmented data between procedure calls. When the segmenting control is **FIRST** or **ONLY**, this value is ignored but must be declared. Important: Application programs must not alter the contents of this variable between related **FIRST**, **MIDDLE**, and **LAST** calls.

MAC_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **MAC** parameter. The value must be equal to the number of MAC bytes to be verified, up to a maximum of 64.

MAC

Direction: Input
Type: String

The field that contains the MAC value you want to verify.

Restrictions

The restrictions for CSNBMVR2.

None.

Required commands

The required commands for CSNBMVR2.

The CSNBMVR2 verb requires the commands listed in Table 110 on page 393 to be enabled in the active role.

Table 110. Required commands for CSNBMVR2.

Hash method/Rule-array keword	Offset	Command
AES	X'0337'	**MAC Verify2 - AES CMAC**
SHA-1	X'00F7' [1)]	**HMAC Verify - SHA-1**
SHA-224	X'00F8'[1)]	**HMAC Verify - SHA-224**
SHA-256	X'00F9'[1)]	**HMAC Verify - SHA-256**
SHA-384	X'00FA'[1)]	**HMAC Verify - SHA-384**
SHA-512	X'00FB'[1)]	**HMAC Verify - SHA-512**
[1)] A role with this offset enabled can also use theHMAC Verify verb.		

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBMVR2.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBMVR2J.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBMVR2J(
    hikmNativeInteger   return_code,
    hikmNativeInteger   reason_code,
    hikmNativeInteger   exit_data_length,
    byte[]              exit_data,
    hikmNativeInteger   rule_array_count,
    byte[]              rule_array,
    hikmNativeInteger   key_identifier_length,
    byte[]              key_identifier,
    hikmNativeInteger   message_text_length,
    byte[]              message_text,
    hikmNativeInteger   chaining_vector_length,
    byte[]              chaining_vector,
    hikmNativeInteger   MAC_length,
    byte[]              MAC);
```

MDC Generate (CSNBMDG)

Use this verb to create a 128-bit hash value (Modification Detection Code) on a data string whose integrity you intend to confirm.

After using this verb to generate an MDC, you can compare the MDC to a known value or communicate the value to another entity so that they can compare the MDC hash value to one that they calculate. This verb enables you to perform the following tasks:

- Specify the two-encipherment or four-encipherment version of the algorithm.
- Segment your text into a series of verb calls.
- Use the default or a keyed-hash algorithm.

The user must enable the **MDC Generate** command (offset X'008A') with a Trusted Key Entry (TKE) workstation before using this verb.

For a description of the MDC calculations, see "Modification Detection Code calculation" on page 930.

Specifying two or four encipherments: Four encipherments per algorithm round improve security; two encipherments per algorithm round improve performance. To specify the number of encipherments, use the **MDC-2**, **MDC-4**, **PADMDC-2**, or **PADMDC-4** keyword with the *rule_array* parameter. Two encipherments create results that differ from four encipherments; ensure that the same number of encipherments are used to verify the MDC.

Segmenting text: This verb lets you segment text into a series of verb calls. If you can present all of the data to be hashed in a single invocation of the verb (32 MB) of data, use the *rule_array* keyword **ONLY**. Alternatively, you can segment your text and present the segments with a series of verb calls. Use the *rule_array* keywords and **LAST** for the first and last segments. If more than two segments are used, specify the *rule_array* keyword **MIDDLE** for the additional segments.

Between verb calls, unprocessed text data and intermediate information from the partial MDC calculation is stored in the *chaining_vector* variable and the MDC key in the *MDC* variable. During segmented processing, the application program must not change the data in either of these variables.

Keyed hash: This verb can be used with a default key, or as a keyed-hash algorithm. A default key is used whenever the **ONLY** or **FIRST** segmenting and key control keywords are used. To use the verb as a keyed-hash algorithm, do the following:

1. On the first call to the verb, place the non-null key into the *MDC* variable.
2. Ensure that the *chaining_vector* variable is set to null (18 bytes of X'00').
3. Decide if the text will be processed in a single segment or multiple segments.
   - For a single segment of text, use the **LAST** keyword.
   - For multiple segments of text, begin with the **MIDDLE** keyword and continue using the **MIDDLE** keyword up to the final segment of text. For the final segment, use the **LAST** keyword.

As with the default key, you must not alter the value of the *MDC* or *chaining_vector* variables between calls.

Format

The format of CSNBMDG.

```
CSNBMDG(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             text_length,
             text,
             rule_array_count,
             rule_array,
             chaining_vector,
             MDC)
```

Parameters

The parameters for CSNBMDG.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

text_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *text* variable. See "Restrictions" on page 396.

text

Direction: Input
Type: String

A pointer to a string variable containing the text for which the verb calculates the MDC value.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value can be 0, 1, or 2.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. A keyword specifies the method for calculating the RSA digital signature. Each keyword is left-aligned in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage. The *rule_array* keywords are described in Table 111.

Table 111. Keywords for MDC Generate control information

Keyword	Description
Segmenting and key control (One, optional)	
ONLY	Specifies that segmenting is not used and the default key is used. This is the default.
FIRST	Specifies the first segment of text, and use of the default key.
MIDDLE	Specifies an intermediate segment of text, or the first segment of text and use of a user-supplied key.
LAST	Specifies the last segment of text, or that segmenting is not used, and use of a user-supplied key.
Algorithm mode (One, optional)	
MDC-2	Specifies two encipherments for each 8-byte block using MDC procedures. This is the default.
MDC-4	Specifies four encipherments for each 8-byte block using MDC procedures.
PADMDC-2	Specifies two encipherments for each 8-byte block using PADMDC procedures.
PADMDC-4	Specifies four encipherments for each 8-byte block using PADMDC procedures.

chaining_vector

Direction: Input/Output
Type: String

A pointer to an 18-byte string variable the security server uses as a work area to hold segmented data between verb invocations.

Important: When segmenting text, the application program must not change the data in this string between verb calls to the MDC Generate verb.

MDC

Direction: Input/Output
Type: String

A pointer to a user-supplied MDC key or to a 16-byte string variable containing the MDC value. This value can be the key that the application program provides. This variable is also used to hold the intermediate MDC result when segmenting text.

IMPORTANT: When segmenting text, the application program must not change the data in this string between verb calls to the MDC Generate verb.

Restrictions

The restrictions for CSNBMDG.

- When padding is requested (by specifying an algorithm mode keyword of **PADMDC-2** or **PADMDC-4**), a text length of zero is valid for any segment-control keyword specified in the *rule_array* variable **FIRST**, **MIDDLE**, **LAST**, or **ONLY**). When **LAST** or **ONLY** is specified, the supplied text is padded with X'FF' bytes and a padding count in the last byte to bring the total text length to the next multiple of 8 that is greater than or equal to 16.
- When no padding is requested (by specifying an algorithm mode keyword of **MDC-2** or **MDC-4**), the total length of text provided (over a single or segmented calls) must be a minimum of 16 bytes and a multiple of eight bytes. For segmented calls (that is, segmenting and key control keyword is not **ONLY**), a text length of zero is valid on any of the calls.

Required commands

The required commands for CSNBMDG.

In releases prior to CCA 4.1.0 and for installations without CPACF support this verb requires the **MDC Generate** command (offset X'008A') to be enabled in the active role. This command is no longer required in CCA 4.1.0 when CPACF clear-key function is available (KM, function 1), and enabled for CCA use (environment variable CSU_HCPUACLR is set to '1', the default value).

The user must enable the **MDC Generate** command with a Trusted Key Entry (TKE) workstation before using this verb.

Usage notes

The usage notes for CSNBMDG.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBMDGJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBMDGJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger text_length,
           byte[]            text_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            chaining_vector,
           byte[]            MDC);
```

One-Way Hash (CSNBOWH)

Use the One-Way Hash verb to generate a one-way hash on specified text.

These SHA based hashing functions are supported with the CPACF exploitation layer: SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512. For details about CPACF, see "CPACF support" on page 12.

Format

The format of CSNBOWH.

```
CSNBOWH(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             text_length,
             text,
             chaining_vector_length,
             chaining_vector,
             hash_length,
             hash)
```

Parameters

The parameters for CSNBOWH.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1 or 2.

rule_array

Direction: Input
Type: String array

These keywords provide control information to the verb. The optional chaining flag keyword indicates whether calls to this verb are chained together logically to overcome buffer size limitations. Each keyword is left-aligned in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage. The *rule_array* keywords are described in Table 112.

Table 112. Keywords for One-Way Hash control information

Keyword	Description
Hash method (One, required). The SHA-based hashing functions use CPACF by default. For details about CPACF, see "CPACF support" on page 12.	
MD5	Hash algorithm is MD5 algorithm. Use this hash method for PKCS-1.0 and PKCS-1.1. Length of hash generated is 16 bytes.
RPMD-160	Hash algorithm is RIPEMD-160. Length of hash generated is 20 bytes.
SHA-1	Hash algorithm is SHA-1 algorithm. Length of hash generated is 20 bytes.
SHA-224	Hash algorithm is SHA-224 algorithm. Length of hash generated is 20 bytes.
SHA-256	Hash algorithm is SHA-256 algorithm. Length of hash generated is 20 bytes.
SHA-384	Hash algorithm is SHA-384 algorithm. Length of hash generated is 20 bytes.
SHA-512	Hash algorithm is SHA-512 algorithm. Length of hash generated is 20 bytes.
Chaining flag (One, optional)	
FIRST	Specifies this is the first call in a series of chained calls. Intermediate results are stored in the *hash* field.
LAST	Specifies this is the last call in a series of chained calls.
MIDDLE	Specifies this is a middle call in a series of chained calls. Intermediate results are stored in the *hash* field.
ONLY	Specifies this is the only call and the call is not chained. This is the default.

text_length

Direction: Input
Type: Integer

The length of the *text* parameter in bytes.

Note: If you specify the **FIRST** or **MIDDLE** keyword, the text length must be a multiple of the block size of the hash method. For **MD5**, **RPMD-160**, and **SHA-1**, this is a multiple of 64 bytes.

For **ONLY** and **LAST**, this verb performs the required padding according to the algorithm specified.

text

Direction: Input
Type: String

The application-supplied text on which this verb performs the hash.

chaining_vector_length

Direction: Input

Type: Integer

The byte length of the *chaining_vector* parameter. This must be 128 bytes.

chaining_vector

Direction: Input/Output
Type: String

This field is a 128-byte work area. Your application must not change the data in this string. The chaining vector permits chaining data from one call to another.

hash_length

Direction: Input
Type: Integer

The length of the supplied *hash* field in bytes.

Note: For **SHA-1** and **RPMD-160** this must be a minimum of 20 bytes. For **MD5** this must be a minimum of 16 bytes.

hash

Direction: Input/Output
Type: String

This field contains the hash, left-aligned. The processing of the rest of the field depends on the implementation. If you specify the **FIRST** or **MIDDLE** keyword, this field contains the intermediate hash value. Your application must not change the data in this field between the sequence of **FIRST**, **MIDDLE**, and **LAST** calls for a specific message.

Restrictions

The restrictions for CSNBOWH.

None.

Required commands

The required commands for CSNBOWH.

None.

Usage notes

The usage notes for CSNBOWH.

Although the algorithms accept zero bit length text, it is not supported for any hashing method.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBOWHJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBOWHJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  hikmNativeInteger rule_array_count,
  byte[]            rule_array,
  hikmNativeInteger text_length,
  byte[]            text,
  hikmNativeInteger chaining_vector_length,
  byte[]            chaining_vector,
  hikmNativeInteger hash_length,
  byte[]            hash);
```

Chapter 9. Key storage mechanisms

You can use key storage mechanisms and the associated key record verbs to perform operations on key tokens and key records located in AES, DES, and PKA key storage.

A key-token record consists of a key-token name (key label) and a key token of format null, internal, or external. The operations to be performed are: creating, writing, reading, listing, and deleting key tokens or key records.

The verbs described in this chapter include:

Key labels and key-storage management

Use these verbs to manage AES, DES, and PKA key storage.

The CCA software manages key storage as an indexed repository of key records. Access key storage using a key label with verbs that have a key-label or key-identifier parameter.

An independent key-storage system can be used to manage records for AES key records, DES key records, and PKA key records:

AES key storage
Holds null and internal AES key tokens

DES key storage
Holds null, external, and internal DES key tokens

PKA key storage
Holds null PKA key tokens, and both internal and external public and private PKA key tokens

Private RSA keys are generated and optionally retained within the coprocessor using the PKA Key Generate verb. Depending on the other uses for coprocessor storage, between 75 and 150 keys can normally be retained within the coprocessor.

Key storage must be initialized before any records are created. Before a key token can be stored in key storage, a key-storage record must be created using the AES Key Record Create, DES Key Record Create, or PKA Key Record Create verb.

Use the AES Key Record Delete, DES Key Record Delete, or PKA Key Record Delete verb to delete a key token from a key record, or to entirely delete the key record from key storage.

Use the AES Key Record List, DES Key Record List, or PKA Key Record List verb to determine the existence of key records in key storage. These list verbs create a key-record-list file with information about select key records. The wildcard character, represented by an asterisk (*), is used to obtain information about multiple key records. The file can be read using conventional workstation-data-management services.

Individual key tokens can be read using the AES Key Record Read, DES Key Record Read, and PKA Key Record Read verbs or written using the AES Key Record Write, DES Key Record Write, and PKA Key Record Write verbs.

Environment variables for the key storage file

These environment variables contain the name of the key storage file.

There is one for each type: AES, DES, and PKA.

CSUAESDS
AES key storage file.

CSUDESDS
DES key storage file.

CSUPKADS
PKA key storage file.

Key-label content

Use a key label to identify a record in key storage managed by a CCA implementation.

The key label must be left-aligned in the 64-byte string variable used as input to the verb. Some verbs use a key label while others use a key identifier. Calls that use a key identifier accept either a key token or a key label.

A key-label character string has the following properties:

- It contains 64 bytes of data.
- The first character is within the range X'20' - X'FE'. If the first character is within this range, the input is treated as a key label, even if it is otherwise not valid. Inputs beginning with a byte valued in the range X'00' - X'1F' are considered to be some form of key token. A first byte valued to X'FF' is not valid.
- The first character of the key label cannot be numeric (0 - 9).
- The label is ended by a space character on the right (in ASCII it is X'20', and in EBCDIC it is X'40'). The remainder of the 64-byte field is padded with space characters.

- Construct a label with 1 - 7 name tokens, each separated by a period (.). The key label must not end with a period.
- A name token consists of 1 - 8 characters in the character set A - Z, 0 - 9, and three additional characters relating to different character symbols in the various national language character sets as listed in Table 113.

Table 113. Valid symbols for the name token

ASCII systems	EBCDIC systems	USA graphic (for reference)
X'23'	X'7B'	#
X'24'	X'5B'	$
X'40'	X'7C'	@

The alphabetic and numeric characters and the period should be encoded in the normal character set for the computing platform that is in use, either ASCII or EBCDIC.

Note:

1. Some CCA implementations accept the characters a - z and fold these to their uppercase equivalents, A - Z. For compatibility reasons, only use the uppercase alphabetic characters.
2. Some implementations internally transform the EBCDIC encoding of a key label to an ASCII string. Also, the label might be put in tokenized form by dropping the periods and formatting each name token into 8-byte groups, padded on the right with space characters.

Some verbs accept a key label containing a wild card represented by an asterisk (*). (X'2A' in ASCII; X'5C' in EBCDIC). When a verb permits the use of a wild card, the wild card can appear as the first character, as the last character, or as the only character in a name token. Any of the name tokens can contain a wild card.

Examples of valid key labels include the following:

```
A
ABCD.2.3.4.5555
ABCDEFGH
BANKSYS.XXXXX.43*.PDQ
```

Examples of key labels that are not valid are listed in Table 114.

Table 114. Key labels that are not valid

Key label not valid	Problem with key label
A/.B	A slash is an unacceptable character
ABCDEFGH9	Name token is greater than 8 characters
1111111.2.3.4.55555	First character cannot be numeric
A1111111.2.3.4.55555.6.7.8	Number of name tokens exceeds 7
BANKSYS.XXXXX.*43*.D	Number of wild cards exceeds 1
A.B.	Last character cannot be a period

Key storage with Linux on z Systems, in contrast to z/OS on z Systems

Key storage for IBM z/OS and for Linux on the IBM platforms other than IBM z Systems, diverged in design at their very inception.

Background information about master key management

There are four types (or sets) of master keys (Symmetric DES, AES, Asymmetric RSA (PKA), and APKA).

There are three master key registers for each of the four types of master key. In other words, there are a total of twelve master key registers.

The APKA master-key register set, introduced to CCA beginning with Release 4.1.0, is used to encrypt and decrypt the Object Protection Key (OPK) that is itself used to wrap the key material of an Elliptic Curve Cryptography (ECC) key. ECC keys are asymmetric.

For each of the four types, there is a master key register in one of these three categories:

New master-key (NMK) register
This register holds a master key that is not yet usable for decrypting key tokens for normal cryptographic operations.

The NMK register can be in one of these states:

EMPTY
No key parts have been loaded yet.

PARTIALLY FULL
Some key parts have been loaded, but not the LAST key part. See "Master Key Process (CSNBMKP)" on page 122.

FULL The LAST key part has been loaded, but the SET command has not yet been called. See "Master Key Process (CSNBMKP)" on page 122.

Current master-key (CMK) register
This register holds a master key that can be used to decrypt internal key tokens for keys in use with normal cryptographic operations. Internal key tokens are protected under the master key; the keys are actually stored outside the adapter.

The CMK register can be in one of these states:

EMPTY
No valid key has yet been established with the SET command in the life of this adapter, or the adapter has been re-initialized to clear the master key registers.

VALID
A master key has been loaded with the SET command.

Old master-key (OMK) register
This is the master key that previously has been the CMK, before the master key that is now in the CMK register. The OMK register can also be used to decrypt internal key tokens, but for these keys a warning with return code 0 and reason code 2 is returned, along with the results from the requested cryptographic operation.

The OMK register can be in one of these states:

EMPTY
No valid key is in this register.

VALID
A master key that previously was in the CMK register has been

shifted to the OMK register by the SET command. The same invocation of the SET command also shifted the contents of the NMK register into the CMK register.

SET command

The SET command transfers the current master-key to the old master-key register, and the new master-key to the current master-key register. It then clears the new master-key register.

The SET command is invoked with "Master Key Process (CSNBMKP)" on page 122, and performs these commands:

1. The master key from the CMK register is copied to the OMK register.
2. The master key from the FULL NMK register is copied to the CMK register.
3. The NMK register status is changed to EMPTY.

Key Storage on z/OS (RTNMK-focused)

Design point - Keys should be re-enciphered to a master key in the NMK register.

This forces the following process to be followed when changing the master key:

- Load all the master key parts for a NMK, such that the LAST key part has been loaded, but the SET command has not been issued. Now the NMK register is in the FULL state.
- Re-encipher all of (for example: CKDS) an existing key storage to a copy of that key storage that is not online, using the RTNMK *rule_array* keyword of "Key Token Change (CSNBKTC)" on page 254 (for AES or DES) or "PKA Key Token Change (CSNDKTC)" on page 652 (for PKA), creating CKDS-pending. Keys in this copy are enciphered under the NMK register, and so are not usable for normal cryptographic operations.
- Invoke the SET command for the NMK. See "SET command." Now the master keys in the current CKDS are enciphered under the OMK (because of the shift), and are usable. Also, the master keys in the CKDS-pending are also usable because the NMK has now become the CMK.
- Delete the old CKDS and change CKDS-pending to be the normal CKDS, completing the process.

Key Storage for traditional IBM systems other than IBM z Systems (RTCMK-focused: Linux, AIX, Windows)

Design point - Keys should be re-enciphered to a master key in the CMK register.

This forces the following process to be followed when changing the master key:

- Load all the master key parts for a NMK, such that the LAST key part has been loaded, then issue the SET command. Now the previous OMK is gone, the previous CMK is now the OMK, and the CMK contains the newly-loaded value. See "SET command."
- Re-encipher all of an existing CCA host key storage data file's key tokens, which are enciphered under the OMK, to be enciphered under the CMK. This is done using the RTCMK *rule_arry* keyword of "Key Token Change (CSNBKTC)" on page 254 or "PKA Key Token Change (CSNDKTC)" on page 652.
  - This **immediately replaces operational keys** with the re-enciphered version.
  - The CCA key storage file has a data structure with the verification pattern of the most recently SET master key. The key storage implementation also allows

writing external tokens into the key storage. This means that external key tokens, and the internal key tokens encrypted under current master key, will be allowed into the key storage.

It is impossible with current implementation to use RTNMK together with CCA key storage.

- During the re-encipherment:
  - Some of the keys in the CCA key storage files are enciphered under the OMK (because of the shift) and are usable
  - Some of the keys in the CCA key storage files are enciphered under the CMK, either because they are new or because they have been re-enciphered.
  - No new key tokens can be created with the key wrapped using the OMK.

  Both types are usable for cryptographic operations.

Changing the master key for two or more adapters that have the same master key, with shared CCA key storage

Because the verification pattern of the CMK is stored in a header in key storage, changing the master key for a configuration of multiple adapters requires extra care.

The master key verification pattern in key storage has the following properties:

- It is checked once when a process starts.
- It is repopulated when the first CEX*C has its master key changed.

These two properties force the user to use the same process to change the master key for all CEX*C cards after the first CEX*C. If the process exits (such as when the application completes), then the next time that the application starts the key storage header will be checked and the master key verification pattern will reflect the newly SET master key, which will cause a future attempt to set that same master key to a second or third CEX*C to have a conflict with the key storage header. Therefore, using the same process to change the master keys in all the CEX*C adapters is the only way to proceed if CCA key storage is being used.

There are several ways to change the master keys, most of which do not suffer from this limitation:

- A TKE can be used to change the master keys for all the CEX*C adapters in a group.
- An operator can directly change the master keys for a domain on a CEX*C from an IBM z Systems management interface (physical access is needed).
- A user application built to use the **libcsulccamk.so** library for this purpose, which can be programmed to:
  1. Allocate a CEX*C by invoking "Cryptographic Resource Allocate (CSUACRA)" on page 109.
  2. Change the master key.
  3. Deallocate each adapter in the group before exiting, by invoking "Cryptographic Resource Deallocate (CSUACRD)" on page 112.
- Note that the included utility, named panel.exe, is not designed to change the master keys for all the cards in a group; this is a more sophisticated operation.

  For details about panel.exe, see "The panel.exe utility" on page 1003.

Key storage file ownership

The last user to access the key storage file owns it, due to the internals of the key storage functions.

The file is recreated after being compressed, and due to the file creation the owner is changed.

Having the set-group-id bit (g+s) on in the directory permission causes the file to be created with the group owner the same as the directory group owner. The group read/write permissions on the file then allow the other members of the group continued access to the file.

The Linux on z Systems approach

Because the CCA key storage design point for the Linux platform host release has always been CMK-focused, this design point was taken forward for the Linux on z Systems approach.

At this time, CCA host key storage does not support nor ship with an additional utility to manage the 'store-in-pending' approach to re-enciphering key tokens. This additional utility is necessary to work with use of the RTNMK keyword for “Key Token Change (CSNBKTC)” on page 254 and “PKA Key Token Change (CSNDKTC)” on page 652. Therefore, it is suggested that users wanting to make use of CCA host key storage management follow the 'RTCMK-focused' approach described in “Key Storage for traditional IBM systems other than IBM z Systems (RTCMK-focused: Linux, AIX, Windows)” on page 405.

However it is also desirable to provide as much host-support equivalence with the z/OS approach as possible, given that the underlying system is running on a z Systems platform and likely to collaborate with z/OS software. Therefore, the RTNMK keyword is provided for “Key Token Change (CSNBKTC)” on page 254 and “PKA Key Token Change (CSNDKTC)” on page 652 to allow users who have their own utility or key storage management facility to manage key tokens using the method most familiar from z/OS:

- The key tokens to be enciphered should be passed directly (not by label) to “Key Token Change (CSNBKTC)” on page 254 and “PKA Key Token Change (CSNDKTC)” on page 652 for re-encipherment, and stored outside CCA host key storage.
- When re-encipherment is complete and the “Master Key Process (CSNBMKP)” on page 122 SET' command has been issued, the re-enciphered key tokens can be reintroduced to CCA host key storage if desired, using the standard mechanisms.

AES Key Record Create (CSNBAKRC)

Use the AES Key Record Create verb to create a key-token record in AES key-storage.

The new key record can be a null AES key-token or a valid internal AES key-token. It is identified by the key label specified with the *key_label* parameter.

After creating an AES key-record, use any of the following verbs to add or update a key token in the record:

- AES Key Record Delete
- AES Key Record Write

- Key Generate
- Key Token Change
- Key Token Change2
- Symmetric Key Generate
- Symmetric Key Import
- Symmetric Key Import2

Note:

1. To delete a key record from AES key-storage, use the AES Key Record Delete verb.
2. AES key records are stored in the external key-storage file defined by the CSUAESDS environment variable.

Format

The format of CSNBAKRC.

```
CSNBAKRC (
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array_count,
             key_label,
             key_token_length,
             key_token)
```

Parameters

The parameters for CSNBAKRC.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0.

rule_array

Direction: Input
Type: String array

This parameter is ignored.

key_label

Direction: Input
Type: String

A pointer to a string variable containing the key label of the AES key-record to be created.

key_token_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *key_token* variable. If the value of the *key_token_length* variable is zero, a record with a null AES key-token is created.

key_token

Direction: Input
Type: String

A pointer to a string variable containing the key token being written to AES key-storage.

Restrictions

The restrictions for CSNBAKRC.

The record must have a unique label. Therefore, there cannot be another record in the AES key storage file with the same label and a different key type.

Required commands

The required commands for CSNBAKRC.

In order to access key storage, this verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBAKRC.

None.

Related information

Additional information about CSNBAKRC.

See “Key storage with Linux on z Systems, in contrast to z/OS on z Systems” on page 403.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBAKRCJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBAKRCJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  hikmNativeInteger rule_array_count,
  byte[]            rule_array,
  byte[]            key_label,
  hikmNativeInteger key_token_length,
  byte[]            key_token);
```

AES Key Record Delete (CSNBAKRD)

Use the AES Key Record Delete verb to perform one of the tasks listed in the AES key storage file.

- Overwrite (delete) a key token or key tokens in AES key-storage, replacing the key token of each selected record with a null AES key-token.
- Delete an entire key record or key records, including the key label and the key token of each selected record, from AES key-storage.

Identify a task with the *rule_array* keyword, and the key record or records with the *key_label* parameter. To identify multiple records, use a wild card (*) in the key label.

Note: AES key records are stored in the external key-storage file defined by the CSUAESDS environment variable.

Format

The format of CSNBAKRD.

```
CSNBAKRD (
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_label)
```

Parameters

The parameters for CSNBAKRD.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0 or 1.

rule_array

Direction: Input
Type: String array

A pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* keywords are described in Table 115.

Table 115. Keywords for AES Key Record Delete control information

Keyword	Description
Task (One, optional)	

Table 115. Keywords for AES Key Record Delete control information (continued)

Keyword	Description
TOKEN-DL	Deletes a key token from a key record in AES key storage. This is the default.
LABEL-DL	Deletes an entire key record, including the key label, from AES key storage.

key_label

Direction: Input
Type: String

A pointer to a string variable containing the key label of a key-token record or records in AES key-storage. Use a wild card (*) in the *key_label* variable to identify multiple records in key storage.

Restrictions

The restrictions for CSNBAKRD.

The record defined by the *key_label* must be unique.

Required commands

The required commands for CSNBAKRD.

In order to access key storage, this verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBAKRD.

None.

Related information

Additional information about CSNBAKRD.

See “Key storage with Linux on z Systems, in contrast to z/OS on z Systems” on page 403.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBAKRDJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBAKRDJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  hikmNativeInteger rule_array_count,
  byte[]            rule_array,
  byte[]            key_identifier);
```

AES Key Record List (CSNBAKRL)

The AES Key Record List verb creates a key-record-list file containing information about specified key records in key storage.

Information listed includes whether record validation is correct, the type of key, and the date and time the record was created and last updated.

Specify the key records to be listed using the key-label variable. To identify multiple key records, use the wild card (*) in the key label.

Note:

1. To list all the labels in key storage, specify the *key_label* parameter with *, *.*, *.*.*, and so forth, up to a maximum of seven name tokens (*.*.*.*.*.*.*).
2. AES key records are stored in the external key-storage file defined by the CSUAESDS environment variable.

This verb creates the key-record-list file and returns the name of the file and the length of the file name to the calling application. This file has a header record, followed by 0 - *n* detail records, where *n* is the number of key records with matching key-labels.

The file is kept in the `/opt/IBM/CCA/keys/deslist` directory (assuming the directory name was not changed during installation). These list files are created under the ownership of the environment of the user that requests the list service. Make sure the files created kept the same group ID as your installation requires. This can also be achieved by setting the 'set-group-id-on-execution' bit on in this directory. See the g+s flags in the **chmod** command for full details. Not doing this might cause errors to be returned on key-record-list verbs.

Format

The format of CSNBAKRL.

```
CSNBAKRL(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_label,
    dataset_name_length,
    dataset_name,
    security_server_name)
```

Parameters

The parameters for CSNBAKRL.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters common to all verbs” on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0.

rule_array

Direction: Input
Type: String array

A pointer to a string variable containing an array of keywords. This verb currently does not use keywords.

key_label

Direction: Input
Type: String

A pointer to a string variable containing the key label of a key-token record in key storage. In a key label, you can use a wild card (*) to identify multiple records in key storage.

dataset_name_length

Direction: Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data returned by the verb in the *dataset_name* variable. The maximum returned length is 64 bytes.

dataset_name

Direction: Output
Type: String

A pointer to a string variable containing the name of the file returned by the verb. The file contains the AES key-record information. When the verb stores a key-record-list file, it overlays any older file with the same name.

The file name returned by this verb is defined by the CSUAESLD environment variable.

This verb returns the file name as a fully qualified file specification (for example, /opt/IBM/CCA/keys/KYRLT*nnn*.LST), where *nnn* is the numeric portion of the name. This value increases by one every time that you use this verb. When this value reaches 999, it resets to 001.

security_server_name

Direction: Output
Type: String

A pointer to a string variable. The information in this variable is not currently used, but the variable must be declared.

Restrictions

The restrictions for CSNBAKRL.

None.

Required commands

The required commands for CSNBAKRL.

In order to access key storage, this verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBAKRL.

None.

Related information

Additional information about CSNBAKRL.

See "Key storage with Linux on z Systems, in contrast to z/OS on z Systems" on page 403.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBAKRLJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBAKRLJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  hikmNativeInteger rule_array_count,
  byte[]            rule_array,
  byte[]            key_label,
  hikmNativeInteger data_set_name_length,
  byte[]            data_set_name,
  byte[]            security_server_name);
```

AES Key Record Read (CSNBAKRR)

Use the AES Key Record Read verb to read a key-token record from AES key-storage and return a copy of the key token to application storage.

The returned key token can be null. In this event, the *key_length* variable contains a value of 64 and the *key-token* variable contains 64 bytes of X'00' beginning at offset 0.

Note: AES key records are stored in the external key-storage file defined by the CSUAESDS environment variable.

Format

The format of CSNBAKRR.

```
CSNBAKRR (
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_label,
             key_token_length,
             key_token)
```

Parameters

The parameters for CSNBAKRR.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0.

rule_array

Direction: Input
Type: String array

This parameter is ignored.

key_label

Direction: Input
Type: String

A pointer to a string variable containing the key label of the record to be read from AES key-storage.

key_token_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *key_token* variable. The maximum length is 64.

key_token

Direction: Output
Type: String

A pointer to a string variable containing the key token read from AES key-storage. This variable must be large enough to hold the AES key token being read. On completion, the *key_token_length* variable contains the actual length of the token being returned.

Restrictions

The restrictions for CSNBAKRR.

None.

Required commands

The required commands for CSNBAKRR.

In order to access key storage, this verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBAKRR.

None.

Related information

Additional information about CSNBAKRR.

See "Key storage with Linux on z Systems, in contrast to z/OS on z Systems" on page 403.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBAKRRJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBAKRRJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  hikmNativeInteger rule_array_count,
  byte[]            rule_array,
  byte[]            key_label,
  hikmNativeInteger key_token_length,
  byte[]            key_token);
```

AES Key Record Write (CSNBAKRW)

Use this verb to write a copy of an AES key-token from application storage into AES key-storage.

This verb can perform the following processing options:

- Write the new key-token only if the old token was null.
- Write the new key-token regardless of content of the old token.

AES key records are stored in the external key-storage file defined by the CSUAESDS environment variable.

Note: Before using this verb, use the verb "AES Key Record Create (CSNBAKRC)" on page 407 to create a key record in the key storage file.

Format

The format of CSNBAKRW.

```
CSNBAKRW (
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            key_label,
            key_token_length,
            key_token)
```

Parameters

The parameters for CSNBAKRW.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0 or 1.

rule_array

Direction: Input
Type: String array

A pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters.

The *rule_array* keywords are described in Table 116.

Table 116. Keywords for AES Key Record Write control information

Keyword	Description
Processing option (One, optional)	
CHECK	Specifies that the record is written only if a record of the same label in AES key-storage contains a null key-token. This is the default.
OVERLAY	Specifies that the record is overwritten regardless of the current content of the record in AES key-storage.

key_label

Direction: Input
Type: String

A pointer to a string variable containing the key label that identifies the record in AES key-storage where the key token is to be written.

key_token_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *key_token* variable. This value must be 64.

key_token

Direction: Input
Type: String

A pointer to a string variable containing the AES key-token to be written into AES key-storage.

Restrictions

The restrictions for CSNBAKRW.

The record defined by the *key_label* parameter must be unique and must already exist in the key storage file.

Required commands

The required commands for CSNBAKRW.

In order to access key storage, this verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBAKRW.

None.

Related information

Additional information about CSNBAKRW.

You can use this verb with the key record create verb to write an initial record to key storage. Use it following the Key Import and Key Generate verb to write an operational key imported or generated by these verbs directly to the key storage file.

See "Key storage with Linux on z Systems, in contrast to z/OS on z Systems" on page 403.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBAKRWJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBAKRWJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  hikmNativeInteger rule_array_count,
  byte[]            rule_array,
  byte[]            key_label,
  hikmNativeInteger key_token_length,
  byte[]            key_token);
```

DES Key Record Create (CSNBKRC)

Use the DES Key Record Create verb to add a key record to the DES key storage file.

The record contains a key token set to binary zeros and is identified by the label passed in the *key_label* parameter. The key label must be unique.

DES key records are stored in the external key-storage file defined by the CSUDESDS environment variable.

Format

The format of CSNBKRC.

```
CSNBKRC(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             key_label)
```

Parameters

The parameters for CSNBKRC.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

key_label

Direction: Input
Type: String

The 64-byte label of a record in the DES key storage file that is the target of this verb. The created record contains a key token set to binary zeros and has a key type of **NULL**.

Restrictions

The restrictions for CSNBKRC.

The record must have a unique label. Therefore, there cannot be another record in the DES key storage file with the same label and a different key type.

Required commands

The required commands for CSNBKRC.

In order to access key storage, this verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBKRC.

None.

Related information

Additional information for CSNBKRC.

See "Key storage with Linux on z Systems, in contrast to z/OS on z Systems" on page 403.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKRCJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBKRCJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  byte[]            key_label);
```

DES Key Record Delete (CSNBKRD)

Use the DES Key Record Delete verb to perform one of the following tasks in the DES key storage file.

- Replace the token in a key record with a null key token
- Delete an entire key record, including the key label, from the key storage file

DES key records are stored in the external key-storage file defined by the CSUDESDS environment variable.

Format

The format of CSNBKRD.

```
CSNBKRD(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_label)
```

Parameters

The parameters for CSNBKRD.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1.

rule_array

Direction: Input
Type: String array

The 8-byte keyword that defines the action to be performed. The *rule_array* keywords are described in Table 117.

Table 117. Keywords for DES Key Record Delete control information

Keyword	Description
Task (One required)	
TOKEN-DL	Deletes a key token from a key record in DES key storage.

Table 117. Keywords for DES Key Record Delete control information (continued)

Keyword	Description
LABEL-DL	Deletes an entire key record, including the key label, from DES key storage.

key_label

Direction: Input
Type: String

The 64-byte label of a record in the key storage file that is the target of this verb.

Restrictions

The restrictions for CSNBKRD.

The record defined by the *key_label* must be unique.

Required commands

The required commands for CSNBKRD.

In order to access key storage, this verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBKRD.

None.

Related information

Additional information about CSNBKRD.

See "Key storage with Linux on z Systems, in contrast to z/OS on z Systems" on page 403.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKRDJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBKRDJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  hikmNativeInteger rule_array_count,
  byte[]            rule_array,
  byte[]            key_label);
```

DES Key Record List (CSNBKRL)

The DES Key Record List verb creates a key-record-list file containing information about specified key records in key storage.

Information listed includes whether record validation is correct, the type of key, and the date and time the record was created and last updated.

Specify the key records to be listed using the key-label variable. To identify multiple key records, use the wild card (*) in the key label.

Note: To list all the labels in key storage, specify the key_label parameter with *, *.*, *.*.*, and so forth, up to a maximum of seven name tokens (*.*.*.*.*.*.*).

This verb creates the key-record-list file and returns the name of the file and the length of the file name to the calling application. This file has a header record, followed by 0 - *n* detail records, where *n* is the number of key records with matching key-labels. The file is kept in the `/opt/IBM/CCA/keys/deslist` directory (assuming the directory name was not changed during installation). These list files are created under the ownership of the environment of the user that requests the list service. Make sure the files created kept the same group ID as your installation requires. This can also be achieved by setting the "set-group-id-on-execution" bit on in this directory. See the g+s flags in the **chmod** command for full details. Not doing this might cause errors to be returned on key-record-list verbs.

DES key records are stored in the external key-storage file defined by the CSUDESDS environment variable.

Format

The format of CSNBKRL.

```
CSNBKRL(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    key_label,
    dataset_name_length,
    dataset_name,
    security_server_name)
```

Parameters

The parameters for CSNBKRL.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

key_label

Direction: Input
Type: String

The *key_label* parameter is a pointer to a string variable containing the key label of a key-token record in key storage. In a key label, you can use a wild card (*) to identify multiple records in key storage.

dataset_name_length

Direction: Output
Type: Integer

The *dataset_name_length* parameter is a pointer to an integer variable containing the number of bytes of data returned by the verb in the *dataset_name* variable.

The maximum returned length is 64 bytes.

dataset_name

Direction: Output
Type: String

The *dataset_name* parameter is a pointer to a 64-byte string variable containing the name of the file returned by the verb. The file contains the key-record information.

The verb returns the file name as a fully qualified file specification.

Note: When the verb stores a key-record-list file, it overlays any older file with the same name.

security_server_name

Direction: Output
Type: String

The *security_server_name* parameter is a pointer to a string variable. The information in this variable is not currently used, but the variable must be declared.

Restrictions

The restrictions for CSNBKRL.

None.

Required commands

The required commands for CSNBKRL.

In order to access key storage, this verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBKRL.

None.

Related information

Additional information about CSNBKRL.

See “Key storage with Linux on z Systems, in contrast to z/OS on z Systems” on page 403.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKRLJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBKRLJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
```

```
byte[]            exit_data,
byte[]            key_label,
hikmNativeInteger data_set_name_length,
byte[]            data_set_name,
byte[]            security_server_name);
```

DES Key Record Read (CSNBKRR)

Use the DES Key Record Read verb to copy an internal key token from the DES key storage file to application storage.

Other cryptographic services can then use the copied key token directly. The key token can also be used as input to the token copying functions of Key Generate or Key Import verbs to create additional NOCV keys.

DES key records are stored in the external key-storage file defined by the CSUDESDS environment variable.

Format

The format of CSNBKRR.

```
CSNBKRR(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             key_label,
             key_token)
```

Parameters

The parameters for CSNBKRR.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

key_label

Direction: Input
Type: String

The 64-byte label of a record in the DES key storage file. The internal key token in this record is returned to the caller.

key_token

Direction: Output
Type: String

The 64-byte internal key token retrieved from the DES key storage file.

Restrictions

The restrictions for CSNBKRR.

The record defined by the *key_label* parameter must be unique and must already exist in the key storage file.

Required commands

The required commands for CSNBKRR.

In order to access key storage, this verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBKRR.

None.

Related information

Additional information about CSNBKRR.

See "Key storage with Linux on z Systems, in contrast to z/OS on z Systems" on page 403.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKRRJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBKRRJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  byte[]            key_label,
  byte[]            key_token);
```

DES Key Record Write (CSNBKRW)

Use the DES Key Record Write verb to copy an internal DES key token from application storage into the DES key storage file.

The key label must be unique and the record must already exist in the key storage file.

DES key records are stored in the external key-storage file defined by the CSUDESDS environment variable.

Note: Before you use this verb, use the DES Key Record Create verb (see "DES Key Record Create (CSNBKRC)" on page 418) to create a key record in the key storage file.

Format

The format of CSNBKRW.

```
CSNBKRW(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             key_token,
             key_label)
```

Parameters

The parameters for CSNBKRW.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

key_token

Direction: Input/Output
Type: String

The 64-byte internal key token that is written to the DES key storage file.

key_label

Direction: Input
Type: String

The 64-byte label of a record in the DES key storage file that is the target of this verb. The record is updated with the internal key token supplied in the *key_token* parameter.

Restrictions

The restrictions for CSNBKRW.

The record defined by the *key_label* parameter must be unique and must already exist in the key storage file.

Required commands

The required commands for CSNBKRW.

In order to access key storage, this verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBKRW.

None.

Related information

Additional information about CSNBKRW.

You can use this verb with the key record create verb to write an initial record to key storage. Use it following the Key Import and Key Generate verb to write an operational key imported or generated by these verbs directly to the key storage file.

See "Key storage with Linux on z Systems, in contrast to z/OS on z Systems" on page 403.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKRWJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBKRWJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           byte[]            key_token,
           byte[]            key_label);
```

PKA Key Record Create (CSNDKRC)

This verb writes a new record to the PKA key storage file.

PKA key records are stored in the external key-storage file defined by the CSUPKADS environment variable.

Format

The format of CSNDKRC.

```
CSNDKRC(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             label,
             token_length,
             token)
```

Parameters

The parameters for CSNDKRC.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0.

rule_array

Direction: Input
Type: String array

This parameter is ignored.

label

Direction: Input
Type: String

The label of the record to be created, 64-byte character string.

token_length

Direction: Input
Type: Integer

The length of the field containing the token to be written to the PKA key storage file. If zero is specified, a null token will be added to the file. The maximum value of *token_length* is the maximum length of a private RSA token.

token

Direction: Input
Type: String

Data to be written to the PKA key storage file if *token_length* is nonzero. An RSA private token in either external or internal format, or an RSA public token.

Restrictions

The restrictions for CSNDKRC.

None.

Required commands

The required commands for CSNDKRC.

In order to access key storage, this verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNDKRC.

None.

Related information

Additional information about CSNDKRC.

See “Key storage with Linux on z Systems, in contrast to z/OS on z Systems” on page 403.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDKRCJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNDKRCJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            key_label,
           hikmNativeInteger key_token_length,
           byte[]            key_token);
```

PKA Key Record Delete (CSNDKRD)

Use PKA Key Record Delete to delete a record from the PKA key storage file.

PKA key records are stored in the external key-storage file defined by the CSUPKADS environment variable.

Format

The format of CSNDKRD.

```
CSNDKRD(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_identifier)
```

Parameters

The parameters for CSNDKRD.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0 or 1.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. Each keyword is left-aligned in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage. The *rule_array* keywords are described in Table 118 on page 430.

Table 118. Keywords for PKA Key Record Delete control information

Keyword	Description
Deletion mode (One, optional). Specifies whether the record is to be deleted entirely or whether only its contents are to be erased.	
LABEL-DL	Specifies the record will be deleted from the PKA key storage file entirely. This is the default deletion mode.
TOKEN-DL	Specifies only the contents of the record are to be deleted. The record will still exist in the PKA key storage file, but will contain only binary zeros.

key_identifier

Direction: Input
Type: String

The label of the record to be deleted, a 64-byte character string.

Restrictions

The restrictions for CSNDKRD.

None.

Required commands

The required commands for CSNDKRD.

In order to access key storage, this verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNDKRD.

None.

Related information

Additional information about CSNDKRD.

See "Key storage with Linux on z Systems, in contrast to z/OS on z Systems" on page 403.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDKRDJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNDKRDJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            key_identifier);
```

PKA Key Record List (CSNDKRL)

The PKA Key Record List verb creates a key-record-list file containing information about specified key records in PKA key-storage.

Information includes whether record validation is correct, the type of key, and the dates and times when the record was created and last updated.

Specify the key records to be listed using the *key_label* parameter. To identify multiple key records, use the wild card (*) in a key label.

Note: To list all the labels in key storage, specify the *key_label* parameter with *, *.*, *.*.*, and so forth, up to a maximum of seven name tokens (*.*.*.*.*.*.*).

This verb creates the list file and returns the name of the file and the length of the file name to the calling application. This verb also returns the name of the security server where the file is stored. The PKA Key Record List file has a header record, followed by 0 - *n* detail records, where *n* is the number of key records with matching key labels. The file is kept in the **/opt/IBM/CCA/keys/pkalist** directory (assuming the directory name was not changed during installation). These list files are created under the ownership of the environment of the user that requests the list verb. Make sure the files created kept the same group ID as your installation requires. This can also be achieved by setting the "set-group-id-on-execution" bit on in this directory. See the g+s flags in the **chmod** command for full details. Not doing this might cause errors to be returned on key-record-list verbs.

PKA key records are stored in the external key-storage file defined by the CSUPKADS environment variable.

For information concerning the location of the key-record-list directory, refer to the *IBM 4764 PCI-X Cryptographic Coprocessor CCA Support Program Installation Manual*.

Format

The format of CSNDKRL.

```
CSNDKRL(
    return_code,
    reason_code,
    exit_data_length,
    edit_data,
    rule_array_count,
    rule_array,
    key_label,
    dataset_name_length,
    dataset_name,
    security_server_name)
```

Parameters

The parameters for CSNDKRL.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input

Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0.

rule_array

Direction: Input
Type: String array

This parameter is ignored.

key_label

Direction: Output
Type: String

The *key_label* parameter is a pointer to a string variable containing a key record in PKA key-storage. You can use a wild card (*) to identify multiple records in key storage.

dataset_name_length

Direction: Input
Type: Integer

The *dataset_name_length* parameter is a pointer to an integer variable containing the number of bytes of data returned in the *dataset_name* variable. The maximum returned length is 64 bytes.

dataset_name

Direction: Output
Type: String

The *dataset_name* parameter is a pointer to a 64-byte string variable containing the name of the file returned by the verb. The file contains the key-record information.

The verb returns the file name as a fully qualified file specification.

Note: When the verb stores a key-record-list file, it overlays any older file with the same name.

security_server_name

Direction: Output
Type: String

The *security_server_name* parameter is a pointer to a string variable. The information in this variable is not currently used, but the variable must be declared.

Restrictions

The restrictions for CSNDKRL.

None.

Required commands

The required commands for CSNDKRL.

In order to access key storage, this verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNDKRL.

None.

Related information

Additional information about CSNDKRL.

See "Key storage with Linux on z Systems, in contrast to z/OS on z Systems" on page 403.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDKRLJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNDKRLJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            key_label,
           hikmNativeInteger data_set_name_length,
           byte[]            data_set_name,
           byte[]            security_server_name);
```

PKA Key Record Read (CSNDKRR)

Reads a record from the PKA key storage file and returns the content of the record. This is true even when the record contains a null PKA token.

PKA key records are stored in the external key-storage file defined by the CSUPKADS environment variable.

Format

The format of CSNDKRR.

```
CSNDKRR(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             label,
             token_length,
             token)
```

Parameters

The parameters for CSNDKRR.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0.

rule_array

Direction: Input
Type: String array

This parameter is ignored.

label

Direction: Input
Type: String

The label of the record to be read, a 64-byte character string.

token_length

Direction: Input/Output
Type: Integer

The length of the area to which the record is to be returned. On successful completion of this verb, *token_length* will contain the actual length of the record returned.

token

Direction: Output
Type: String

This is the area into which the returned record will be written. The area should be at least as long as the record.

Restrictions

The restrictions for CSNDKRR.

None.

Required commands

The required commands for CSNDKRR.

In order to access key storage, this verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNDKRR.

None.

Related information

Additional information about CSNDKRR.

See "Key storage with Linux on z Systems, in contrast to z/OS on z Systems" on page 403.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDKRRJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNDKRRJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            key_label,
           hikmNativeInteger key_token_length,
           byte[]            key_token);
```

PKA Key Record Write (CSNDKRW)

Writes over an existing record in the PKA key storage file.

PKA key records are stored in the external key-storage file defined by the CSUPKADS environment variable.

Format

The format of CSNDKRW.

```
CSNDKRW(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             label,
             token_length,
             token)
```

Parameters

The parameters for CSNDKRW.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0 or 1.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. Each keyword is left-aligned in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage. The *rule_array* keywords are described in Table 119.

Table 119. Keywords for PKA Key Record Write control information

Keyword	Description
Write mode (One, optional). Specifies the circumstances under which the record is to be written.	
CHECK	Specifies the record will be written only if a record of type **NULL** with the same label exists in the PKA key storage file. If such a record exists, it is overwritten. This is the default condition.
OVERLAY	Specifies the record will be overwritten regardless of the current content of the record. If a record with the same label exists in the PKA key storage file, is overwritten.

label

Direction: Input
Type: String

The label of the record to be overwritten, a 64-byte character string.

token_length

Direction: Input
Type: Integer

The length of the field containing the token to be written to the PKA key storage file.

token

Direction: Input
Type: String

The data to be written to the PKA key storage file, which is an RSA private token in either external or internal format, or an RSA public token.

Restrictions

The restrictions for CSNDKRW.

None.

Required commands

The required commands for CSNDKRW.

In order to access key storage, this verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNDKRW.

None.

Related information

Additional information about CSNDKRW.

See "Key storage with Linux on z Systems, in contrast to z/OS on z Systems" on page 403.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDKRWJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNDKRWJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            key_label,
           hikmNativeInteger key_token_length,
           byte[]            key_token);
```

Retained Key Delete (CSNDRKD)

Use this verb to delete a PKA key-record currently retained within the cryptographic engine.

Both public and private keys can be retained within the cryptographic engine using verbs such as PKA Key Generate. A list of retained keys can be obtained using the Retained Key List verb.

Important: Before using this verb, see the information about retained keys in "Using retained keys" on page 438.

Format

The format of CSNDRKD.

```
CSNDRKD(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_label)
```

Parameters

The parameters for CSNDRKD.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input

Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0.

rule_array

Direction: Input
Type: String array

This parameter is ignored.

key_label

Direction: Input
Type: String

A pointer to a string variable containing the key label of a PKA key-record that has been retained within the cryptographic engine. The use of a wild card in the *key_label* variable is not permitted.

Using retained keys

Retained key use is discouraged on the IBM z Systems platform because a retained key can exist only in one CEX*C Cryptographic adapter, by definition.

- This has potential problems:
  - The key cannot be exported, so it cannot be backed up.
  - The key cannot be exported to another card in the same group, so operations concerning the retained key cannot participate in load-balancing.
  - There is an exception to the above points, in that keys generated in a deterministic fashion using externally saved regeneration data (it is possible to save so-called 'regen data' securely) can be recreated from that data or created in multiple cards across a card group.

    However, this is a very sophisticated topic, and is beyond the scope of this document. Also, the complexity required to implement this properly, as well as the sophistication involved in its data management, present formidable obstacles.

Retained key support is offered in this release, however. The following verbs work with retained keys:

- "PKA Key Generate (CSNDPKG)" on page 635 generates an RSA retained key. The same restrictions that Integrated Cryptographic Service Facility (ICSF) has for retained key creation are implemented here. These are:
  - Notice that PKA Key Token Build will let you create 'key-mgmt' skeleton key tokens, and this is as designed. You can still pass these to PKA Key Generate and have a key pair created. What is not allowed is specifying that this 'key-mgmt' token is to be generated in PKA Key Generate as a RETAIN key token: a retained key. Such an attempt will fail with error 12 reason code 3046.
  - The maximum modulus size is 2048 bits.
  - The usage flags are restricted to signature generation.

    Specifically, key management usage for retained keys is not allowed because of the dangers of losing your key encrypting key (kek) for important keys, when that kek exists only inside a single adapter.
- "Retained Key List (CSNDRKL)" on page 439 lists the retained keys inside an adapter.

- "Retained Key Delete (CSNDRKD)" on page 437 deletes a retained key from adapter internal storage.

Restrictions

The restrictions for CSNDRKD.

None.

Required commands

The required commands for CSNDRKD.

This verb requires the **Retained Key Delete** command (offset X'0203') to be enabled in the active role.

Usage notes

The usage notes for CSNDRKD.

This verb is not impacted by the AUTOSELECT option. See "Verbs that ignore AUTOSELECT" on page 10 for more information.

Related information

Additional information about CSNDRKD.

See "Key storage with Linux on z Systems, in contrast to z/OS on z Systems" on page 403.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDRKDJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNDRKDJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            key_label);
```

Retained Key List (CSNDRKL)

Use this verb to list the key labels of selected PKA key records that have been retained within the cryptographic engine.

Specify the keys to be listed using the *key_label_mask* variable. To identify multiple keys, use a wild card (*) in the mask. Only labels with matching characters to those in the mask up to the first "*" is returned. To list all retained key labels, specify a mask of an *, followed by 63 space characters. For example, if the cryptographic engine has retained key labels a.a, a.a1, a.b.c.d, and z.a, and you specify the mask a.*, the verb returns a.a, a.a1 and a.b.c.d. If you specify a mask of a.a*, the verb returns a.a and a.a1.

To retain PKA keys within the coprocessor, use the PKA Key Generate verb. To delete retained keys from the coprocessor, use the Retained Key Delete verb.

Important: Before using this verb, see the information about retained keys in "Using retained keys" on page 438.

Format

The format of CSNDRKL.

```
CSNDRKL(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_label_mask,
             retained_keys_count,
             key_labels_count,
             key_labels)
```

Parameters

The parameters for CSNDRKL.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0.

rule_array

Direction: Input
Type: String array

This parameter is ignored.

key_label_mask

Direction: Input
Type: String

A pointer to a string variable containing a key-label mask that is used to filter the list of key names returned by the verb. Use a wild card (*) to identify multiple key records retained within the coprocessor.

retained_keys_count

Direction: Input/Output
Type: Integer

A pointer to an integer variable to receive the total number of retained-key records stored within the coprocessor.

key_labels_count

Direction: Input/Output
Type: Integer

A pointer to an integer variable which on input defines the maximum number of key labels to be returned, and which on output defines the number of key labels returned by the coprocessor.

key_labels

Direction: Output
Type: String array

A pointer to a string array variable containing the returned key labels. The coprocessor returns zero or more 64-byte array elements, each of which contains the key label of a PKA key-record retained within the coprocessor.

Restrictions

The restrictions for CSNDRKL.

None.

Required commands

The required commands for CSNDRKL.

This verb requires the **Retained Key List** command (offset X'0230') to be enabled in the active role.

Usage notes

The usage notes for CSNDRKL.

This verb is not impacted by the AUTOSELECT option. See "Verbs that ignore AUTOSELECT" on page 10 for more information.

Related information

Related information about CSNDRKL.

See "Key storage with Linux on z Systems, in contrast to z/OS on z Systems" on page 403.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDRKLJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNDRKLJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            key_label_mask,
           hikmNativeInteger retained_keys_count,
           hikmNativeInteger key_labels_count,
           byte[]            key_labels);
```

Chapter 10. Financial services

The process of validating personal identities in a financial transaction system is called *personal authentication*.

The personal identification number (PIN) is the basis for verifying the identity of a customer across financial industry networks. CCA provides verbs to translate, verify, and generate PINs. You can use the verbs to prevent unauthorized disclosures when organizations handle PINs.

The following verbs are described in this topic:

How personal identification numbers (PINs) are used

CCA allows your applications to generate PINs, to verify supplied PINs, and to translate PINs from one format or encryption key to another.

Many people are familiar with PINs, which are used to access an automated teller machine (ATM). From the system point of view, PINs are used primarily in financial networks to authenticate users. Typically, a user is assigned a PIN and enters the PIN at automated teller machines (ATMs) to gain access to his or her accounts. It is extremely important that the PIN be kept private so no one other than the account owner can use it.

How Visa card verification values are used

The Visa International Service Association (VISA) and MasterCard International, Incorporated have specified a cryptographic method to calculate a value that relates to the personal account number (PAN), the card expiration date, and the service code.

The Visa card-verification value (CVV) and the MasterCard card-verification code (CVC) can be encoded on either track 1 or track 2 of a magnetic striped card and are used to detect forged cards. Because most online transactions use track-2, the CCA verbs generate and verify the CVV[1] by the track-2 method.

The Visa CVV Generate verb calculates a 1-byte to 5-byte value through the DES-encryption of the PAN, the card expiration date, and the service code using two data-encrypting keys or two MAC keys. The Visa CVV Verify verb calculates the CVV by the same method, compares it to the CVV supplied by the application (which reads the credit card's magnetic stripe) in the *CVV_value*, and issues a return code that indicates whether the card is authentic.

The CVV Key Combine verb combines two operational DES keys into one operational TDES key. The verb accepts as input two single-length keys that are suitable for use with the CVV (card-verification value) algorithm. The resulting double-length key meets a more recent industry standard of using TDES to support PIN-based transactions. In addition, the double-length key is in a format that can be wrapped using the Key Export to TR31 verb.

Translating data and PINs in networks

More and more data is being transmitted across networks where, for various reasons, the keys used on one network cannot be used on another network.

Encrypted data and PINs that are transmitted across these boundaries must be translated securely from encryption under one key to encryption under another key. For example, a traveler visiting a foreign city might want to use an ATM to access an account at home. The PIN entered at the ATM might need to be encrypted at the ATM and sent over one or more financial networks to the traveler's home bank. At the home bank, the PIN must be verified before access is allowed. On intermediate systems (between networks), applications can use the Encrypted PIN Translate verb to re-encrypt a PIN block from one key to another. Running on CCA, such applications can ensure that PINs never appear in the clear and that the PIN-encrypting keys are isolated on their own networks.

Working with Europay-Mastercard-Visa Smart cards

There are several verbs you can use in secure communications with Europay-Mastercard-Visa (EMV) smart cards.

The processing capabilities are consistent with the specifications provided in these documents:

- *EMV 2000 Integrated Circuit Card Specification for Payment Systems Version 4.0 (EMV4.0) Book 2*
- *Design Visa Integrated Circuit Card Specification Manual*

1. The Visa CVV and the MasterCard CVC refer to the same value. CVV is used here to mean both CVV and CVC.

- *Integrated Circuit Card Specification (VIS) 1.4.0 Corrections*

EMV smart cards include the following processing capabilities:

- The Diversified Key Generate verb with rule-array options **TDES-XOR**, **TDESEMV2**, and **TDESEMV4** enables you to derive a key used to cipher and authenticate messages, and more particularly message parts, for exchange with an EMV smart card. You use the derived key with verbs such as: Encipher, Decipher, MAC Generate, MAC Verify, Secure Messaging for Keys, and Secure Messaging for PINs. These message parts can be combined with message parts created using the Secure Messaging for Keys and Secure Messaging for PINs verbs.
- The Secure Messaging for Keys verb enables secure incorporation of a key into a message part (generally the value portion of a TLV component of a secure message for a card). Similarly, the Secure Messaging for PINs verb enables secure incorporation of a PIN block into a message part.
- PIN Change/Unblock verb enables encryption of a new PIN to send to a new EMV card, or to update the PIN value on an initialized EMV card. This verb generates both the required session key (from the master encryption key) and the required authentication code (from the master authentication key).
- The **ZERO-PAD** option of the PKA Encrypt enables validation of a digital signature created according to ISO 9796-2 standard by encrypting information that you format, including a hash value of the message to be validated. You compare the resulting enciphered data to the digital signature accompanying the message to be validated.
- The MAC Generate and MAC Verify verbs post-pad a X'80'...X'00' string to a message as required for authenticating messages exchanged with EMV smart cards.

PIN verbs

CCA supports PIN verbs, various PIN algorithms, and PIN block formats.

It also explains the use of PIN-encrypting keys.

You use the PIN verbs to generate, verify, and translate PINs.

Generating a PIN

To generate personal identification numbers, call the Clear PIN Generate or Encrypted PIN Generate verb.

Using a PIN generation algorithm, data used in the algorithm, and the PIN generation key, the Clear PIN Generate verb generates a clear PIN and a PIN verification value, or offset. Using a PIN generation algorithm, data used in the algorithm, the PIN generation key, and an outbound PIN encrypting key, the Encrypted PIN Generate verb generates and formats a PIN and encrypts the PIN block.

Encrypting a PIN

To format a PIN into a supported PIN block format and encrypt the PIN block, call the Clear PIN Encrypt verb.

Generating a PIN validation value from an encrypted PIN block

To generate a clear VISA PIN validation value (PVV) from an encrypted PIN block, call the Clear PIN Generate Alternate verb.

The PIN block can be encrypted under an input PIN-encrypting key (**IPINENC**) or an output PIN encrypting key (**OPINENC**).

Verifying a PIN

To verify a supplied PIN, call the Encrypted PIN Verify verb.

You supply the enciphered PIN, the PIN-encrypting key that enciphers the PIN, and other data. You must also specify the PIN verification key and PIN verification algorithm. The Encrypted PIN Verify verb generates a verification PIN. This verb compares the two personal identification numbers and if they are the same, it verifies the supplied PIN.

Translating a PIN

To translate a PIN block format from one PIN-encrypting key to another or from one PIN block format to another, call the Encrypted PIN Translate verb.

You must identify the input PIN-encrypting key that originally enciphered the PIN. You also need to specify the output PIN-encrypting key that you want the verb to use to encipher the PIN. If you want to change the PIN block format, specify a different output PIN block format from the input PIN block format.

Algorithms for generating and verifying a PIN

CCA supports the following algorithms for generating and verifying personal identification numbers.

- IBM 3624 institution-assigned PIN
- IBM 3624 customer-selected PIN (through a PIN offset)
- IBM German Bank Pool PIN (verify through an institution key)
- VISA PIN through a VISA PIN validation value
- Interbank PIN

The algorithms are described in detail in Chapter 20, “PIN formats and algorithms,” on page 913.

Using PINs on different systems

CCA allows you to translate different PIN block formats, which lets you use personal identification numbers on different systems.

CCA supports the following formats:

- IBM 3624
- IBM 3621 (same as IBM 5906)
- IBM 4704 encrypting PINPAD format
- ISO 0 (same as ANSI 9.8, VISA 1, and ECI 1)
- ISO 1 (same as ECI 4)
- ISO 2
- VISA 2
- VISA 3

- VISA 4
- ECI 2
- ECI 3

The algorithms are described in detail in Chapter 20, “PIN formats and algorithms,” on page 913.

PIN-Encrypting keys

A unique master key variant enciphers each type of key.

Note that the PIN block variant constant (PBVC) are not supported in this version of CCA.

Derived unique key per transaction algorithms

CCA supports ANSI X9.24 derived unique key per transaction algorithms to generate PIN-encrypting keys from user data.

CCA supports both single-length and double-length key generation. Keywords for single-length and double-length key generation cannot be mixed.

Encrypted PIN Translate

The **UKPTIPIN**, **IPKTOPIN**, and **UKPTBOTH** keywords will cause the verb to generate single-length keys. and **DUKPT-IP**, **DKPT-OP**, and **DUKPT-BH** are the respective keywords to generate double-length keys.

The *input_PIN_profile* and *output_PIN_profile* parameters must supply the current key serial number when these keywords are specified.

Encrypted PIN Verify

The **UKPTIPIN** keyword will cause the verb to verify single-length keys. **DUKPT-IP** is the keyword for double-length key generation.

The *input_PIN_profile* parameter must supply the current key serial number when these keywords are specified.

ANSI X9.8 PIN restrictions

These access control points implement the PIN-block processing restrictions of the ANSI X9.8 standard implemented in CCA 4.1.0.

These access control points are available on the IBM z196 starting with the CEX3C feature. These access control points are disabled in the default role. A TKE Workstation is required to enable them.

These are the access control points:

- **ANSI X9.8 PIN - Enforce PIN block restrictions** (offset X'0350'). See “ANSI X9.8 PIN - Enforce PIN block restrictions” on page 448.
- **ANSI X9.8 PIN - Allow modification of PAN** (offset X'0351') See “ANSI X9.8 PIN - Allow modification of PAN” on page 448.
- **ANSI X9.8 PIN - Allow only ANSI PIN blocks** (offset X'0352') See “ANSI X9.8 PIN - Allow only ANSI PIN blocks” on page 449.

- **ANSI X9.8 PIN - Use stored decimalization tables only** (offset X'0356') See "Use stored decimalization tables only" on page 449.

These verbs are affected by these access control points:

- Clear PIN Generate Alternate (CSNBCPA)
- Encrypted PIN Generate (CSNBEPG)
- Encrypted PIN Translate (CSNBPTR)
- Encrypted PIN Verify (CSNBPVR)
- Secure Messaging for PINs (CSNBSPN)

PIN decimalization tables can be stored in the Coprocessor, starting with CEX3C, for use by CCA verbs. Only tables that have been activated can be used. A TKE Workstation is required to manage the tables in the coprocessors.

ANSI X9.8 PIN - Enforce PIN block restrictions

When the ANSI X9.8 PIN - Enforce PIN block restrictions access control point is enabled, restrictions are enforced.

The following restrictions are enforced:

- The Encrypted PIN Translate and Secure Messaging for PINs verbs will not accept IBM 3624 PIN format in the output profile parameter when the input profile parameter is not IBM 3624.
- The Encrypted PIN Translate verb will not accept ISO-0 or ISO-3 formats in the input PIN profile unless ISO-0 or ISO-3 is in the output PIN profile.
- The Encrypted PIN Translate and Secure Messaging for PINs verbs will not accept ISO-1 or ISO-2 formats in the output profile parameter when the input profile parameter contains ISO-0, ISO-3, or VISA4.
- When the input profile parameter for the Encrypted PIN Translate and Secure Messaging for PINs verbs contains either ISO-0 or ISO-3 formats, the PAN within the decrypted PIN block will be extracted. This PAN must be the same as the PAN that was supplied as the input PAN parameter, and this PAN must be the same as the PAN supplied as the output PAN parameter.
- The input PAN and output PAN parameters for the Encrypted PIN Translate and Secure Messaging for PINs verbs must be equivalent.
- When the rule array for the Clear PIN Generate Alternate verb contains VISA-PVV, the input PIN profile must contain ISO-0 or ISO-3 formats.

ANSI X9.8 PIN - Allow modification of PAN

In order to enable the ANSI X9.8 PIN - Allow modification of PAN access control point, the ANSI X9.8 PIN - Enforce PIN block restrictions must also be enabled.

The ANSI X9.8 PIN - Allow modification of PAN access control point cannot be enabled by itself.

When the ANSI X9.8 PIN - Allow modification of PAN access control point is enabled, the input PAN and output PAN parameters will be tested in the Encrypted PIN Translate and Secure Messaging for PINs verbs. The input PAN will be compared to the portions of the PAN that are recoverable from the decrypted PIN block. If the PANs are the same, the account number will be changed in the output PIN block.

ANSI X9.8 PIN - Allow only ANSI PIN blocks

In order to enable the ANSI X9.8 PIN - Allow only ANSI PIN blocks access control point, the ANSI X9.8 PIN - Enforce PIN block restrictions must also be enabled.

The **ANSI X9.8 PIN - Allow only ANSI PIN blocks** access control point cannot be enabled by itself.

When this access control point is enabled, the Encrypted PIN Translate verb allows reformatting of the PIN block as shown in Table 120.

Table 120. ANSI X9.8 PIN - Allow only ANSI PIN blocks

Reformat to: / **Reformat from:**	**ISO Format 0**	**ISO Format 1**	**ISO Format 3**
ISO Format 0	Reformat permitted. Change of PAN not permitted	Not permitted	Reformat permitted. Change of PAN not permitted.
ISO Format 1	Reformat permitted	Reformat permitted	Reformat permitted
ISO Format 3	Reformat permitted. Change of PAN not permitted.	Not permitted	Reformat permitted. Change of PAN not permitted.

Use stored decimalization tables only

The **ANSI X9.8 PIN - Use stored decimalization tables only** access control point can be enabled by itself.

When this access control point is enabled, the Secure Messaging for PINs, Clear PIN Generate Alternate, Encrypted PIN Generate, and Encrypted PIN Verify verbs must supply a decimalization table that matches the active decimalization tables stored in the coprocessors. The decimalization table in the *data_array* parameter will be compared against the active decimalization tables in the coprocessor, and if the supplied table matches a stored table, the request will be processed. If the supplied table doesn't match any of the stored tables or there are no stored tables, the request fails.

PIN decimalization tables can be stored in the in the Coprocessor, starting with CEX3C, for use by CCA verbs. Only tables that have been activated can be used. A TKE Workstation is required to manage the tables in the coprocessors.

The PIN profile

The PIN profile components include a block format, format control, pad digit, and key serial number.

The PIN profile consists of the following:

- PIN block format (see "PIN block format" on page 450)
- Format control (see "Format control" on page 452)
- Pad digit (see "Pad digit" on page 452)
- Current Key Serial Number (for **UKPT** and **DUKPT** – see "Current key serial number" on page 453)

Table 121 on page 450 shows the format of a PIN profile.

Table 121. Format of a PIN profile

Bytes	Description
0 - 7	PIN block format
8 - 15	Format control
16 - 23	Pad digit
24 - 47	Current Key Serial Number (for UKPT and DUKPT)

PIN block format

This keyword specifies the format of the PIN block. The 8-byte value must be left-aligned and padded with blanks.

Refer to Table 122 for a list of valid values.

Table 122. Format values of PIN blocks

Format value	Description
ECI-2	Eurocheque International format 2
ECI-3	Eurocheque International format 3
ISO-0	ISO format 0, ANSI X9.8, VISA 1, and ECI 1
ISO-1	ISO format 1 and ECI 4
ISO-2	ISO format 2
ISO-3	ISO format 3
VISA-2	VISA format 2
VISA-3	VISA format 3
VISA-4	VISA format 4
3621	IBM 3621 and 5906
3624	IBM 3624
4704-EPP	IBM 4704 encrypting PIN pad

PIN block format and PIN extraction method keywords

In the Clear PIN Generate Alternate, Encrypted PIN Translate, and Encrypted PIN Verify verbs, you can specify a PIN extraction keyword for a given PIN block format.

In the table below, the allowable PIN extraction methods are listed for each PIN block format. The first PIN extraction method keyword listed for a PIN block format is the default.

Table 123. PIN block format and PIN extraction method keywords

PIN block format	PIN extraction method keywords	Description
ECI-2	**PINLEN04**	The PIN extraction method keywords specify a PIN extraction method for a PINLEN04 format.
ECI-3	**PINBLOCK**	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
ISO-0	**PINBLOCK**	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.

Table 123. PIN block format and PIN extraction method keywords (continued)

PIN block format	PIN extraction method keywords	Description
ISO-1	**PINBLOCK**	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
ISO-2	**PINBLOCK**	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
ISO-3	**PINBLOCK**	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
VISA-2	**PINBLOCK**	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
VISA-3	**PINBLOCK**	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
VISA-4	**PINBLOCK**	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
3621	**PADDIGIT**, **HEXDIGIT**, **PINLEN04** - **PINLEN12**, **PADEXIST**	The PIN extraction method keywords specify a PIN extraction method for an IBM 3621 PIN block format. The first keyword, **PADDIGIT**, is the default PIN extraction method for the PIN block format.
3624	**PADDIGIT**, **HEXDIGIT**, **PINLEN04** - **PINLEN16**, **PADEXIST**	The PIN extraction method keywords specify a PIN extraction method for an IBM 3624 PIN block format. The first keyword, **PADDIGIT**, is the default PIN extraction method for the PIN block format.
4704-EPP	**PINBLOCK**	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.

The PIN extraction methods operate as follows:

PINBLOCK
Specifies that the service verb use one of these:
- The PIN length, if the PIN block contains a PIN length field
- The PIN delimiter character, if the PIN block contains a PIN delimiter character.

PADDIGIT
Specifies that the verb use the pad value in the PIN profile to identify the end of the PIN.

HEXDIGIT
Specifies that the verb use the first occurrence of a digit in the range from X'A' to X'F' as the pad value to determine the PIN length

PINLEN*nn*
Specifies that the verb use the length specified in the keyword, where *nn* can range from 04 - 16, the number of digits used to identify the PIN.

PADEXIST
Specifies that the verb use the character in the 16th position of the PIN block as the value of the pad value.

Enhanced PIN security mode

An enhanced PIN security mode is available. This optional mode is selected by enabling the **Enhanced PIN Security** (offset X'0313') access control point in the CEX*C default role.

When active, this control point affects all PIN verbs that extract or format a PIN using a PIN-block format of 3621or 3624 with a PIN-extraction method of **PADDIGIT** .

Table 124 summarizes the verbs affected by the enhanced PIN security mode, and describes the effect that the mode has when the access control point is enabled.

Table 124. Verbs affected by enhanced PIN security mode

PIN-block format and PIN-extraction method	Affected verbs	PIN processing changes when Enhanced PIN Security Mode enabled
ECI-2, 3621, or 3624 formats AND **PINLEN***nn*	Clear PIN Generate Alternate Encrypted PIN Translate Encrypted PIN Verify	The **PINLEN***nn* keyword in the *rule_array* parameter for PIN extraction method is not allowed if the Enhanced PIN Security Mode is enabled. **Note:** The verb will fail with return code 8 and reason code X'7E0'.
3621 or 3624 format and **PADDIGIT**	Clear PIN Generate Alternate Encrypted PIN Translate Encrypted PIN Verify PIN Change/Unblock	PIN extraction determines the PIN length by scanning from right to left until a digit, not equal to the PAD digit, is found. The minimum PIN length is set at four digits, so scanning ceases one digit past the position of the fourth PIN digit in the block.
3621 or 3624 format and **PADDIGIT**	Clear PIN Encrypt Encrypted PIN Generate Encrypted PIN Translate	PIN formatting does not examine the PIN, in the output PIN block, to see if it contains the PAD digit.
3621 or 3624 format and **PADDIGIT**	Encrypted PIN Translate	Restricted to non-decimal digit for PAD digit.

Format control

This keyword specifies whether there is any control on the user-supplied PIN format.

The 8-byte value must be left-aligned and padded with blanks. The only permitted value is NONE, which indicates no format control will be used.

Pad digit

Some PIN formats require the pad digit parameter.

If the PIN format does not need a pad digit, the verb ignores this parameter. Table 125 shows the format of a pad digit. The PIN profile pad digit must be specified in upper case.

Table 125. Format of a pad digit

Bytes	Description
16 - 22	Seven space characters
23	Character representation of a hexadecimal pad digit or a space if a pad digit is not needed. Characters must be one of the following: digits 0 - 9, letters A - F, or a blank.

Each PIN format supports only a pad digit in a certain range. Table 126 lists the valid pad digits for each PIN block format.

Table 126. Pad digits for PIN block formats

PIN Block Format	Output PIN Profile	Input PIN Profile
ECI-2	Pad digit is not used	Pad digit is not used
ECI-3	Pad digit is not used	Pad digit is not used
ISO-0	F	Pad digit is not used
ISO-1	Pad digit is not used	Pad digit is not used
ISO-2	Pad digit is not used	Pad digit is not used
ISO-3	Pad digit is not used	Pad digit is not used
VISA-2	0 - 9	Pad digit is not used
VISA-3	0 - F	Pad digit is not used
VISA-4	F	Pad digit is not used
3621	0 - F	0 - F
3624	0 - F	0 - F
4704-EPP	F	Pad digit is not used

The verb returns an error indicating that the PAD digit is not valid if all of these conditions are met:

- The **Enhanced PIN Security** (offset X'0313') access control point is enabled in the active role.
- The output PIN profile specifies 3621 or 3624 as the PIN-block format.
- The output PIN profile specifies a decimal digit (0 - 9) as the PAD digit.

Recommendations for the pad digit

IBM recommends you use a non-decimal pad digit in the range of A - F when processing IBM 3624 and IBM 3621 PIN blocks.

If you use a decimal pad digit, the creator of the PIN block must ensure that the calculated PIN does not contain the pad digit, or unpredictable results might occur.

For example, you can exclude a specific decimal digit from being in any calculated PIN by using the IBM 3624 calculation procedure and by specifying a decimalization table that does not contain the desired decimal pad digit.

Current key serial number

The current key serial number is the concatenation of the initial key serial number (a 59-bit value) and the encryption counter (a 21-bit value).

The concatenation is an 80-bit (10-byte) value. Table 127 on page 454 shows the format of the current key serial number.

When **UKPT** or **DUKPT** is specified, the PIN profile parameter is extended to a 48-byte field and must contain the current key serial number.

Table 127. Format of the Current Key Serial Number Field

Bytes	Description
24 - 47	Character representation of the current key serial number used to derive the initial PIN encrypting key. It is left-aligned and padded with 4 blanks.

Decimalization tables

Decimalization tables can be loaded in the coprocessors to restrict attacks using modified tables.

The management of the tables requires a TKE workstation.

These verbs make use of the stored decimalization tables:

- Clear PIN Generate (CSNBPGN)
- Clear PIN Generate Alternate (CSNBCPA)
- Encrypted PIN Generate (CSNBEPG)
- Encrypted PIN Verify (CSNBPVR)

The **ANSI X9.8 PIN - Use stored decimalization table only** (offset X'0356') access control point is used to restrict the use of the stored decimalization tables. When the access control point is enabled, the table supplied by the verb will be compared against the active tables stored in the coprocessor. If the supplied table does not match any of the active tables, the request will fail.

A TKE workstation (Version 7.1 or later) is required to manage the PIN decimalization tables. The tables must be loaded and then activated. Only active tables are checked when the access control point is enabled.

Note: CCA routes work to all active coprocessors based on workload. All coprocessors must have the same set of decimalization tables for the decimalization table access control point to be effective.

Format preserving encryption

Format preserving encryption (FPE) is a method of encryption where the resulting ciphertext has the same form as the input clear text. The form of the text can vary according to the usage and the application. The contained information provides background information which is helpful for using the FPE services provided by CCA.

One example for format preserving encryption is a 16 digit credit card number. After using FPE to encrypt a credit card number, the resulting ciphertext is another 16 digit number. In this example of the credit card number, the output ciphertext is limited to numeric digits only.

The FPE services require some knowledge of the input clear text character set in order to create the appropriate output cipher text. The CSNBFPEE, CSNBFPED, CSNBFPET, and CSNBPTRE callable services use the tables in the following subsections to determine valid character sets for the clear text input parameters.

Base-10 alphabet

The original data type of the source field may be of any type. This alphabet requires the following values to be used in the VFPE algorithm:

```
Number of characters in alphabet('n'): 10
```

Table 128. Base-10 alphabet.

VFPE alphabet number	Character	ISO 7811 modified 5-bit ASCII	ISO 7811 modified 7-bit ASCII	Normal data type encoding		
				4-bit binary coded decimal	7-bit ASCII	8-bit EBCDIC
0	0	10000	0010000	0000	0110000	11110000
1	1	00001	1010001	0001	0110001	11110001
2	2	00010	1010010	0010	0110010	11110010
3	3	10011	0010011	0011	0110011	11110011
4	4	00100	1010100	0100	0110100	11110100
5	5	10101	0010101	0101	0110101	11110101
6	6	10110	0010110	0110	0110110	11110110
7	7	00111	1010111	0111	0110111	11110111
8	8	01000	1011000	1000	0111000	11111000
9	9	11001	0011001	1001	0111001	11111001

Base-16 alphabet

Cards are encoded with the special ISO 7811 modified 5-bit ASCII encoding for track 2. This data type allows parity checking of the digits. Many systems require this encoding to be converted into standard data types for processing. Other data fields may use base-16 encoding and would use this same alphabet when performing VFPE. These data types support values in the ranges 0 - 9 and A - F.

VFPE requires translation of the characters of the VFPE alphabet number prior to encryption. Therefore, any of the data types shown in Table 190 are supported. Decryption may use the same or a different data type than the original encoding. This alphabet requires the following values to be used in the VFPE algorithm:

```
Number of characters in alphabet('n'): 16
```

Table 129. Base-16 alphabet.

VFPE alphabet number	ISO 7811 modified 5-bit ASCII encoding		Normal data type encoding			
	Character	Binary	Character	4-bit binary coded decimal	7-bit ASCII	8-bit EBCDIC
0	0	10000	0	0000	0110000	11110000
1	1	00001	1	0001	0110001	11110001
2	2	00010	2	0010	0110010	11110010
3	3	10011	3	0011	0110011	11110011
4	4	00100	4	0100	0110100	11110100
5	5	10101	5	0101	0110101	11110101
6	6	10110	6	0110	0110110	11110110
7	7	00111	7	0111	0110111	11110111

Table 129. Base-16 alphabet (continued).

VFPE alphabet number	ISO 7811 modified 5-bit ASCII encoding		Normal data type encoding			
	Character	Binary	Character	4-bit binary coded decimal	7-bit ASCII	8-bit EBCDIC
8	8	01000	8	1000	0111000	11111000
9	9	11001	9	1001	0111001	11111001
10	:	11010	A	1010	1000001	11000001
11	;	01011	B	1011	1000010	11000010
12	<	11100	C	1100	1000011	11000011
13	=	01101	D	1101	1000100	11000100
14	>	01110	E	1110	1000101	11000101
15	?	11111	F	1111	1000110	11000110

Track 1 alphabet

This alphabet requires the following values to be used in the VFPE algorithm:

```
Number of characters in alphabet('n'): 41
```

Table 130. Track 1 alphabet.

FPE alphabet number	Character	ISO 7811 modified 7-bit ASCII encoding	Standard data types 7-bit ASCII	Standard data types 8-bit ASCII
0	space	1000000	0100000	01000000
1	$	0000100	0100100	01011011
2	(	0001000	0101000	01001101
3	)	1001001	0101001	01011101
4	-	0001101	0101101	01100000
5	0	0010000	0110000	11110000
6	1	1010001	0110001	11110001
7	2	1010010	0110010	11110010
8	3	0010011	0110011	11110011
9	4	1010100	0110100	11110100
10	5	0010101	0110101	11110101
11	6	0010110	0110110	11110110
12	7	1010111	0110111	11110111
13	8	1011000	0111000	11111000
14	9	0011001	0111001	11111001
15	A	1100001	1000001	11000001
16	B	1100010	1000010	11000010
17	C	0100011	1000011	11000011
18	D	1100100	1000100	11000100
19	E	0100101	1000101	11000101

Table 130. Track 1 alphabet (continued).

FPE alphabet number	Character	ISO 7811 modified 7-bit ASCII encoding	Standard data types 7-bit ASCII	Standard data types 8-bit ASCII
20	F	0100110	1000110	11000110
21	G	1100111	1000111	11000111
22	H	1101000	1001000	11001000
23	I	0101001	1001001	11001001
24	J	0101010	1001010	11010001
25	K	1101011	1001011	11010010
26	L	0101100	1001100	11010011
27	M	1101101	1001101	11010100
28	N	1101110	1001110	11010101
29	O	0101111	1001111	11010110
30	P	1110000	1010000	11010111
31	Q	0110001	1010001	11011000
32	R	0110010	1010010	11011001
33	S	1110011	1010011	11100010
34	T	0110100	1010100	11100011
35	U	1110101	1010101	11100100
36	V	1110110	1010110	11100101
37	W	0110111	1010111	11100110
38	X	0111000	1011000	11100111
39	Y	1111001	1011001	11101000
40	Z	1111010	1011010	11101001

Usage notes for FPE Encipher (CSNBFPEE) and FPE Decipher (CSNBFPED) services

The CSNBFPEE and CSNBFPED services support two options:

1. the standard encryption or decryption option which uses the DES CBC mode of operation
2. the Visa Format Preserving Encryption (VFPE) option.

If the standard encryption or decryption option was selected, the plain text data was formatted into blocks and then encrypted or decrypted with triple-DES with a static TDES key or a DUKPT double length data encryption or decryption key. For the decryption operation, the data blocks must be decrypted and unblocked to produce the plain text. If the data was encrypted or decrypted with the VFPE option, it was processed in place without changing the data type or length of the field. Also, DUKPT key management is used.

These services can be used to encrypt or decrypt one or all of the following fields:

- the primary account number (PAN),
- the cardholder name,
- the track 1 discretionary data, or
- the track 2 discretionary data.

There are three encryption or decryption options:

1. the standard option with CBC mode TDES and DUKPT keys
2. the VFPE option with DUKPT keys
3. the standard option with CBC mode TDES and double-length TDES keys.

To use these services, you must specify the following:

- the processing method, which is limited to Visa Data Secure Platform (VDSP)
- the key management method, either STATIC or DUKPT
- the algorithm, which is limited to TDES
- the mode, either CBC or Visa Format Preserving Encryption (VFPE)
- the plain text to be encrypted or decrypted
- the character set of each field to be encrypted or decrypted using rule-array keywords
- the base derivation key and key serial number if DUKPT key management is used, or a double-length TDES key if STATIC key management is used
- a compliance or non-compliance indicator for the check digit of the PAN to be processed if VFPE is specified.

The services return the encrypted or decrypted fields and optionally, the DUKPT PIN key, if the DUKPT key management is selected and the PINKEY rule is specified.

Authentication Parameter Generate (CSNBAPG)

The Authentication Parameter Generate service generates an authentication parameter (AP) and returns it encrypted using the key supplied in the **AP_encrypting_key_identifier** parameter.

Format

The format of CSNBAPG.

```
CSNBAPG(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             inbound_PIN_encrypting_key_identifier_length,
             inbound_PIN_encrypting_key_identifier,
             encrypted_PIN_block,
             issuer_domestic_code,
             card_secure_code,
             PAN_data,
             AP_encrypting_key_identifier_length,
             AP_encrypting_key_identifier,
             AP_value)
```

Parameters

The parameters for CSNBAPG.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. This value must be 0, 1, or 2.

rule_array

Direction: Input
Type: String array

The keywords that provide control information to the verb. The **rule_array** keywords are described in Table 131.

Table 131. Keywords for Authentication Parameter Generate control information

Keyword	Description
AP Protection Method (One, optional)	
ENCRYPT	Specifies that the AP value should be returned encrypted under the **AP_encrypting_key_identifier** parameter. This is the default.
CLEAR	Specifies that the AP value should be returned in the clear.
AP Value Format (One, optional)	
BCD	Specifies the output format of the AP as packed binary coded decimal. This is the default.

inbound_PIN_encrypting_key_identifier_length

Direction: Input
Type: Integer

Length of the **inbound_PIN_encrypting_key_identifier** parameter in bytes. This value must be 64.

inbound_PIN_encrypting_key_identifier

Direction: Input
Type: String

An operational key token or the label of the AES or DES key storage record containing a double length IPINENC key that decrypts the PIN block.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

encrypted_PIN_block

Direction: Input
Type: String

The ISO-0 PIN block encrypted with the **inbound_PIN_encrypting_key_identifier**. The PIN within the PIN block must be a 5-digit value.

issuer_domestic_code

Direction: Input
Type: String

A 5 byte alphanumeric character string.

card_secure_code

Direction: Input
Type: String

An 8 byte string of digits grouped into two 4 byte sections. The 4 digits in a section cannot all be zero, for example, the value `0000` is invalid.

PAN_data

Direction: Input
Type: String

The personal account number (PAN). Must be 12 characters long.

AP_encrypting_key_identifier_length

Direction: Input
Type: Integer

The length of the **AP_encrypting_key_identifier** parameter in bytes. This value is 64 when a label is supplied. When the key identifier is a key token, the value is the length of the token. The maximum value is 725. The value may be 0 when the CLEAR **rule_array** keyword is specified.

AP_encrypting_key_identifier

Direction: Input
Type: String

An internal key token or the label of the AES or DES key storage record containing a double length DATA key used to encrypt the **AP_value**. If the AP Protection Method was specified as CLEAR keyword in the **rule_array** parameter, this parameter is ignored.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

AP_value

Direction: Output
Type: String

An 8 byte character string containing the generated authentication parameter.

Restrictions

The restrictions for CSNBAPG.

None.

Required commands

The required commands for CSNBAPG.

The Authentication Parameter Generate verb requires the **Authentication Parameter Generate** command (offset X'02B1') to be enabled in the active role.

In addition, rule-array keyword CLEAR requires the **Authentication Parameter Generate - Clear** command (offset X'02B2') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBAPG.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBAPGJ .

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBAPGJ(
           hikmNativeInteger  return_code,
           hikmNativeInteger  reason_code,
           hikmNativeInteger  exit_data_length,
           byte[]             exit_data,
           hikmNativeInteger  rule_array_count,
           byte[]             rule_array,
           hikmNativeInteger  inbound_PIN_encrypting_key_identifier_length,
           byte[]             inbound_PIN_encrypting_key_identifier,
           byte[]             encrypted_PIN_block,
           byte[]             issuer_domestic_code,
           byte[]             card_secure_code,
           byte[]             PAN_data,
           hikmNativeInteger  AP_encrypting_key_identifier_length,
           byte[]             AP_encrypting_key_identifier,
           byte[]             AP_value);
```

Clear PIN Encrypt (CSNBCPE)

The Clear PIN Encrypt verb formats a PIN into one of the following PIN block formats and encrypts the results.

You can use this verb to create an encrypted PIN block for transmission. With the **RANDOM** keyword, you can have the verb generate random PIN numbers.

Note: A clear PIN is a sensitive piece of information. Ensure your application program and system design provide adequate protection for any clear PIN value.

- IBM 3621 format
- IBM 3624 format
- ISO-0 format (same as the ANSI X9.8, VISA-1, and ECI formats)
- ISO-1 format (same as the ECI-4 format)
- ISO-2 format
- ISO-3 format
- IBM 4704 encrypting PINPAD (4704-EPP) format
- VISA 2 format
- VISA 3 format
- VISA 4 format
- ECI2 format
- ECI3 format

Format

The format of CSNBCPE.

```
CSNBCPE(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             PIN_encrypting_key_identifier,
             rule_array_count,
             rule_array,
             clear_PIN,
             PIN_profile,
             PAN_data,
             sequence_number
             encrypted_PIN_block)
```

Parameters

The parameter definitions for CSNBCPE.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

PIN_encrypting_key_identifier

Direction: Input/Output
Type: String

The 64-byte string containing an internal key token or a key label of an internal key token. The internal key token contains the key that encrypts the PIN block. The control vector in the internal key token must specify an **OPINENC** key type and have the CPINENC usage bit set to B'1'.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. Valid values are 0 and 1.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. The keyword is left-aligned in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage. The *rule_array* keywords are described in Table 132

Table 132. Keywords for Clear PIN Encrypt control information

Keyword	Description
Process Rule (Optional)	
ENCRYPT	This is the default. Use of this keyword is optional.
RANDOM	Causes the verb to generate a random PIN value. The length of the PIN is based on the value in the *clear_PIN* variable. Set the value of the clear PIN to zero and use as many digits as the desired random PIN. Pad the remainder of the clear PIN variable with space characters.

clear_PIN

Direction: Input
Type: String

A 16-character string with the clear PIN. The value in this variable must be left-aligned and padded on the right with space characters.

PIN_profile

Direction: Input
Type: String array

A 24-byte string containing three 8-byte elements with a PIN block format keyword, the format control keyword, NONE, and a pad digit as required by certain formats. See "The PIN profile" on page 449 for additional information.

PAN_data

Direction: Input
Type: String

A 12-byte PAN in character format. The verb uses this parameter if the PIN profile specifies the **ISO-0**, **ISO-3**, or **VISA-4** keyword for the PIN block format. Otherwise, ensure this parameter is a 12-byte variable in application storage. The information in this variable will be ignored, but the variable must be specified.

Note: When using the **ISO-0** or **ISO-3** keyword, use the 12 rightmost digits of the PAN data, excluding the check digit. When using the **VISA-4** keyword, use the 12 leftmost digits of the PAN data, excluding the check digit.

sequence_number

Direction: Input
Type: Integer

The 4-byte character integer. The verb currently ignores the value in this variable. For future compatibility, the suggested value is 99999.

encrypted_PIN_block

Direction: Output
Type: String

The field that receives the 8-byte encrypted PIN block.

Restrictions

The restrictions for CSNBCPE.

The format control specified in the PIN profile must be NONE.

Required commands

The required commands for CSNBCPE.

This verb requires the **Clear PIN Encrypt** command (offset X'00AF') to be enabled in the active role.

An enhanced PIN security mode is available for formatting an encrypted PIN-block into IBM 3621 or 3624 format using the **PADDIGIT** PIN-extraction

method. This mode limits checking of the PIN to decimal digits. No other PIN-block consistency checking will occur. To activate this mode, enable the **Enhanced PIN Security** command (offset X'0313') in the active role.

In order to access key storage, this verb requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

Usage notes for CSNBCPE.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBCPEJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBCPEJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           byte[]            PIN_encrypting_key_identifier,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            clear_PIN,
           byte[]            PIN_profile,
           byte[]            PAN_data,
           hikmNativeInteger sequence_number,
           byte[]            encrypted_PIN_block);
```

Clear PIN Generate (CSNBPGN)

Use the Clear PIN Generate verb to generate a clear PIN, a PIN validation value (PVV), or an offset according to an algorithm.

You supply the algorithm or process rule using the *rule_array* parameter.

- IBM 3624 (IBM-PIN or IBM-PINO)
- VISA PIN validation value (VISA-PVV)
- Interbank PIN (INBK-PIN)

For guidance information about VISA, see their appropriate publications.

Format

The format of CSNBPGN.

```
CSNBPGN(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             PIN_generating_key_identifier,
             rule_array_count,
             rule_array,
             PIN_length,
             PIN_check_length,
             data_array,
             returned_result)
```

Parameters

The parameters for CSNBPGN.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

PIN_generating_key_identifier

Direction: Input/Output
Type: String

The 64-byte key label or internal key token that identifies the PIN generation (**PINGEN**) key.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1.

rule_array

Direction: Input
Type: String array

The process rule provides control information to the verb. The keyword is left-aligned in an 8-byte field and padded on the right with blanks. The *rule_array* keyword is described in Table 133.

Table 133. Keywords for Clear PIN Generate control information

Keyword	Description
Process Rule (One, required)	
GBP-PIN	The IBM German Bank Pool PIN, which uses the institution **PINGEN** key to generate an institution PIN (IPIN).
IBM-PIN	The IBM 3624 PIN, which is an institution-assigned PIN. It does not calculate the PIN offset.
IBM-PINO	The IBM 3624 PIN offset, which is a customer-selected PIN and calculates the PIN offset (the output).
INBK-PIN	The Interbank PIN that is generated.
VISA-PVV	The VISA PIN validation value. Input is the customer PIN.

PIN_length

Direction: Input
Type: Integer

The length of the PIN used for the IBM algorithms only, **IBM-PIN** or **IBM-PINO**. Otherwise, this parameter is ignored. Specify an integer in the range 4 - 16.

PIN_check_length

Direction: Input
Type: Integer

The length of the PIN offset used for the **IBM-PINO** process rule only. Otherwise, this parameter is ignored. Specify an integer from 4 - 16.

Note: The PIN check length must be less than or equal to the integer specified in the *PIN_length* parameter.

data_array

Direction: Input
Type: String

Three 16-byte data elements required by the corresponding *rule_array* parameter. The data array consists of three 16-byte fields or elements whose specification depends on the process rule. If a process rule only requires one or two 16-byte fields, the rest of the data array is ignored by the verb. Table 134 describes the array elements.

Table 134. Array elements for the Clear PIN Generate verb

Array element	Description
Clear_PIN	Clear user selected PIN of 4 - 12 digits of 0 - 9. Left-aligned and padded with spaces. For **IBM-PINO**, this is the clear customer PIN (CSPIN).
Decimalization_table	Decimalization table for IBM and GBP only. Sixteen digits of 0 - 9. If the **ANSI X9.8 PIN - Use stored decimalization table only** access control point (X'0356') is enabled in the active role, this table must match one of the active decimalization tables in the coprocessors.
Trans_sec_parm	For VISA only, the leftmost sixteen digits. Eleven digits of the personal account number (PAN). One digit key index. Four digits of customer selected PIN. For Interbank only, sixteen digits. Eleven rightmost digits of the personal account number (PAN). A constant of 6. One digit key selector index. Three digits of PIN validation data.
Validation_data	Validation data for IBM and IBM German Bank Pool padded to 16 bytes. One to sixteen characters of hexadecimal account data left-aligned and padded on the right with blanks.

Table 135 lists the data array elements required by the process rule (*rule_array* parameter). The numbers refer to the process rule's position within the array.

Table 135. Array elements for Clear PIN Generate

Process Rule	IBM-PIN	IBM-PINO	GBP-PIN	GBP-PINO	VISA-PVV	INBK-PIN
Decimalization_table	1	1	1	1		
Validation_data	2	2	2	2		

Table 135. Array elements for Clear PIN Generate (continued)

Process Rule	IBM-PIN	IBM-PINO	GBP-PIN	GBP-PINO	VISA-PVV	INBK-PIN
Clear_PIN		3		3		
Trans_sec_parm					1	1

Note: Generate offset for GBP algorithm is equivalent to IBM offset generation with *PIN_check_length* of 4 and *PIN_length* of 6.

returned_result

Direction: Output
Type: String

The 16-byte generated output, left-aligned, and padded on the right with blanks.

Restrictions

The restrictions for CSNBPGN.

None.

Required commands

The required commands for CSNBPGN.

This verb requires the **Clear PIN Generate - 3624** command (offset X'00A0') to be enabled in the active role.

Whenever the **ANSI X9.8 PIN - Use stored decimalization table only** command (offset X'0356') is enabled in the active role, the Decimalization_table element of the *data_array* value must match one of the PIN decimalization tables that are in the active state on the coprocessor. Use of this command provides improved security and control for PIN decimalization tables.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBPGN.

If you are using the IBM 3624 PIN and IBM German Bank Pool PIN algorithms, you can supply an unencrypted customer selected PIN to generate a PIN offset.

Related information

Related information about CSNBPGN.

The algorithms are described in detail in Chapter 20, "PIN formats and algorithms," on page 913.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBPGNJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBPGNJ(
           hikmNativeInteger   return_code,
           hikmNativeInteger   reason_code,
           hikmNativeInteger   exit_data_length,
           byte[]              exit_data,
           byte[]              PIN_generating_key_identifier,
           hikmNativeInteger   rule_array_count,
           byte[]              rule_array,
           hikmNativeInteger   PIN_length,
           hikmNativeInteger   PIN_check_length,
           byte[]              data_array,
           byte[]              returned_result);
```

Clear PIN Generate Alternate (CSNBCPA)

Use the Clear PIN Generate Alternate verb to generate a clear VISA PVV (PIN validation value) from an input encrypted PIN block or to produce a 3624 offset from a customer-selected encrypted PIN.

The PIN block can be encrypted under either an input PIN-encrypting key (**IPINENC**) or an output PIN-encrypting key (**OPINENC**).

Format

The format of CSNBCPA.

```
CSNBCPA(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             PIN_encryption_key_identifier,
             PIN_generation_key_identifier,
             PIN_profile,
             PAN_data,
             encrypted_PIN_block,
             rule_array_count,
             rule_array,
             PIN_check_length,
             data_array,
             returned_PVV)
```

Parameters

The parameters for CSNBCPA.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

PIN_encryption_key_identifier

Direction: Input/Output
Type: String

A 64-byte string consisting of an internal token that contains an **IPINENC** or **OPINENC** key or the label of an **IPINENC** or **OPINENC** key that is used to encrypt the PIN block. If you specify a label, it must resolve uniquely to either an **IPINENC** or **OPINENC** key.

PIN_generation_key_identifier

Direction: Input/Output
Type: String

A 64-byte string that consists of an internal token that contains a PIN generation (**PINGEN**) key or the label of a PINGEN key.

PIN_profile

Direction: Input
Type: String

The three 8-byte character elements that contain information necessary to extract a PIN from a formatted PIN block. The pad digit is needed to extract the PIN from a 3624 or 3621 PIN block in the Clear PIN Generate Alternate verb. See "The PIN profile" on page 449 for additional information.

PAN_data

Direction: Input
Type: String

A 12-byte field that contains 12 characters of PAN data. The personal account number recovers the PIN from the PIN block if the PIN profile specifies **ISO-0** or **VISA-4** block formats. Otherwise it is ignored, but you must specify this parameter.

For **ISO-0** or **ISO-3**, use the rightmost 12 digits of the PAN, excluding the check digit. For **VISA-4**, use the leftmost 12 digits of the PAN, excluding the check digit.

encrypted_PIN_block

Direction: Input
Type: String

An 8-byte field that contains the encrypted PIN that is input to the VISA PVV generation algorithm. The verb uses the **IPINENC** or **OPINENC** key that is specified in the *PIN_encryption_key_identifier* parameter to encrypt the block.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1 or 2. If the default extraction method for a PIN block format is desired, specify the *rule_array_count* value as 1.

rule_array

Direction: Input
Type: String array

The process rule for the PIN generation algorithm. Specify **IBM-PINO** or **VISA-PVV** (the VISA PIN verification value) in an 8-byte field, left-aligned, and padded with blanks. The *rule_array* points to an array of one or two 8-byte elements. The *rule_array* keywords are described in Table 136.

Table 136. Keywords for Clear PIN Generate Alternate control information

Keyword	Description
PIN calculation method (One required)	

Table 136. Keywords for Clear PIN Generate Alternate control information (continued)

Keyword	Description
IBM-PINO	This keyword specifies use of the IBM 3624 PIN Offset calculation method.
VISA-PVV	This keyword specifies use of the VISA PVV calculation method.
PIN extraction method (One optional) See the text following this table.	

If the *PIN extraction method* is provided, one of the PIN extraction method keywords shown in Table 123 on page 450 can be specified for the given PIN block format. See "PIN block format and PIN extraction method keywords" on page 450 for additional information. If the default extraction method for a PIN block format is desired, specify the *rule_array_count* value as 1.

The PIN extraction methods operate as follows:

PINBLOCK
Specifies that the verb use one of the following:

- The PIN length, if the PIN block contains a PIN length field
- The PIN delimiter character, if the PIN block contains a PIN delimiter character.

PADDIGIT
Specifies that the verb use the pad value in the PIN profile to identify the end of the PIN.

HEXDIGIT
Specifies that the verb use the first occurrence of a digit in the range from X'A' to X'F' as the pad value to determine the PIN length.

PINLEN*nn*
Specifies that the verb use the length specified in the keyword, where *nn* can range from 04 - 16, to identify the PIN.

The PINLEN*nn* keywords are disabled for this verb by default. If these keywords are used, return code 8 with reason code 33 is returned. To enable them, the **Enhanced PIN Security** command (bit X'0313') must be enabled using a TKE.

PADEXIST
Specifies that the verb use the character in the 16th position of the PIN block as the value of the pad value.

PIN_check_length

Direction: Input
Type: Integer

The length of the PIN offset used only for the **IBM-PINO** process rule. Otherwise, this parameter is ignored. Specify an integer from 4 - 16.

Note: The PIN check length must be less than or equal to the integer specified in the *PIN_length* parameter.

data_array

Direction: Input
Type: String

Three 16-byte elements. Table 137 describes the format when **IBM-PINO** is specified. Table 138 describes the format when **VISA-PVV** is specified.

Table 137. Array elements for Clear PIN Generate Alternate, data_array (IBM-PINO)

Array element	Description
Decimalization_table	This element contains the decimalization table of 16 characters (0 - 9) that are used to convert hexadecimal digits (X'0' - X'F') of the enciphered validation data to the decimal digits (X'0' - X'9'). If the ANSI X9.8 PIN - Use stored decimmalization table only access control point (X'0356') is enabled in the active role, this table must match one of the active decimalization tables in the coprocessors.
validation_data	This element contains 1 - 16 characters of account data. The data must be left-aligned and padded on the right with space characters.
Reserved-3	This field is ignored, but you must specify it.

Table 138. Array elements for Clear PIN Generate Alternate, data_array (VISA-PVV)

Array element	Description
Trans_sec_parm	For **VISA-PVV** only, the leftmost twelve digits. Eleven digits of the personal account number (PAN). One digit key index. The rest of the field is ignored.
Reserved-2	This field is ignored, but you must specify it.
Reserved-3	This field is ignored, but you must specify it.

returned_PVV

Direction: Output
Type: String

A 16-byte area that contains the 4-byte PVV left-aligned and padded with blanks.

Restrictions

The restrictions for CSNBCPA.

None.

Required commands

The required commands for CSNBCPA.

This verb requires the commands shown in the following table to be enabled in the active role based on the keyword specified for the PIN-calculation method:

Rule-array keyword	Offset	Command
IBM-PINO	X'00A4'	**Clear PIN Generate Alternate - 3624 Offset**
VISA-PVV	X'00BB'	**Clear PIN Generate Alternate - VISA PVV**

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

An enhanced PIN security mode, on the CEX*C is available for extracting PINs from encrypted PIN blocks. This mode only applies when specifying a

PIN-extraction method for an IBM 3621 or an IBM 3624 PIN-block. To do this, you must enable the **Enhanced PIN Security** (offset X'0313') access control point in the default role. When activated, this mode limits checking of the PIN to decimal digits and a PIN length minimum of 4 is enforced. No other PIN-block consistency checking will occur.

An enhanced PIN security mode starting with CEX3C is available beginning with Release 4.1.0, to implement restrictions required by the ANSI X9.8 PIN standard. The restrictions are to accept only a *PIN_profile* variable that contains a PIN-block format of ISO-0 or ISO-3. To enforce these restrictions, you must enable the following access control points in the default role:

- **ANSI X9.8 PIN - Enforce PIN block restrictions** (X'0350')

For more information, see "ANSI X9.8 PIN restrictions" on page 447.

Note: A role with offset X'0350' enabled also affects access control of the Encrypted PIN Translate and the Secure Messaging for PINs verbs.

Whenever the **ANSI X9.8 PIN - Use stored decimalization tables only** command (offset X'0356') is enabled in the active role, the Decimalization_table element of the *data_array* value must match one of the PIN decimalization tables that are in the active state on the coprocessor. Use of this command provides improved security and control for PIN decimalization tables. The **VISA-PVV** PIN-calculation method does not have a Decimalization_table element and is therefore not affected by this command.

Usage notes

The usage notes for CSNBCPA.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBCPAJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBCPAJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           byte[]            inbound_PIN_encrypting_key_identifier,
           byte[]            PIN_generating_key_identifier,
           byte[]            input_PIN_profile,
           byte[]            PAN_data,
           byte[]            encrypted_PIN_block,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger PIN_check_length,
           byte[]            data_array,
           byte[]            returned_result);
```

CVV Generate (CSNBCSG)

Use the CVV Generate verb to generate a VISA Card Verification Value (CVV) or MasterCard Card Verification Code (CVC) as defined for track 2.

This verb generates a CVV that is based on the information that the *PAN_data*, the *expiration_date*, and the *service_code* parameters provide. This verb uses the Key-A and the Key-B keys to cryptographically process this information. Key-A and Key-B can be single-length **DATA** or **MAC** keys, or a combined Key-A, Key-B double length DATA or MAC key. If the requested CVV is shorter than 5 characters, the CVV is padded on the right by space characters. The CVV is returned in the 5-byte variable that the *CVV_value* parameter identifies. When you verify a CVV, compare the result to the value that the *CVV_value* supplies.

See CVV Key Combine (CSNBCKC) for information on combining two single-length MAC-capable keys into one double-length key.

Format

The format of CSNBCSG.

```
CSNBCSG(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              rule_array_count,
              rule_array,
              PAN_data,
              expiration_date,
              service_code,
              CVV_key_A_Identifier,
              CVV_key_B_Identifier,
              CVV_value)
```

Parameters

The parameters for CSNBCSG.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0, 1, or 2.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. Each keyword is left-aligned in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage. The *rule_array* keywords are described in Table 139.

Table 139. Keywords for CVV Generate control information

Keyword	Description
PAN data length (One, optional)	
PAN-13	Specifies that the length of the PAN data is 13 bytes. **PAN-13 is the default value.**

Table 139. Keywords for CVV Generate control information (continued)

Keyword	Description
PAN-14	Specifies that the length of the PAN data is 14 bytes.
PAN-15	Specifies that the length of the PAN data is 15 bytes.
PAN-16	Specifies that the length of the PAN data is 16 bytes.
PAN-17	Specifies that the length of the PAN data is 17 bytes.
PAN-18	Specifies that the length of the PAN data is 18 bytes.
PAN-19	Specifies that the length of the PAN data is 19 bytes.
CVV length (One, optional)	
CVV-1	Specifies that the CVV is to be computed as one byte, followed by four blanks. **CVV-1 is the default value.**
CVV-2	Specifies that the CVV is to be computed as two bytes, followed by three blanks.
CVV-3	Specifies that the CVV is to be computed as three bytes, followed by two blanks.
CVV-4	Specifies that the CVV is to be computed as four bytes, followed by one blank.
CVV-5	Specifies that the CVV is to be computed as five bytes.

PAN_data

Direction: Input
Type: String

The *PAN_data* parameter specifies an address that points to the place in application data storage that contains personal account number (PAN) information in character form. The PAN is the account number as defined for the track-2 magnetic-stripe standards. If the **PAN-***nn* keyword is specified in the *rule_array*, where *nn* is a value between 13 and 19, then *nn* number of characters are processed.

If you specify the **PAN-***nn* keyword in the *rule_array* where *nn* is less than 16, the server might copy 16 bytes to a work area. Therefore, ensure that the verb can address 16 bytes of storage.

expiration_date

Direction: Input
Type: String

The *expiration_date* parameter specifies an address that points to the place in application data storage that contains the card expiration date in numeric character form in a 4-byte field. The application programmer must determine whether the CVV will be calculated with the date form of YYMM or MMYY.

service_code

Direction: Input
Type: String

The *service_code* parameter specifies an address that points to the place in application data storage that contains the service code in numeric character form in a 3-byte field. The service code is the number that the track-2 magnetic-stripe standards define. The service code of X'000' is supported.

CVV_key_A_Identifier

Direction: Input/Output
Type: String

A 64-byte string that is the internal key token containing a single or double-length **DATA** or **MAC** key, or the label of a key storage record containing a single or double-length **DATA** or **MAC** key.

When this key is a double-length key, *CVV_key_B_identifier* must be 64 byte of binary zero. When a double-length **MAC** key is used, the CV bits 0 - 3 must indicate a CVVKEY-A key (B'0010').

A single-length key contains the key-A key that encrypts information in the CVV process. The left half of a double-length key contains the key-A key that encrypts information in the CVV process and the right half contains the key-B key that decrypts information.

CVV_key_B_Identifier

Direction: Input/Output
Type: String

A pointer to a 64-byte internal key token or a key label of a single-length **DATA** or **MAC** key that decrypts information in the CCV process. The internal key token contains the Key-B key that decrypts information in the CVV process.

When *CVV_key_A_identifier* is a double-length key, this parameter must be 64 byte of binary zero.

CVV_value

Direction: Output
Type: String

A pointer to the location in application data storage that will be used to store the computed 5-byte character output value.

Restrictions

The restrictions for CSNBCSG.

None.

Required commands

The required commands for CSNBCSG.

This verb requires the **VISA CVV Generate** command (offset X'00DF') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBCSG.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBCSGJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBCSGJ(
          hikmNativeInteger return_code,
          hikmNativeInteger reason_code,
          hikmNativeInteger exit_data_length,
          byte[]            exit_data,
          hikmNativeInteger rule_array_count,
          byte[]            rule_array,
          byte[]            PAN_data,
          byte[]            expiration_date,
          byte[]            service_code,
          byte[]            CVV_key_A_Identifier,
          byte[]            CVV_key_B_Identifier,
          byte[]            CVV_value);
```

CVV Key Combine (CSNBCKC)

Use the CVV Key Combine verb to combine two operational DES keys into one operational TDES key.

The verb accepts as input two single-length keys that are suitable for use with the CVV (card-verification value) algorithm. The resulting double-length key meets a more recent industry standard of using TDES to support PIN-based transactions. In addition, the double-length key is in a format that can be wrapped using the Key Export to TR31 verb.

The CVV Generate and CVV Verify verbs use the CVV algorithm to generate and verify card security codes required by Visa (CVV) and MasterCard (CVC). Previously, these verbs only accepted as input two single-length MAC-capable keys. These verbs will additionally accept as input a double-length MAC or MAC-capable **DATA** key that contains key-A as the left half of the key, and key-B as the right half of the key. The double-length key must be usable with either the CVV Generate verb, the CVV Verify verb, or both.

The CVV Key Combine verb allows combining most pairs of single-length DES keys that formerly functioned as a separate key-A and key-B into one double-length CVVKEY-A key. The CVVKEY-A attribute in the control vector is now changed to mean single-length CVV key containing key-A or double-length CVV key containing key-A and key-B.

To use this verb, specify the following:

- Up to two optional rule-array keywords:
  1. A key wrapping method keyword that specifies whether to use the new enhanced wrapping method, the original wrapping method, or the wrapping method defined as the default according to a configuration setting.
  2. A translation control keyword that restricts the translation method to the enhanced method.
- A single-length operational DES key for **key-A**

  Identify a single-length operational DES key that has a key type of **MAC** or **DATA**. The key identifier length must be 64, which is the length of a DES key-token or a key label. This parameter identifies the **key-A** key used with the

CVV algorithm. It is placed in the **left** half of the double-length output key. When a **MAC** key is identified, it must have as its subtype extension ANY-MAC (CV bits 0 - 3 = B'0000') or CVVKEY-A (CV bits 0 - 3 = B'0010'). If a **DATA** key is identified, it must have its MAC generate bit on (CV bit 20), its MAC verify bit on (CV bit 21), or both bits on.

- A single-length operational DES key for **key-B**

  Identify a single-length operational DES key that has a key type of **MAC** or **DATA**. The key identifier length must be 64, which is the length of a DES key-token or a key label. This parameter identifies the **key-B** key used with the CVV algorithm. It is placed in the **right** half of the double-length output key. When a **MAC** key is identified, it must have as its subtype extension ANY-MAC (CV bits 0 - 3 = B'0000') or CVVKEY-B (CV bits 0 - 3 = B'0011'). If a **DATA** key is identified, it must have its MAC generate bit on (CV bit 20), its MAC verify bit on (CV bit 21), or both bits on.

- An output key identifier

  Identify a null key-token in a 64-byte buffer, or the key label of a DES null key-token. If the input parameter identifies a key label, the output key is placed in DES key-storage. otherwise, the output is returned in the buffer provided.

  The following table shows the various output combinations that are returned for the MAC generate and MAC verify attributes. These results are based on the three possible MAC generate and MAC verify control-vector-bit combinations (bits 20 - 21) that the pair of input keys can have.

CV bits 20 - 21 of input key key-A, single length	**CV bits 20 - 21 of input key key-B, single length**		
	MAC generate andMAC verify	**MAC generate only,single length**	**MAC verify only**
MAC generate andMAC verify	MAC generate and MAC verifydouble-length key-A	MAC generate only double-length key-A	MAC verify only double-length key-A
MAC generate only	MAC generate onlydouble-length key-A	MAC generate only double-length key-A	Invalid combination, control vector conflict
MAC verify only	MAC verify onlydouble-length key-A	Invalid combination, control vector conflict	MAC verify only double-length key-A

Format

The format of CSNBCKC.

```
CSNBCKC(
           return_code,
           reason_code,
           exit_data_length,
           exit_data,
           rule_array_count,
           rule_array,
           key_a_identifier_length,
           key_a_identifier,
           key_b_identifier_length,
           key_b_identifier,
           output_key_identifier_length
           output_key_identifier)
```

Parameters

The parameters for CSNBCKC.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 0, 1, or 2.

rule_array

Direction: Input
Type: String array

A pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* keywords are:

Keyword	Meaning
Key wrapping method (one, optional)	
USECONFG	This is the default. Specifies to wrap the key using the configuration setting for the default wrapping method. The default wrapping method configuration setting may be changed using the TKE. This keyword is ignored for AES keys.
WRAP-ECB	Specifies to wrap the key using the legacy wrapping method.
WRAP-ENH	Specifies to wrap the key using the enhanced wrapping method.
Translation control (optional). This is valid only with wrapping method **WRAP-ENH** or with **USECONFG** when the default wrapping method is **WRAP-ENH**. This option cannot be used on a key with a control vector valued to binary zeros.	
ENH-ONLY	Specifies to restrict the key from being wrapped with the legacy wrapping method after it has been wrapped with the enhanced wrapping method. Sets bit 56 (ENH-ONLY) of the control vector to B'1'.

There are restrictions on the available wrapping methods for the output key derived from the wrapping methods employed and control vector restrictions of the input keys. These are detailed in Table 140.

Table 140. Key-wrapping matrix for the CVV Key Combine verb

key-A or key-B wrapped using WRAP-ENH method	key-A or key-B has ENH-ONLY bit on (CV bit 56 = B'1')	WRAP-ENH (by keyword or by default)	ENH-ONLY keyword	Resulting form of output key or error
No	No	No to both	No	ECB wrapped
Yes	No	No to both	No	Wrap type conflict, 8/2161 (X'871')
No	No	Yes to either	No	WRAP-ENH, CV bit 56 = B'0' (NOT set)
Yes	No	Yes to either	No	
No	No	Yes to either	Yes	WRAP-ENH, CV bit 56 = B'1' (IS set)
Yes	No	Yes to either	Yes	
Yes	Yes	Yes to either	No	
Yes	Yes	Yes to either	Yes	

Table 140. Key-wrapping matrix for the CVV Key Combine verb (continued)

key-A or key-B wrapped using WRAP-ENH method	key-A or key-B has ENH-ONLY bit on (CV bit 56 = B'1')	WRAP-ENH (by keyword or by default)	ENH-ONLY keyword	Resulting form of output key or error
No	No	No to both	Yes	CV bit 56 conflict, 8/2111 (X'83F')
Yes	No	No to both	Yes	
Yes	Yes	No to both	No	
Yes	Yes	No to both	Yes	

key_a_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes in the *key_a_identifier* variable. This value must be 64.

key_a_identifier

Direction: Input
Type: String

A pointer to a string variable containing either the operational single-length DES key-token of the key-A key, or the label of such a key token. This key must be a MAC key or a DATA key that can perform MAC operations.

key_b_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes in the *key_b_identifier* variable. This value must be 64.

key_b_identifier

Direction: Input
Type: String

A pointer to a string variable containing either the operational single-length DES key-token of the key-B key, or the label of such a key token. This key must be a **MAC** key or a **DATA** key that can perform MAC operations.

output_key_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes in the *output_key_identifier* variable. This value must be 64.

output_key_identifier

Direction: Input/Output
Type: String

A pointer to a string variable containing a NULL key token, or the key label of a null DES key-token.

Restrictions

The restrictions for CSNBCKC.

Input key-A and input key-B cannot have different export control bits (CV bit 17 and 57); these bits must match. Use the Prohibit Export verb to change XPORT-OK to NO-XPORT (CV bit 17), or the Restrict Key Attribute verb to change T31XPTOK to NOT31XPT (CV bit 57). These two bits are propagated to the output key-token.

Required commands

The required commands for CSNBCKC.

The CVV Key Combine verb requires the **CVV Key Combine** command (offset X'0155') to be enabled in the active role.

In addition, these commands are required to be enabled in the active role, depending on the key-wrapping method keyword:

Keyword	Offset	Command
WRAP-ECB	X'0156'	**CVV Key Combine - Allow wrapping override keywords**
WRAP-ENH	X'0156'	**CVV Key Combine - Allow wrapping override keywords**

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

One additional restriction is related to combining the key-A and key-B pair of keys when there are mixed types. To permit the combination of mixed key types into a single set of types (**ANY-MAC**, **CVVKEY-A**, **CVVKEY-B**, and **DATA**), enable the **CVV Key Combine - Permit mixed key types** command (offset X'0157') in the active role. See Table 141 for when this command is required:

Table 141. Required commands for the CVV Key Combine verb, mixed key types

	Input key-B			
Input key-A	**ANY-MAC**	**CVVKEY-A**	**CVVKEY-B**	**DATA**
ANY-MAC	Always returns double-length **CVVKEY-A** key	Invalid combination, control vector conflict	Only returns double-length **CVVKEY-A** key if X'0157' enabled	Only returns double-length **CVVKEY-A** key if X'0157' enabled
CVVKEY-A	Only returns double-length **CVVKEY-A** key if X'0157' enabled, else access error	Invalid combination, control vector conflict	Always returns double-length **CVVKEY-A** key	Only returns double-length **CVVKEY-A** key if X'0157' enabled
CVVKEY-B	Invalid combination	Invalid combination	Invalid combination	Invalid combination
DATA	Only returns double-length **CVVKEY-A** key if X'0157' enabled, else access error	Invalid combination, control vector conflict	Only returns double-length **CVVKEY-A** key if X'0157' enabled, else access error	Always returns double-length **CVVKEY-A** key

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBCKCJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBCKCJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger key_a_identifier_length,
           byte[]            key_a_identifier,
           hikmNativeInteger key_b_identifier_length,
           byte[]            key_b_identifier,
           hikmNativeInteger output_key_identifier_length,
           byte[]            output_key_identifier);
```

CVV Verify (CSNBCSV)

Use the CVV Verify verb to verify a VISA Card Verification Value (CVV) or MasterCard Card Verification Code (CVC) as defined for track 2.

This verb generates a CVV based on the information the *PAN_data*, the *expiration_date*, and the *service_code* parameters provide. This verb uses the Key-A and the Key-B keys to cryptographically process this information. If the requested CVV is shorter than 5 characters, the CVV is padded on the right by space characters. The generated CVV is then compared to the value that the *CVV_value* supplies for verification.

See CVV Key Combine (CSNBCKC) for information on combining two single-length MAC-capable keys into one double-length key.

Format

The format of CSNBCSV.

```
CSNBCSV(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             PAN_data,
             expiration_date,
             service_code,
             CVV_key_A_Identifier,
             CVV_key_B_Identifier,
             CVV_value)
```

Parameters

The parameters for CSNBCSV.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0, 1, or 2.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. Each keyword is left-aligned in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage. The *rule_array* keywords are described in Table 142.

Table 142. Keywords for CVV Verify control information

Keyword	Description
PAN data length (One, optional)	
PAN-13	Specifies that the length of the PAN data is 13 bytes. **PAN-13 is the default value.**
PAN-14	Specifies that the length of the PAN data is 14 bytes.
PAN-15	Specifies that the length of the PAN data is 15 bytes.
PAN-16	Specifies that the length of the PAN data is 16 bytes.
PAN-17	Specifies that the length of the PAN data is 17 bytes.
PAN-18	Specifies that the length of the PAN data is 18 bytes.
PAN-19	Specifies that the length of the PAN data is 19 bytes.
CVV length (One, optional)	
CVV-1	Specifies that the CVV is to be computed as one byte, followed by four blanks. **CVV-1 is the default value.**
CVV-2	Specifies that the CVV is to be computed as two bytes, followed by three blanks.
CVV-3	Specifies that the CVV is to be computed as three bytes, followed by two blanks.
CVV-4	Specifies that the CVV is to be computed as four bytes, followed by one blank.
CVV-5	Specifies that the CVV is to be computed as five bytes.

PAN_data

Direction: Input
Type: String

The *PAN_data* parameter specifies an address that points to the place in application data storage that contains personal account number (PAN) information in character form. The PAN is the account number as defined for the track-2 magnetic-stripe standards. If the **PAN-***nn* keyword is specified in the *rule_array*, where *nn* is a value between 13 and 19, then *nn* number of characters are processed.

If you specify the **PAN-***nn* keyword in the *rule_array* where *nn* is less than 16, the server might copy 16 bytes to a work area. Therefore, ensure that the verb can address 16 bytes of storage.

expiration_date

Direction: Input
Type: String

The *expiration_date* parameter specifies an address that points to the place in application data storage that contains the card expiration date in numeric character form in a 4-byte field. The application programmer must determine whether the CVV will be calculated with the date form of YYMM or MMYY.

service_code

Direction: Input
Type: String

The *service_code* parameter specifies an address that points to the place in application data storage that contains the service code in numeric character form in a 3-byte field. The service code is the number that the track-2 magnetic-stripe standards define. The service code of X'000' is supported.

CVV_key_A_Identifier

Direction: Input/Output
Type: String

A 64-byte string that is the internal key token containing a single or double-length **DATA** or **MAC** key, or the label of a key storage record containing a single or double-length **DATA** or **MAC** key.

When this key is a double-length key, *CVV_key_B_Identifier* must be 64 byte of binary zero. When a double-length **MAC** key is used, the CV bits 0 - 3 must indicate a CVVKEY-A key (B'0010').

A single-length key contains the key-A key that encrypts information in the CVV process. The left half of a double-length key contains the key-A key that encrypts information in the CVV process and the right half contains the key-B key that decrypts information.

CVV_key_B_Identifier

Direction: Input/Output
Type: String

A pointer to a 64-byte internal key token or a key label of a single-length **DATA** or **MAC** key that decrypts information in the CCV process. The internal key token contains the Key-B key that decrypts information in the CVV process.

When *CVV_key_A_Identifier* is a double-length key, this parameter must be 64 byte of binary zero.

CVV_value

Direction: Input
Type: String

The *CVV_value* parameter specifies an address that contains the CVV value which will be compared to the computed CVV value. This is a 5-byte field.

Restrictions

The restrictions for CSNBCSV.

None.

Required commands

The required commands for CSNBCSV.

This verb requires the **VISA CVV Verify** command (offset X'00E0') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBCSV.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBCSVJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBCSVJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            PAN_data,
           byte[]            expiration_date,
           byte[]            service_code,
           byte[]            CVV_key_A_Identifier,
           byte[]            CVV_key_B_Identifier,
           byte[]            CVV_value);
```

Encrypted PIN Generate (CSNBEPG)

The Encrypted PIN Generate verb formats a PIN and encrypts the PIN block.

To generate the PIN, the verb uses one of the following PIN calculation methods:

- IBM 3624 PIN
- IBM German Bank Pool Institution PIN
- Interbank PIN

To format the PIN, the verb uses one of the following PIN block formats:

- IBM 3621 format
- IBM 3624 format
- ISO-0 format (same as the ANSI X9.8, VISA-1, and ECI-1 formats)
- ISO-1 format (same as the ECI-4 format)
- ISO-2 format

- ISO-3 format
- IBM 4704 encrypting PINPAD (**4704-EPP**) format
- VISA 2 format
- VISA 3 format
- VISA 4 format
- ECI-2 format
- ECI-3 format

Format

The format of CSNBEPG.

```
CSNBEPG(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             PIN_generating_key_identifier,
             outbound_PIN_encrypting_key_identifier
             rule_array_count,
             rule_array,
             PIN_length,
             data_array,
             PIN_profile,
             PAN_data,
             sequence_number
             encrypted_PIN_block)
```

Parameters

The parameters for CSNBEPG.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

PIN_generating_key_identifier

Direction: Input/Output
Type: String

The 64-byte internal key token or a key label of an internal key token in the DES key storage file. The internal key token contains the PIN-generating key. The control vector must specify the **PINGEN** key type and have the EPINGEN usage bit set to B'1'.

outbound_PIN_encrypting_key_identifier

Direction: Input
Type: String

A 64-byte internal key token or a key label of an internal key token in the DES key storage file. The internal key token contains the key to be used to encrypt the formatted PIN and must contain a control vector that specifies the OPINENC key type and has the EPINGEN usage bit set to B'1'.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the

rule_array variable. This value must be 1.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. Each keyword is left-aligned in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage. The *rule_array* keywords are described in Table 143.

Table 143. Keywords for Encrypted PIN Generate control information

Keyword	Description
Processing rule (One, required)	
GBP-PIN	This keyword specifies the IBM German Bank Pool Institution PIN calculation method is to be used to generate a PIN.
IBM-PIN	This keyword specifies the IBM 3624 PIN calculation method is to be used to generate a PIN.
INBK-PIN	This keyword specifies the Interbank PIN calculation method is to be used to generate a PIN.

PIN_length

Direction: Input
Type: String

An integer defining the PIN length for those PIN calculation methods with variable length PINs. Otherwise, the variable should be set to zero.

data_array

Direction: Input
Type: Integer

Three 16-byte character strings, which are equivalent to a single 48-byte string. The values in the data array depend on the keyword for the PIN calculation method. Each element is not always used, but you must always declare a complete data array. The numeric characters in each 16-byte string must be from 1 - 16 bytes in length, uppercase, left-aligned, and padded on the right with space characters. Table 144 describes the array elements.

Table 144. Array elements for Encrypted PIN Generate data_array parameter

Array element	Description
Decimalization_table	Decimalization table for IBM and GBP only. Sixteen characters that are used to map the hexadecimal digits (X'0' - X'F') of the encrypted validation data to decimal digits (X'0' - X'9'). If the **ANSI X9.8 PIN - Use stored decimalization tables only** command (offset X'0356') access control point is enabled in the active role, this table must match one of the active decimalization tables in the coprocessors.
Trans_sec_parm	For Interbank only, sixteen digits. Eleven rightmost digits of the personal account number (PAN). A constant of 6. One digit key selector index. Three digits of PIN validation data.
Validation_data	Validation data for IBM and IBM German Bank Pool padded to 16 bytes. 1 - 16 characters of hexadecimal account data left-aligned and padded on the right with blanks.

Table 145 on page 487 lists the data array elements required by the process rule (*rule_array* parameter). The numbers refer to the process rule's position within the array.

Table 145. Keywords for Encrypted PIN Generate control information

Process rule	IBM-PIN	GBP-PIN	INBK-PIN
Decimalization_table	1	1	
Validation_data	2	2	
Trans_sec_parm			1

PIN_profile

Direction: Input
Type: String array

A 24-byte string containing the PIN profile including the PIN block format. See "The PIN profile" on page 449 for additional information.

PAN_data

Direction: Input
Type: String

A 12-byte string that contains 12 digits of Personal Account Number (PAN) data. The verb uses this parameter if the PIN profile specifies the **ISO-0**, **ISO-3**, or **VISA-4** keyword for the PIN block format. Otherwise, ensure that this parameter is a 4-byte variable in application storage. The information in this variable will be ignored, but the variable must be specified.

Note: When using the **ISO-0** or **ISO- 3** keywords, use the 12 rightmost digits of the PAN data, excluding the check digit. When using the **VISA-4** keyword, use the 12 leftmost digits of the PAN data, excluding the check digit.

sequence_number

Direction: Input
Type: Integer

The 4-byte string that contains the sequence number used by certain PIN block formats. The verb uses this parameter if the PIN profile specifies the 3621 or **4704-EPP** keyword for the PIN block format. Otherwise, ensure this parameter is a 4-byte variable in application data storage. The information in the variable will be ignored, but the variable must be declared. To enter a sequence number, do the following:

- Enter 99999 to use a random sequence number that the service generates.
- For the 3621 PIN block format, enter a value in the range from 0 - 65,535.
- For the 4704-EPP PIN block format, enter a value in the range from 0 - 255.

encrypted_PIN_block

Direction: Output
Type: String

The field where the verb returns the 8-byte encrypted PIN.

Restrictions

The restrictions for CSNBEPG.

The format control specified in the PIN profile must be NONE.

Required commands

The required commands for CSNBEPG.

This verb requires the commands, as shown in the following table, to be enabled in the active role based on the keyword specified for the PIN-calculation methods.

Rule-array keyword	Offset	Command
IBM-PIN	X'00B0'	**Encrypted PIN Generate - 3624**
GBP-PIN	X'00B1'	**Encrypted PIN Generate - GBP**
INBK-PIN	X'00B2'	**Encrypted PIN Generate - Interbank**

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

An enhanced PIN security mode is available for formatting an encrypted PIN block into IBM 3621 or 3624 format using the **PADDIGIT** PIN-extraction method. This mode limits checking of the PIN to decimal digits, and a minimum PIN length of 4 is enforced; no other PIN-block consistency checking will occur. To activate this mode, enable the **Enhanced PIN Security** command (offset X'0313') in the active role.

Whenever the **ANSI X9.8 PIN - Use stored decimalization tables only** command (offset X'0356') command is enabled in the active role, the Decimalization_table element of the *data_array* value must match one of the PIN decimalization tables that are in the active state on the coprocessor. Use of this command provides improved security and control for PIN decimalization tables. The **INBK-PIN** PIN-calculation method does not have a Decimalization_table element and is therefore not affected by this command.

Usage notes

The usage notes for CSNBEPG.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBEPGJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBEPGJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           byte[]            PIN_generating_key_identifier,
           byte[]            outbound_PIN_encrypting_key_identifier,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger PIN_length,
           byte[]            data_array,
           byte[]            PIN_profile,
           byte[]            PAN_data,
           hikmNativeInteger sequence_number,
           byte[]            encrypted_PIN_block);
```

Encrypted PIN Translate (CSNBPTR)

Use the Encrypted PIN Translate verb to re-encipher a PIN block from one PIN-encrypting key to another and, optionally, to change the PIN block format, such as the pad digit or sequence number.

The unique-key-per-transaction key derivation for single and double-length keys is available for the Encrypted PIN Translate verb. This support is available for the *input_PIN_encrypting_key_identifier* and the *output_PIN_encrypting_key_identifier* parameters for both **REFORMAT** and **TRANSLAT** process rules. The *rule_array* keyword determines which PIN keys are derived keys.

The Encrypted PIN Translate verb can be used for unique-key-per-transaction key derivation.

Format

The format of CSNBPTR.

```
CSNBPTR(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              input_PIN_encrypting_key_identifier,
              output_PIN_encrypting_key_identifier,
              input_PIN_profile,
              PAN_data_in,
              PIN_block_in,
              rule_array_count,
              rule_array,
              output_PIN_profile,
              PAN_data_out,
              sequence_number,
              PIN_block_out)
```

Parameters

The parameters for CSNBPTR.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

input_PIN_encrypting_key_identifier

Direction: Input/Output
Type: String

The input PIN-encrypting key (**IPINENC**) for the *PIN_block_in* parameter specified as a 64-byte internal key token or a key label. If keyword **UKPTOPIN**, **UKPTBOTH**, **DUKPT-IP**, or **DUKPT-BH** is specified in the *rule_array* parameter, the *input_PIN_encrypting_key_identifier* must specify a key token or key label of a **KEYGENKY** with the UKPT usage bit enabled.

output_PIN_encrypting_key_identifier

Direction: Input/Output
Type: String

The output PIN-encrypting key (**OPINENC**) for the *PIN_block_out* parameter specified as a 64-byte internal key token or a key label. If keyword

UKPTOPIN, **UKPTBOTH**, **DUKPT-IP**, or **DUKPT-BH** is specified in the *rule_array* parameter, the *output_PIN_encrypting_key_identifier* must specify a key token or key label of a **KEYGENKY** with the UKPT usage bit enabled.

input_PIN_profile

Direction: Input
Type: String

The three 8-byte character elements that contain information necessary to either create a formatted PIN block or extract a PIN from a formatted PIN block. A particular PIN profile can be either an input PIN profile or an output PIN profile depending on whether the PIN block is being enciphered or deciphered by the verb. See "The PIN profile" on page 449 for additional information.

If you choose the **TRANSLAT** processing rule or the **REFORMAT** processing rule in the *rule_array* parameter, the input PIN profile and output PIN profile can have different PIN block formats. If you specify **UKPTIPIN** with **DUKPT-IP** or **UKPTBOTH** with **DUKPT-BH** in the *rule_array* parameter, the *input_PIN_profile* is extended to a 48-byte field and must contain the current key serial number. See "The PIN profile" on page 449 for additional information.

The pad digit is needed to extract the PIN from a 3624 or 3621 PIN block in the Encrypted PIN Translate verb with a process rule (*rule_array* parameter) of **REFORMAT**. If the process rule is **TRANSLAT**, the pad digit is ignored.

The **PINLEN***nn* keywords are disabled for this verb by default. If these keywords are used, return code 8 with reason code 33 is returned. To enable them, the PTR Enhanced PIN Security access control point (bit X'0313') must be enabled using a TKE.

PAN_data_in

Direction: Input
Type: String

The personal account number (PAN) if the process rule (*rule_array* parameter) is **REFORMAT** and the input PIN format is **ISO-0**, **ISO-3**, or **VISA-4** only. Otherwise, this parameter is ignored. Specify 12 digits of account data in character format.

For **ISO-0** or **ISO-3**, use the rightmost 12 digits of the PAN, excluding the check digit.

For **VISA-4**, use the leftmost 12 digits of the PAN, excluding the check digit.

PIN_block_in

Direction: Input
Type: String

The 8-byte enciphered PIN block that contains the PIN to be translated.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1, 2, or 3.

rule_array

Direction: Input
Type: String array

The process rule for the verb is described in Table 146.

Table 146. Keywords for Encrypted PIN Translate control information

Keyword	Description
Processing rule (One, required)	
REFORMAT	Changes the PIN format, the contents of the PIN block, and the PIN-encrypting key.
TRANSLAT	Changes the PIN-encrypting key only. It does not change the PIN format and the contents of the PIN block.
PIN block format and PIN extraction method (Optional)	See "PIN block format and PIN extraction method keywords" on page 450 for additional information and a list of PIN block formats and PIN extraction method keywords. **Note:** If a PIN extraction method is not specified, the first one listed in Table 123 on page 450 for the PIN block format will be the default.
DUKPT keywords - Single length key derivation (One, optional)	
UKPTIPIN	The *input_PIN_encrypting_key_identifier* is derived as a single length key. The *input_PIN_encrypting_key_identifier* must be a **KEYGENKY** key with the UKPT usage bit enabled. The *input_PIN_profile* must be 48 bytes and contain the key serial number.
UKPTOPIN	The *output_PIN_encrypting_key_identifier* is derived as a single length key. The *output_PIN_encrypting_key_identifier* must be a **KEYGENKY**Y key with the UKPT usage bit enabled. The *output_PIN_profile* must be 48 bytes and contain the key serial number.
UKPTBOTH	Both the *input_PIN_encrypting_key_identifier* and the *output_PIN_encrypting_key_identifier* are derived as a single length key. Both the *input_PIN_encrypting_key_identifier* and the *output_PIN_encrypting_key_identifier* must be **KEYGENKY** keys with the UKPT usage bit enabled. Both the *input_PIN_profile* and the *output_PIN_profile* must be 48 bytes and contain the respective key serial number.
DUKPT keywords - double length key derivation (One, optional)	
DUKPT-IP	The *input_PIN_encrypting_key_identifier* is derived as a double length key. The *input_PIN_encrypting_key_identifier* must be a **KEYGENKY** key with the UKPT usage bit enabled. The *input_PIN_profile* must be 48 bytes and contain the key serial number.
DUKPT-OP	The *output_PIN_encrypting_key_identifier* is derived as a double length key. The *output_PIN_encrypting_key_identifier* must be a **KEYGENKY** key with the UKPT usage bit enabled. The *output_PIN_profile* must be 48 bytes and contain the key serial number.
DUKPT-BH	Both the *input_PIN_encrypting_key_identifier* and the *output_PIN_encrypting_key_identifier* are derived as a double length key. Both the *input_PIN_encrypting_key_identifier* and the *output_PIN_encrypting_key_identifier* must be **KEYGENKY** keys with the UKPT usage bit enabled. Both the *input_PIN_profile* and the *output_PIN_profile* must be 48 bytes and contain the respective key serial number.

`output_PIN_profile`

Direction: Input
Type: String

The three 8-byte character elements that contain information necessary to either create a formatted PIN block or extract a PIN from a formatted PIN block. A particular PIN profile can be either an input PIN profile or an output PIN profile, depending on whether the PIN block is being enciphered or deciphered by the verb.

- If you choose the **TRANSLAT** processing rule in the *rule_array* parameter, the *input_PIN_profile* and the *output_PIN_profile* must specify the same PIN block format.

- If you choose the **REFORMAT** processing rule in the *rule_array* parameter, the input PIN profile and output PIN profile can have different PIN block formats.
- If you specify **UKPTOPIN** or **UKPTBOTH** in the *rule_array* parameter, the *output_PIN_profile* is extended to a 48-byte field and must contain the current key serial number. See "The PIN profile" on page 449 for additional information.
- If you specify **DUKPT-OP** or **DUKPT-BH** in the *rule_array* parameter, the *output_PIN_profile* is extended to a 48-byte field and must contain the current key serial number. See "The PIN profile" on page 449 for additional information.

PAN_data_out

Direction: Input
Type: String

The personal account number (PAN) if the process rule (*rule_array* parameter) is **REFORMAT** and the output PIN format is **ISO-0**, **ISO-3**, or **VISA-4** only. Otherwise, this parameter is ignored. Specify 12 digits of account data in character format.

For **ISO-0** or **ISO-3**, use the rightmost 12 digits of the PAN, excluding the check digit.

For **VISA-4**, use the leftmost 12 digits of the PAN, excluding the check digit.

sequence_number

Direction: Output
Type: Integer

The sequence number if the process rule (*rule_array* parameter) is **REFORMAT** and the output PIN block format is 3621 or 4704-EPP only. Specify the integer value 99999. Otherwise, this parameter is ignored.

PIN_block_out

Direction: Input
Type: String

The 8-byte output PIN block that is re-enciphered.

Restrictions

The restrictions for CSNBPTR.

None.

Required commands

The required commands for CSNBPTR.

This verb requires the commands, as shown in the following table, to be enabled in the active role based on the keyword specified for the PIN-calculation methods.

Rule-array keyword	Input profile format control keyword	Output profile format control keyword	Offset	Command
TRANSLAT	NONE	NONE	X'00B3'	**Encrypted PIN Translate - Translate**
REFORMAT	NONE	NONE	X'00B7'	**Encrypted PIN Translate - Reformat**

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

This verb also requires the **UKPT - PIN Verify, PIN Translate** command (offset X'00E1') to be enabled if you employ UKPT processing.

Note: A role with offset X'00E1' enabled can also use the Encrypted PIN Verify verb with UKPT processing.

An enhanced PIN security mode is available for extracting PINs from a 3621 or 3624 encrypted PIN-block and formatting an encrypted PIN block into IBM 3621 or 3624 format using the **PADDIGIT** PIN-extraction method. This mode limits checking of the PIN to decimal digits, and a minimum PIN length of 4 is enforced; no other PIN-block consistency checking will occur. To activate this mode, enable the **Enhanced PIN Security** command (offset X'0313') in the active role.

The verb returns an error indicating that the PAD digit is not valid if all of these conditions are met:

1. The **Enhanced PIN Security** command is enabled in the active role.
2. The output PIN profile specifies 3621 or 3624 as the PIN-block format.
3. The output PIN profile specifies a decimal digit (0 - 9) as the PAD digit.

Beginning with Release 4.1.0, three new commands are added (offsets X'0350', X'0351', and X'0352'). The list hereafter describes how these three commands affect the PIN processing.

1. Enable the **ANSI X9.8 PIN - Enforce PIN block restrictions** command (offset X'0350') in the active role to apply additional restrictions to PIN processing implemented in CCA 4.1.0, as follows:
   - Do not translate or reformat a non-ISO PIN block into an ISO PIN block. Specifically, do not allow an IBM 3624 PIN-block format in the **output_PIN_profile** variable when the PIN-block format in the **input_PIN_profile** variable is not IBM 3624.
   - Constrain use of ISO-2 PIN blocks to offline PIN verification and PIN change operations in integrated circuit card environments only. Specifically, do not allow ISO-2 input or output PIN blocks.
   - Do not translate or reformat a PIN-block format that includes a PAN into a PIN-block format that does not include a PAN. Specifically, do not allow an ISO-1 PIN-block format in the *output_PIN_profile* variable when the PIN-block format in the *input_PIN_profile* variable is **ISO-0** or **ISO-3**.
   - Do not allow a change of PAN data. Specifically, when performing translations between PIN block formats that both include PAN data, do not allow the *input_PAN_data* and *output_PAN_data* variables to be different from the PAN data enciphered in the input PIN block.

   Note: A role with offset X'0350' enabled also affects access control of the Clear PIN Generate Alternate and the Secure Messaging for PINs verbs.
2. Enable the **ANSI X9.8 PIN - Allow modification of PAN** command (offset X'0351') in the active role to override the restriction to not allow a change of PAN data. This override is applicable only when either the ANSI X9.8 PIN - Enforce PIN block restrictions command (offset X'0350') or the **ANSI X9.8 PIN - Allow only ANSI PIN blocks** command (offset X'0352') or both are enabled in the active role. This override is to support account number changes in issuing environments. Offset X'0351' has no effect if neither offset X'0350' nor offset X'0352' is enabled in the active role.

Note: A role with offset X'0351' enabled also affects access control of the Secure Messaging for PINs verbs.

3. Enable the **ANSI X9.8 PIN - Allow only ANSI PIN blocks** command (offset X'0352') in the active role to apply a more restrictive variation of the **ANSI X9.8 PIN - Enforce PIN block restrictions** command (offset X'0350'). In addition to the previously described restrictions of offset X'0350', this command also restricts the *input_PIN_profile* and the *output_PIN_profile* to contain only ISO-0, ISO-1, and **ISO-3** PIN block formats. Specifically, the IBM 3624 PIN-block format is not allowed with this command. Offset X'0352' overrides offset X'0350'.

   Note: A role with offset X'0352' enabled also affects access control of the Secure Messaging for PINs verbs.

For more information, see "ANSI X9.8 PIN restrictions" on page 447.

Usage notes

The usage notes for CSNBPTR.

Some PIN block formats are known by several names. The following table shows the additional names.

Table 147. Additional names for PIN formats

PIN format	Additional name
ISO-0	ANSI X9.8, VISA format 1, ECI format 1
ISO-1	ECI format 4

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBPTRJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBPTRJ(
          hikmNativeInteger return_code,
          hikmNativeInteger reason_code,
          hikmNativeInteger exit_data_length,
          byte[]            exit_data,
          byte[]            input_PIN_encrypting_key_identifier,
          byte[]            output_PIN_encrypting_key_identifier,
          byte[]            input_PIN_profile,
          byte[]            input_PAN_data,
          byte[]            input_PIN_block,
          hikmNativeInteger rule_array_count,
          byte[]            rule_array,
          byte[]            output_PIN_profile,
          byte[]            output_PAN_data,
          hikmNativeInteger sequence_number,
          byte[]            output_PIN_block);
```

Encrypted PIN Translate Enhanced (CSNBPTRE)

The Encrypted PIN Translate Enhanced verb reformats a PIN into a different PIN-block format using an enciphered PAN field. You can use this verb in an interchange-network application, or to change the PIN block to conform to the format and encryption key used in a PIN-verification database.

The CSNBPTRE verb supports Visa Data Secure Platform (VDSP, formerly known as Visa Merchant Data Secure (VMDS)) processing. With this verb you can also use *derived unique key per transaction (DUKPT)* PIN-block encryption (ANS X9.24) for both input and output PIN blocks. The verb supports translation of PINs whose PAN information has been enciphered using the VDSP standard and Visa Format Preserving Encryption (VFPE) methods.

PIN blocks are sometimes formatted using the PAN information. For the CSNBPTRE verb, either the input PIN block profile or the output PIN block profile must specify a PIN block format that incorporates a PAN. The PIN block formats which incorporate a PAN are ISO-0, ISO-3, and Visa Format 4. VDSP enciphered PAN data can be enciphered using DUKPT key management or static TDES key management. The enciphered PAN could be enciphered with the CBC mode or the VFPE mode. VDSP requires that the same key management scheme and type of keys are used for both the PIN and PAN. For VDSP, the following pairings are supported:

Table 148. Pairings supported for VDSP.

Function	Source		Target	
	Key management	**VDSP option**	**Key management**	**VDSP option**
Translation	DUKPT	Standard CBC	Static TDES non-DUKPT (Zone Encryption Keys)	Standard CBC
		VFPE		
	Static TDES non-DUKPT	Standard CBC		

Terminology: The VDSP specification speaks of two key management methods: *DUKPT (derived unique key per transaction)* and *Zone Encryption Keys*. The process for deriving these keys is documented in ANS X9.24 Part 1. *Zone Encryption Keys* are called *static keys* in CCA. Static keys are presented for use and are not derived during verb processing. They are double length TDES keys for this service which are called *static TDES keys* in this document.

The verb operates in reformat mode. In reformat mode, the verb performs the translate-mode functions (changes the wrapping key) and, in addition, processes the clear text information. Following the rules that you specify, the PIN is extracted from the recovered clear text PIN block using the specified input PIN encrypting key and formatted into an output PIN block according to the output PIN profile for encryption. The PIN block is re-enciphered with the specified output PIN encrypting key. Change of PAN data is not allowed.

The Encrypted PIN Translate Enhanced verb performs the following processing:

- It decrypts the input PIN-block by using the supplied IPINENC key in ECB mode, or derives the decryption key using the specified KEYGENKY key and the current-key serial number (CKSN), and then uses ANS X9.24-specified special decryption or the Triple-DES (TDES) method. The PAN must be

deciphered using either the data decryption key derived from the base derivation key and CKSN or using the specified static TDES data decryption key (DECIPHER, CIPHER).

- Checks the control vector of the input PIN encryption key to ensure that for an IPINENC key the REFORMAT bit (CV bit 22) is set to B'1' for reformat mode, or for a KEYGENKY key, that the UKPT bit (CV bit 18) is set to B'1'. Likewise the OPINENC key must have the REFORMAT bit set according to the requested mode.
- In reformat mode, performs these steps:
  - It extracts the PIN from the specified PIN-block format using the method specified by default or by a rule-array keyword. If required by the PIN-block format, PAN data is used in the extraction process.
  - Formats the extracted-PIN into the format declared for the output PIN-block. As required by the PIN-block format, the verb incorporates PAN data, sequence number, and pad character information in formatting the output.
- It enciphers the output PIN-block by using the supplied static OPINENC key in ECB mode, or derives the decryption key using the specified KEYGENKY key and the output current-key serial number (CKSN) from the output PIN profile and uses ANS X9.24-specified special encryption or Triple-DES method. The REFORMAT bit must be set to B'1' in the OPINENC control vector, or the UKPT bit must be set to B'1' in the KEYGENKY control vector.

Format

The format of CSNBPTRE.

```
CSNBPTRE(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             input_PIN_encrypting_key_identifier_length,
             input_PIN_encrypting_key_identifier,
             output_PIN_encrypting_key_identifier_length,
             output_PIN_encrypting_key_identifier,
             PAN_encrypting_key_identifier_length,
             PAN_encrypting_key_identifier,
             input_PIN_profile_length,
             input_PIN_profile,
             PAN_data_length,
             PAN_data,
             input_PIN_block_length,
             input_PIN_block,
             output_PIN_profile_length,
             output_PIN_profile,
             sequence_number,
             output_PIN_block_length,
             output_PIN_block,
             reserved1_length,
             reserved1,
             reserved2_length,
             reserved2)
```

Parameters

The parameters for CSNBPTRE.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. The value must be in the range 6 – 9.

rule_array

Direction: Input
Type: String array

A pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length, and are left-aligned and padded on the right with space characters. The returned rule array keywords express the contents of the token.

Table 149. Keywords for Encrypted PIN Translate Enhanced control information

Keyword	Description
Processing method (required)	
VMDS	Specifies that the VDSP method (Visa Data Secure Platform method, formally known as the Visa Merchant Data Secure (VMDS) method) is to be used for processing.
Mode (required)	
REFORMAT	Specifies that either the PIN-block format and the PIN-block encryption, or both, are to be changed. If the PIN-extraction method is not chosen by default, another element in the rule array must specify one of the keywords that indicates a PIN-extraction method.
Input PAN data key management method (one, required) These keywords are used to define the PAN-encrypting key used to decrypt the **PAN_data** parameter.	
IN-DUKPT	Specifies that the key to be used to decrypt the PAN data is to be derived using the key specified in the **input_PIN_encrypting_key_identifier**. See the description of the **input_PIN_encrypting_key_identifier** for the requirements of the key. The DUKPT-BH or DUKPT-IP keyword is required.
OUTDUKPT	Specifies that the key to be used to decrypt the PAN data is to be derived using the key specified in the **output_PIN_encrypting_key_identifier**. See the description of the **output_PIN_encrypting_key_identifier** for the requirements of the key. The DUKPT-BH or DUKPT-OP keyword is required.
STATIC	Specifies the use of static double length (2-key) Triple-DES symmetric keys for the PAN.
Input data algorithm (one, required)	
TDES	Specifies that Triple-DES encryption was used for the PAN.
Input data mode (one, required)	
CBC	Specifies that CBC mode encryption was used for the PAN. This is the mode for the Standard Encryption option.
VFPE	Specifies that Visa format preserving encryption was used for the PAN.
PAN4BITX	Specifies that the PAN data character set is 4-bit hex. Two digits per byte. Not valid with the CBC rule.
PAN8BITA	Specifies that the PAN data character set is normal ASCII, represented in binary format. Not valid with CBC rule.
PAN-EBLK	Specifies that the PAN data is in a CBC encrypted block. Valid only with CBC rule.
PAN check digit compliance (one required if mode VFPE and PAN input character set keyword are present, otherwise not allowed)	

Table 149. Keywords for Encrypted PIN Translate Enhanced control information (continued)

Keyword	Description
CMPCKDGT	Last digit of the PAN contains a compliant check digit per ISO/IEC 7812-1.
NONCKDGIT	Last digit of the PAN does not contain a compliant check digit per ISO/IEC 7812-1.
Unique Key Per Transaction (one, optional)	
DUKPT-BH	Specifies the use of UKPT key-derivation and PIN-block ciphering for both input and output processing, Triple-DES method.
DUKPT-IP	Specifies the use of UKPT input-key derivation and PIN-block decryption, Triple-DES method.
DUKPT-OP	Specifies the use of UKPT output-key derivation and PIN-block encryption, Triple-DES method.
PIN-extraction method (one, optional). If the PIN block format is provided, one of the PIN extraction method keywords shown in Table 123 on page 450 can be specified for the given PIN block format. **Note:** Specify the **PIN block format** keyword in the **PIN_profile** variable (see "PIN block format" on page 450). The following PIN-block formats are supported: 3624 ISO-0 ISO-1 ISO-2 ISO-3 The following S390 formats are also supported: VISA-2, VISA-3, VISA-4, OEM-1, ECI-2, ECI-3. See "PIN block format and PIN extraction method keywords" on page 450 for additional information. If the default extraction method for a PIN block format is desired, specify the **rule_array_count** value as 1.	

input_PIN_encrypting_key_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the **input_PIN_encrypting_key_identifier** variable. Set the value to 64.

input_PIN_encrypting_key_identifier

Direction: Input
Type: String

A pointer to a string variable containing an operational fixed-length DES key-token or a key label of an operational fixed-length DES key-token record.

This is the identifier of the key to decrypt the input PIN block. Or it is the key-generating key to be used to derive the key to decrypt the input PIN block. The key-generating key can optionally be used to derive the key to decrypt the PAN data.

The key identifier is an operational token or the key label of an operational token in key storage. If you do not use the UKPT process or you specify the DUKPT-OP rule array keyword, the key token must contain the PIN-encrypting key to be used to decipher the input PIN block. The key algorithm must be DES, the key type must be IPINENC and the key usage REFORMAT bit must be enabled (B'1').

If you use the UKPT process for the input PIN block by specifying the DUKPT-IP or the DUKPT-BH rule array keyword, the key token must contain

the key-generating key to derive the PIN-encrypting key. If you have also specified the IN-DUKPT keyword, the key is used to derive the key to decrypt the PAN data. The key algorithm must be DES, the key type must be KEYGENKY and the key usage UKPT bit must be enabled.

output_PIN_encrypting_key_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the **output_PIN_encrypting_key_identifier** variable. Set the value to 64.

output_PIN_encrypting_key_identifier

Direction: Input
Type: String

A pointer to a string variable containing an operational key token or a key label of an operational key token record.

The identifier of the key to encrypt the output PIN block or the key-generating key to be used to derive the key to encrypt the output PIN block. The key-generating key can optionally be used to derive the key to decrypt the PAN data.

The key identifier is an operational token or the key label of an operational token in key storage. If you do not use the UKPT process or you specify the DUKPT-IP rule array keyword, the key token must contain the PIN-encrypting key to be used to encipher the output PIN block. The key algorithm must be DES, the key type must be OPINENC and the key usage REFORMAT bit must be enabled.

If you use the UKPT process for the output PIN block by specifying the DUKPT-OP or the DUKPT-BH rule array keyword, the key token must contain the key-generating key to derive the PIN-encrypting key. If you have also specified the OUTDUKPT keyword, the key is used to derive the key to decrypt the PAN data. The key algorithm must be DES, the key type must be KEYGENKY and the key usage UKPT bit must be enabled (B'1').

PAN_key_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the **PAN_key_identifier** variable. Set the value to 0 if the PAN key management method keyword specifies DUKPT. Set it to 64 if the PAN key management method specifies STATIC.

PAN_key_identifier

Direction: Input
Type: String

A pointer to a string variable containing an internal fixed-length DES key-token or the key label of such a record in DES key-storage. This token contains a double-length data decryption key if the input PAN data key management method is STATIC. If the key token contains a double-length data encryption key (Zone Encryption Key in the VDSP specification), the token must have a key type of CIPHER or DECIPHER.

The key is used to decipher the input PAN data. If IN-DUKPT is specified as the input PAN data key management method, the base derivation key in the **input_PIN_encrypting_key_identifier** parameter is used to create the decryption key for the PAN.

If OUTDUKPT is specified as the input PAN data key management method, the base derivation key in the **output_PIN_encrypting_key_identifier** parameter is used to create the decryption key for the PAN. This parameter is not used if IN-DUKPT or OUTDUKPT is specified.

The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm must be DES, the key type must be CIPHER or DECIPHER and the key must be a double-length key.

Note: DATA keys with ENC or DEC bits on are not supported. Also, zero CV data keys are not supported.

input_PIN_profile_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the **input_PIN_profile** variable. Set the value to 24 if the profile does not contain a CKSN extension. Otherwise, set the value to 48.

input_PIN_profile

Direction: Input
Type: String

A pointer to a string variable containing three 8-byte character strings with information defining the PIN-block format and, optionally, an additional 24 bytes containing the input CKSN extension. The strings are equivalent to 24-byte or 48-byte strings.

PAN_data_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the **PAN_data** parameter if the mode is CBC. If the mode is VFPE, this variable contains the number of PAN digits. The value is in the range 15 - 19 for VFPE. It is 16 if the standard encryption option is selected.

PAN_data

Direction: Input
Type: String

A pointer to a string variable containing the PAN data. For VFPE mode, if the PAN contains an odd number of 4-bit hex digits, the data must be left justified in the PAN variable and the right-most 4 bits are ignored. The verb uses this data to recover the PIN from the PIN block if you specify the REFORMAT keyword and the input PIN profile specifies the ISO-0, VISA-4 or ISO-3 keyword for the PIN-block format. If the output PIN profile specifies the ISO-0, VISA-4, or ISO-3 keyword for the PIN-block format, the 12 rightmost digits of the (decrypted) PAN, excluding the check digit, are used to format the output PIN block.

input_PIN_block_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the encrypted input PIN block. The value must be 8.

input_PIN_block

Direction: Input
Type: String

A pointer to a string variable containing the encrypted PIN-block.

output_PIN_profile_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the **output_PIN_profile** variable. The value is 24 or 48.

output_PIN_profile

Direction: Input
Type: String

A pointer to a string variable containing three 8-byte character strings with information defining the PIN-block format and, optionally, an additional 24 bytes containing the output CKSN extension. The strings are equivalent to 24-byte or 48-byte strings.

sequence_number

Direction: Input
Type: Integer

A pointer to an integer variable containing the sequence number. Ensure that the referenced integer variable is valued to 99999 if the output PIN block format is 3621 or 4704-EPP. Otherwise, this parameter is ignored.

output_PIN_block_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the re-enciphered PIN block. The value must be at least 8.

output_PIN_block

Direction: Output
Type: String

A pointer to a string variable containing the re-enciphered and, optionally, reformatted PIN-block returned by the verb. The buffer can be larger on input. However, on output this field is updated to indicate the actual number of bytes returned by the card.

reserved1_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the

reserved1 variable. This value must be zero.

reserved1

Direction: Input
Type: String

A pointer to a string variable. This parameter is reserved for future use.

reserved2_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the **reserved2** variable. This value must be zero.

reserved2

Direction: Output
Type: String

A pointer to a string variable. This parameter is reserved for future use.

Restrictions

The restrictions for CSNBPTRE.

None.

Required commands

The required commands for CSNBPTRE.

The Encrypted PIN Translate Enhanced verb requires the **Encrypted PIN Translate Enhanced** command (offset X'02D5') to be enabled in the active role. This ACP is ON by default. The following additional commands must be enabled depending on the capabilities that are requested:

Rule-array keyword	Input profile format control keyword	Output profile format control keyword Explanation	Offset	Command
REFORMAT	NONE	NONE	X'00B7'	**Encrypted PIN Translate - Reformat**
One or more of the following: **IN-DUKPT**, **OUTDUKPT**, **DUKPT-OP**, **DUKPT-IP**, or **DUKPT-BH**			X'001E'	**Reencipher CKDS**

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

An enhanced PIN security mode is available for extracting PINs from a 3621 or 3624 encrypted PIN-block and formatting an encrypted PIN block into 3621 or 3624

format using the PADDIGIT PIN-extraction method. This mode limits checking of the PIN to decimal digits, and a minimum PIN length of 4 is enforced. No other PIN-block consistency checking occurs. To activate this mode, enable the **Enhanced PIN Security** command (offset X'0313') in the active role.

The verb returns an error indicating that the PAD digit is not valid if all of these conditions are met:

1. The **Enhanced PIN Security** command is enabled in the active role.
2. The output PIN profile specifies 3621 or 3624 as the PIN-block format.
3. The output PIN profile specifies a decimal digit (0 - 9) as the PAD digit.

Three additional commands should be considered (offsets X'0350', X'0351', and X'0352'). If enabled, these three commands affect how PIN processing by this and other verbs is performed:

1. Enable the **ANSI X9.8 PIN - Enforce PIN block restrictions** command (offset X'0350') in the active role to apply additional restrictions to PIN processing as follows:
   - Do not translate or reformat a non-ISO PIN block into an ISO PIN block. Specifically, do not allow an IBM 3624 PIN-block format in the **output_PIN_profile** variable when the PIN-block format in the **input_PIN_profile** variable is not 3624.
   - Constrain use of ISO-2 PIN blocks to offline PIN verification and PIN change operations in integrated circuit card environments only. Specifically, do not allow ISO-2 input or output PIN blocks.
   - Do not translate or reformat a PIN-block format that includes a PAN into a PIN-block format that does not include a PAN. Specifically, do not allow an ISO-1 PIN-block format in the **output_PIN_profile** variable when the PIN-block format in the **input_PIN_profile** variable is ISO-0 or ISO-3.
2. Enable the **ANSI X9.8 PIN - Allow modification of PAN** command (offset X'0351') in the active role to override the restriction to not allow a change of PAN data. This override is applicable only when either the **ANSI X9.8 PIN - Enforce PIN block restrictions** command (offset X'0350') or the **ANSI X9.8 PIN - Allow only ANSI PIN blocks** command (offset X'0352') or both are enabled in the active role. This override supports environments that issue account number changes. Offset X'0351' has no effect if neither offset X'0350' nor offset X'0352' is enabled in the active role. This rule does not apply for PTRE. Also, PAN changes are not allowed.
3. Enable the **ANSI X9.8 PIN - Allow only ANSI PIN blocks** command (offset X'0352') in the active role to apply a more restrictive variation of the **ANSI X9.8 PIN - Enforce PIN block restrictions** command (offset X'0350'). In addition to the previously described restrictions of offset X'0350', this command also restricts the **input_PIN_profile** and the **output_PIN_profile** parameters to contain only ISO-0, ISO-1, and ISO-3 PIN block formats. Specifically, the IBM 3624 PIN-block format is not allowed with this command. The command at offset X'0352' overrides the one at offset X'0350'.

Usage notes

The usage notes for CSNBPTRE.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBPTREJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBPTREJ(
          hikmNativeInteger   return_code,
          hikmNativeInteger   reason_code,
          hikmNativeInteger   exit_data_length,
          byte[]              exit_data,
          hikmNativeInteger   rule_array_count,
          byte[]              rule_array,
          hikmNativeInteger   input_PIN_encrypting_key_identifier_length,
          byte[]              input_PIN_encrypting_key_identifier,
          hikmNativeInteger   output_PIN_encrypting_key_identifier_length,
          byte[]              output_PIN_encrypting_key_identifier,
          hikmNativeInteger   PAN_encrypting_key_identifier_length,
          byte[]              PAN_encrypting_key_identifier,
          hikmNativeInteger   input_PIN_profile_length,
          byte[]              input_PIN_profile,
          hikmNativeInteger   PAN_data_length,
          byte[]              PAN_data,
          hikmNativeInteger   input_PIN_block_length,
          byte[]              input_PIN_block,
          hikmNativeInteger   output_PIN_profile_length,
          byte[]              output_PIN_profile,
          hikmNativeInteger   sequence_number,
          hikmNativeInteger   output_PIN_block_length,
          byte[]              output_PIN_block,
          hikmNativeInteger   reserved1_length,
          byte[]              reserved1,
          hikmNativeInteger   reserved2_length,
          byte[]              reserved2);
```

Encrypted PIN Verify (CSNBPVR)

Use the Encrypted PIN Verify verb to verify that one of the customer selected trial PINs is valid.

Use the verb to verify that one of the following PINs is valid:

- IBM 3624 (**IBM-PIN**)
- IBM 3624 PIN offset (**IBM-PINO**)
- IBM German Bank Pool (**GBP-PIN**)
- VISA PIN validation value (**VISA-PVV**)
- VISA PIN validation value (**VISAPVV4**)
- Interbank PIN (**INBK-PIN**)

The unique-key-par-transaction key derivation for single and double-length keys is available for the *input_PIN_encrypting_key_identifier* parameter.

Format

The format of CSNBPVR.

```
CSNBPVR(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             input_PIN_encrypting_key_identifier,
             PIN_verifying_key_identifier,
             input_PIN_profile,
             PAN_data,
             encrypted_PIN_block,
             rule_array_count,
             rule_array,
             PIN_check_length,
             data_array)
```

Parameters

The parameters for CSNBPVR.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

input_PIN_encrypting_key_identifier

Direction: Input/Output
Type: String

The 64-byte key label or internal key token containing the PIN-encrypting key (**IPINENC**) that enciphers the PIN block. If keyword **UKPTIPIN** or **DUKPT-IP** is specified in the *rule_array*, the *input_PIN_encrypting_key_identifier* must specify a key token or key label of a **KEYGENKY** with the UKPT usage bit enabled.

PIN_verifying_key_identifier

Direction: Input/Output
Type: String

The 64-byte key label or internal key token that identifies the PIN verify (**PINVER**) key.

input_PIN_profile

Direction: Input
Type: String

The three 8-byte character elements that contain information necessary to either create a formatted PIN block or extract a PIN from a formatted PIN block. A particular PIN profile can be either an input PIN profile or an output PIN profile depending on whether the PIN block is being enciphered or deciphered by the verb. If you specify **UKPTIPIN** in the *rule_array* parameter, the *input_PIN_profile* is extended to a 48-byte field and must contain the current key serial number. See "The PIN profile" on page 449 for additional information.

If you specify **DUKPT-IP** in the *rule_array* parameter, the *input_PIN_profile* is extended to a 48-byte field and must contain the current key serial number. See "The PIN profile" on page 449 for additional information.

The pad digit is needed to extract the PIN from a 3624 or 3621 PIN block in the Encrypted PIN Verify verb.

The **PINLEN***nn* keywords are disabled for this verb by default. If these keywords are used, return code 8 with reason code 33 is returned. To enable them, the PTR Enhanced PIN Security access control point (bit X'0313') must be enabled using a TKE workstation.

PAN_data

Direction: Input
Type: String

The personal account number (PAN) is required for **ISO-0**, **ISO-3**, and **VISA-4**. Otherwise, this parameter is ignored. Specify 12 digits of account data in character format.

For **ISO-0** or **ISO-3**, use the rightmost 12 digits of the PAN, excluding the check digit.

For **VISA-4**, use the leftmost 12 digits of the PAN, excluding the check digit.

encrypted_PIN_block

Direction: Input
Type: String

The 8-byte enciphered PIN block that contains the PIN to be verified.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1, 2, or 3.

rule_array

Direction: Input
Type: String array

The process rule for the PIN verify algorithm, described in Table 150.

Table 150. Keywords for Encrypted PIN Verify control information

Keyword	Description
Algorithm value (One, required)	
GBP-PIN	The IBM German Bank Pool PIN. It verifies the PIN entered by the customer and compares that PIN with the institution generated PIN by using an institution key.
IBM-PIN	The IBM 3624 PIN, which is an institution-assigned PIN. It does not calculate the PIN offset.
IBM-PINO	The IBM 3624 PIN offset, which is a customer-selected PIN and calculates the PIN offset.
INBK-PIN	The Interbank PIN verify algorithm.
VISA-PVV	The VISA PIN verify value.
VISAPVV4	The VISA PIN verify value. If the length is 4 digits, normal processing for VISA-PVV will occur.

Table 150. Keywords for Encrypted PIN Verify control information (continued)

Keyword	Description
PIN block format and PIN extraction method (Optional)	See "PIN block format and PIN extraction method keywords" on page 450 for additional information and a list of PIN block formats and PIN extraction method keywords. The **PINLEN***nn* keywords are disabled for this verb by default. If these keywords are used, return code 8 with reason code 33 is returned. To enable them, the PTR Enhanced PIN Security access control point (bit X'0313') must be enabled using a TKE workstation. **Note:** If a PIN extraction method is not specified, the first one listed in Table 123 on page 450 for the PIN block format will be the default.
DUKPT keyword - single length key derivation (Optional)	
UKPTIPIN	The *input_PIN_encrypting_key_identifier* is derived as a single length key. The *input_PIN_encrypting_key_identifier* must be a **KEYGENKY** key with the UKPT usage bit enabled. The *input_PIN_profile* must be 48 bytes and contain the key serial number.
DUKPT keyword - double length key derivation (Optional)	
DUKPT-IP	The *input_PIN_encrypting_key_identifier* is to be derived using the DUKPT algorithm. The *input_PIN_encrypting_key_identifier* must be a **KEYGENKY** key with the DUKPT usage bit enabled. The *input_PIN_profile* must be 48 bytes and contain the key serial number.

PIN_check_length

Direction: Input
Type: String

The PIN check length for the **IBM-PIN** or **IBM-PINO** process rules only. Otherwise, it is ignored. Specify the rightmost digits, 4 - 16, for the PIN to be verified.

data_array

Direction: Input
Type: Integer

Three 16-byte elements required by the corresponding *rule_array* parameter. The data array consists of three 16-byte fields whose specification depend on the process rule. If a process rule requires only one or two 16-byte fields, the rest of the data array is ignored by the verb. Table 151 describes the array elements.

Table 151. Array elements for Encrypted PIN Verify data_array parameter

Array element	Description
Decimalization_table	Decimalization table for IBM and GBP only. Sixteen decimal digits of 0 - 9. If the **ANSI X9.8 PIN - Use stored decimalization tables only** command (offset X'0356') access control point is enabled in the active role, this table must match one of the active decimalization tables in the coprocessors.
PIN_offset	Offset data for **IBM-PINO**. One to twelve numeric characters, 0 - 9, left-aligned and padded on the right with blanks. For **IBM-PINO**, the PIN offset length is specified in the *PIN_check_length* parameter. For **IBM-PIN** and **GBP-PIN**, the field is ignored.
Trans_sec_parm	For VISA, only the leftmost twelve digits of the 16-byte field are used. These consist of the rightmost eleven digits of the personal account number (PAN) and a one-digit key index. The remaining four characters are ignored. For Interbank only, all 16 bytes are used. These consist of the rightmost eleven digits of the PAN, a constant of X'6', a one-digit key index, and three numeric digits of PIN validation data.

Table 151. Array elements for Encrypted PIN Verify data_array parameter (continued)

Array element	Description
RPVV	For **VISA-PVV** only, referenced PVV (four bytes) that is left-aligned. The rest of the field is ignored.
Validation_data	Validation data for IBM and GBP padded to 16 bytes. 1 - 16 characters of hexadecimal account data left-aligned and padded on the right with blanks.

Table 152 lists the data array elements required by the process rule (*rule_array* parameter). The numbers refer to the process rule's position within the array.

Table 152. Array elements required by the process rule

Process rule	IBM-PIN	IBM-PINO	GBP-PIN	VISA-PVV	INBK-PIN
Decimalization_table	1	1	1		
Validation_data	2	2	2		
PIN_offset	3	3	3		
Trans_sec_parm				1	1
RPVV				2	

Restrictions

The restrictions for CSNBPVR.

None.

Required commands

The required commands for CSNBPVR.

This verb requires the following commands to be enabled in the active role:

Rule-array keyword	Offset	Command
IBM-PIN, **IBM-PINO**	X'00AB'	**Encrypted PIN Verify - 3624**
GBP-PIN	X'00AC'	**Encrypted PIN Verify - GBP**
VISA-PVV, **VISAPVV4**	X'00AD'	**Encrypted PIN Verify - VISA PVV**
INBK-PIN	X'00AE'	**Encrypted PIN Verify - Interbank**

This verb also requires the **UKPT - PIN Verify, PIN Translate** command (offset X'00E1') to be enabled in the active role if you employ UKPT processing.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Note: A role with offset X'00E1' enabled can also use the Encrypted PIN Translate verb with UKPT processing.

An enhanced PIN security mode is available for extracting PINs from a 3621 or 3624 encrypted PIN-block using the **PADDIGIT** PIN-extraction method. This mode limits checking of the PIN to decimal digits, and a minimum PIN length of four is

enforced. No other PIN-block consistency checking will occur. To activate this mode, enable the **Enhanced PIN Security** command (offset X'0313') in the active role.

Whenever the **ANSI X9.8 PIN - Use stored decimalization tables only** command (offset X'0356') is enabled in the active role, the Decimalization_table element of the *data_array* value must match one of the PIN decimalization tables that are in the active state on the coprocessor. Use of this command provides improved security and control for PIN decimalization tables. The **VISA-PVV**, **VISAPVV4**, and **INBK-PIN** PIN calculation methods do not have a Decimalization_table element and are therefore not affected by this command.

Usage notes

The usage notes for CSNBPVR.

None.

Related information

Additional information about CSNBPVR.

The algorithms are described in detail in Chapter 20, "PIN formats and algorithms," on page 913.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBPVRJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBPVRJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           byte[]            PIN_encrypting_key_identifier,
           byte[]            PIN_verifying_key_identifier,
           byte[]            PIN_profile,
           byte[]            PAN_data,
           byte[]            encrypted_PIN_block,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger PIN_check_length,
           byte[]            data_array);
```

FPE Decipher (CSNBFPED)

The FPE Decipher verb is used to decrypt payment card data for the Visa Data Secure Platform (VDSP) processing.

Format

The format of CSNBFPED.

```
CSNBFPED(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             enc_PAN_length,
             enc_PAN,
             enc_cardholder_name_length,
             enc_cardholder_name,
             enc_dtrack1_data_length,
             enc_dtrack1_data,
             enc_dtrack2_data_length,
             enc_dtrack2_data,
             key_identifier_length,
             key_identifier,
             derivation_data_length,
             derivation_data,
             clear_PAN_length,
             clear_PAN,
             clear_cardholder_name_length,
             clear_cardholder_name,
             clear_dtrack1_data_length,
             clear_dtrack1_data,
             clear_dtrack2_data_length,
             clear_dtrack2_data,
             DUKPT_PIN_key_identifier_length,
             DUKPT_PIN_key_identifier,
             reserved1_length,
             reserved1,
             reserved2_length,
             reserved2)
```

Parameters

The parameters for CSNBFPED.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. The minimum value is 5. The maximum value is 10.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. The **rule_array** keywords are described in Table 153 on page 511.

Note: At least one character set keyword is required.

Table 153. Keywords for FPE Decipher control information

Keyword	Description
Processing method (required)	
VMDS	Specifies that the VDSP method (Visa Data Secure Platform method, formally known as the Visa Merchant Data Secure (VMDS) method) is to be used for processing.
Key management method (one required)	
STATIC	Specifies the use of double length (2-key) triple-DES symmetric keys. This is a non-DUKPT key.
DUKPT	Specifies the use of the transaction unique general purpose Data Encryption Keys generated by the DUKPT process at the point of service for data encryption. This is required if VFPE mode is specified. Otherwise, this is optional.
Algorithm (required)	
TDES	Specifies the use of CBC mode triple-DES encryption.
Mode (one required)	
CBC	Specifies the use of CBC mode. This is the mode for the standard encryption option.
VFPE	Specifies the use of Visa format preserving encryption.
PAN input output character set (one required if the **clear_PAN_length** variable is greater than 0. Otherwise, it is not allowed.)	
PAN8BITA	Specifies that the PAN data character set is ASCII represented in binary form. Valid ASCII values are in the range 0 - 9 (X'30' - X'39').
PAN4BITX	Specifies that the PAN data character set is 4-bit hex with two digits per byte. Valid 4-bit hexadecimal values are in the range X'0' - X'9'.
Cardholder name input output character set (required if the **clear_cardholder_name_length** variable is greater than 0. Otherwise, it is not allowed.)	
CN8BITA	Specifies that the cardholder name character set is ASCII represented in binary format, one character per byte. See Table 130 on page 456 for valid characters.
Track_1 input output character set (required if the **clear_dtrack1_data_length** variable is greater than 0. Otherwise, it is not valid.)	
TK18BITA	Specifies that the track 1 discretionary data character set is ASCII represented in binary format, one character per byte. See Table 130 on page 456 for valid characters.
Track_2 input output character set (required if the **clear_dtrack2_data_length** variable is greater than 0. Otherwise, it is not valid.)	
TK28BITA	Specifies that the track 2 discretionary data character set is ASCII represented in binary format, one character per byte. Valid ASCII values are in the range 0 - 9 (X'30' - X'39') and A - F (X'41' - X'46').
PIN encryption key output selection (one, optional, if DUKPT is specified. Otherwise, it is not valid.)	
NOPINKEY	Do not return a DUKPT PIN encryption key. This is the default.
PINKEY	Return a DUKPT PIN encryption key.
PAN check digit compliance (one required if mode VFPE and the pan character set keyword is present. Otherwise, it is not allowed.)	
CMPCKDGT	Last digit of the PAN contains a compliant check digit per ISO/IEC 7812-1.
NONCKDGT	Last digit of the PAN does not contain a compliant check digit per ISO/IEC 7812-1.

enc_PAN_length

Direction: Input
Type: Integer

Specifies the length of the **enc_PAN** parameter in bytes if the mode is CBC or the number of PAN digits if the mode is VFPE. The value is either 0 or in the range 15 - 19 for VFPE. The value must be 0 or 16 if the standard option with

CBC mode is selected. The value is zero when the PAN has not been presented for decryption.

enc_PAN

Direction: Input
Type: String

The enciphered primary account number (PAN) that is to be decrypted. For VFPE mode, if the PAN contains an odd number of 4-bit digits, the data is left justified in the PAN variable and the right-most 4 bits are ignored.

enc_cardholder_name_length

Direction: Input
Type: Integer

Specifies the length of the **enc_cardholder_name** parameter in bytes. The input value is either 0 or in the range 1 - 32, inclusive for VFPE. For the standard method, the input value is either 0 or in the range 2 - 32 for VFPE. For CBC mode, the input value is 0, 16, 24, 32, or 40. The value is zero when the cardholder name has not been presented for decryption.

enc_cardholder_name

Direction: Input
Type: String

The enciphered cardholder full name that is to be decrypted. Only characters in Table 130 on page 456 are valid.

enc_dtrack1_data_length

Direction: Input
Type: Integer

Specifies the length of the **enc_dtrack1_data** parameter in bytes. The input value is either 0 or in the range 1 - 56, inclusive for VFPE. For the standard method, the input value is either 0 or in the range 1 - 56 for VFPE. For CBC mode, the input value is 0 or 16, 24, 32, 40, 48, 56, or 64. The value is zero when the track 1 discretionary data has not been presented for decryption.

enc_dtrack1_data

Direction: Input
Type: String

The encrypted track 1 data that is to be decrypted. Only characters in Table 130 on page 456 are valid.

enc_dtrack2_data_length

Direction: Input
Type: Integer

Specifies the length of the **enc_dtrack2_data** parameter in bytes. The input value is either 0 or in the range 1 - 19 for VFPE. For mode CBC, the input value is 0, 8, or 16. The value is zero when the track 2 discretionary data is not presented for decryption.

enc_dtrack2_data

Direction: Input
Type: String

The encrypted track 2 data that is to be decrypted.

key_identifier_length

Direction: Input
Type: Integer

Specifies the length of the **key_identifier** parameter in bytes. The value must be 64, because only fixed length DES tokens are supported as the key identifier.

key_identifier

Direction: Input/Output
Type: String

The identifier of the key that is used to either decrypt the card data (key management STATIC) or derive the **DUKPT_PIN_key_identifer** (key management DUKPT). The *key identifier* is an operational token or the key label of an operational token in key storage.

For key management DUKPT, the key type must be KEYGENKY. In addition, it must have a control vector with bit 18 equal to B'1' (UKPT). The base derivation key is the one from which the operational keys are derived using the DUKPT algorithm defined in ANS X9.24 Part 1.

For key management STATIC, (Zone Encryption Key in the VDSP specification), the key type must be either CIPHER or ENCIPHER. For production purposes, it is recommended that the key have left and right halves that are not equal.

Note: Data keys are not supported.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

derivation_data_length

Direction: Input
Type: Integer

Specifies the length of the **derivation_data** parameter in bytes. The value must be 10 if a DUKPT key is specified in the **key_identifier** parameter. If a data encryption key is specified in the **key_identifier** parameter, this value must be set to zero.

derivation_data

Direction: Input
Type: String

Contains the 80 bit (10 byte) derivation data that is used as input to the DUKPT derivation process. The derivation data contains the current key serial number (CKSN), which is composed of the 59 bit initial key serial number, and concatenated with the 21 bit value of the current encryption counter, which the device increments for each new transaction. This field is in binary format.

clear_PAN_length

Direction: Input/Output
Type: Integer

Specifies the number of PAN digits in the **clear_PAN** parameter. This value

must either be 0 or in the range 15 - 19, inclusive on output.

clear_PAN

Direction: Output
Type: String

This parameter returns the deciphered primary account number. The full account number, including check digit, is recovered. The data for this parameter is returned in binary format. It is the binary representation of 4-bit hex (keyword PAN4BITX) or ASCII (keyword PAN8BITA) as indicated by the supplied rule array keyword. The clear PAN is left justified in this field.

clear_cardholder_name_length

Direction: Input/Output
Type: Integer

Specifies the length of the **clear_cardholder_name** parameter in bytes. This output value is either 0 or in the range 2 - 32 on output. The variable can be larger on input. However, on output, this field is updated to indicate the actual number of bytes returned by the service.

clear_cardholder_name

Direction: Output
Type: String

This parameter returns the deciphered cardholder full name. The output data for this parameter is in binary format. It is the binary representation of ASCII as indicated by the supplied rule array keyword.

clear_dtrack1_data_length

Direction: Input/Output
Type: Integer

Specifies the length of the **clear_dtrack1_data** parameter in bytes. The output value is either 0 or in the range 1 - 56. The value can be larger on input. However, on output, this field is updated to indicate the actual number of bytes returned by the service.

clear_dtrack1_data

Direction: Output
Type: String

This parameter returns the deciphered discretionary track 1 data.

clear_dtrack2_data_length

Direction: Input/Output
Type: Integer

Specifies the length of the **clear_dtrack2_data** parameter in bytes. The output value is either 0 or in the range 1 - 19. The value can be larger on input. However, on output, this field is updated to indicate the actual number of bytes returned by the service.

clear_dtrack2_data

Direction: Output
Type: String

This parameter returns the deciphered discretionary track 2 data.

DUKPT_PIN_key_identifier_length

Direction: Input/Output
Type: Integer

Specifies the length of the **DUKPT_PIN_key_identifier** parameter in bytes. If the PINKEY rule-array keyword is specified, set this value to 64. Otherwise, set this value to 0. On output, the variable is updated with the length of the data returned in the **DUKPT_PIN_key_identifier** variable.

DUKPT_PIN_key_identifier

Direction: Input/Output
Type: String

On input, this parameter must contain a DES OPINENC or IPINENC skeleton token. On output, it contains the DES token with the derived DES OPINENC or IPINENC key.

reserved1_length

Direction: Input
Type: Integer

Specifies the length of the **reserved1** parameter in bytes. The value must be 0.

reserved1

Direction: Input
Type: String

This parameter is ignored.

reserved2_length

Direction: Input/Output
Type: Integer

Specifies the length of the **reserved2** parameter in bytes. The value must be 0.

reserved2

Direction: Input
Type: String

This parameter is ignored.

Restrictions

The restrictions for CSNBFPED.

None.

Required commands

The required commands for CSNBFPED.

This verb requires the **FPE Decrypt** command (offset X'02D0') to be enabled in the active role. This ACP is ON by default in z/OS. If DUKPT is specified, the verb also requires the **UKPT - PIN Verify, PIN Translate** command (offset X'00E1') to be enabled if you employ DUKPT processing.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBFPED.

See "Usage notes for FPE Encipher (CSNBFPEE) and FPE Decipher (CSNBFPED) services" on page 457.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBFPEDJ .

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBFPEDJ(
           hikmNativeInteger   return_code,
           hikmNativeInteger   reason_code,
           hikmNativeInteger   exit_data_length,
           byte[]              exit_data,
           hikmNativeInteger   rule_array_count,
           byte[]              rule_array,
           hikmNativeInteger   enc_PAN_length,
           byte[]              enc_PAN,
           hikmNativeInteger   enc_cardholder_name_length,
           byte[]              enc_cardholder_name,
           hikmNativeInteger   enc_dtrack1_data_length,
           byte[]              enc_dtrack1_data,
           hikmNativeInteger   enc_dtrack2_data_length,
           byte[]              enc_dtrack2_data,
           hikmNativeInteger   key_identifier_length,
           byte[]              key_identifier,
           hikmNativeInteger   derivation_data_length,
           byte[]              derivation_data,
           hikmNativeInteger   clear_PAN_length,
           byte[]              clear_PAN,
           hikmNativeInteger   clear_cardholder_name_length,
           byte[]              clear_cardholder_name,
           hikmNativeInteger   clear_dtrack1_data_length,
           byte[]              clear_dtrack1_data,
           hikmNativeInteger   clear_dtrack2_data_length,
           byte[]              clear_dtrack2_data,
           hikmNativeInteger   DUKPT_PIN_key_identifier_length,
           byte[]              DUKPT_PIN_key_identifier,
           hikmNativeInteger   reserved1_length,
           byte[]              reserved1,
           hikmNativeInteger   reserved2_length,
           byte[]              reserved2);
```

FPE Encipher (CSNBFPEE)

The FPE Encipher verb is used to encrypt payment card data for the Visa Data Secure Platform (VDSP) processing.

Format

The format of CSNBFPEE.

```
CSNBFPEE(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             clear_PAN_length,
             clear_PAN,
             clear_cardholder_name_length,
             clear_cardholder_name,
             clear_dtrack1_data_length,
             clear_dtrack1_data,
             clear_dtrack2_data_length,
             clear_dtrack2_data,
             key_identifier_length,
             key_identifier,
             derivation_data_length,
             derivation_data,
             enc_PAN_length,
             enc_PAN,
             enc_cardholder_name_length,
             enc_cardholder_name,
             enc_dtrack1_data_length,
             enc_dtrack1_data,
             enc_dtrack2_data_length,
             enc_dtrack2_data,
             DUKPT_PIN_key_identifier_length,
             DUKPT_PIN_key_identifier,
             reserved1_length,
             reserved1,
             reserved2_length,
             reserved2)
```

Parameters

The parameters for CSNBFPEE.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. The minimum value is 5. The maximum value is 10.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. The *rule_array* keywords are described in Table 154 on page 518.

Note: At least one character set keyword is required.

Table 154. Keywords for CSNBFPEE control information

Keyword	Description
Processing method (required)	
VMDS	Specifies that the VDSP method (Visa Data Secure Platform method, formally known as the Visa Merchant Data Secure (VMDS) method) is to be used for processing.
Key management method (one required)	
STATIC	Specifies the use of double length (2-key) triple-DES symmetric keys. This is a non-DUKPT key.
DUKPT	Specifies the use of the transaction unique general purpose Data Encryption Keys generated by the DUKPT process at the point of service for data encryption. This is required if VFPE mode is specified. Otherwise, this is optional.
Algorithm (required)	
TDES	Specifies the use of CBC mode triple-DES encryption.
Mode (one required)	
CBC	Specifies the use of CBC mode. This is the mode for the standard encryption option.
VFPE	Specifies the use of Visa format preserving encryption.
PAN input output character set (one required if the **clear_PAN_length** variable is greater than 0. Otherwise, it is not allowed.)	
PAN8BITA	Specifies that the PAN data character set is BASE-10 ASCII represented in binary form. Valid ASCII values are in the range 0 - 9 (X'30' - X'39').
PAN4BITX	Specifies that the PAN data character set is BASE-10 4-bit hex with two digits per byte. Valid 4-bit hexadecimal values are in the range X'0' - X'9'.
Cardholder name input output character set (required if the **clear_cardholder_name_length** variable is greater than 0. Otherwise, it is not allowed.)	
CN8BITA	Specifies that the cardholder name character set is ASCII represented in binary format, one character per byte. See Table 130 on page 456 for valid characters.
Track_1 input output character set (required if the **clear_dtrack1_data_length** variable is greater than 0. Otherwise, it is not valid.)	
TK18BITA	Specifies that the track 1 discretionary data character set is ASCII represented in binary format, one character per byte. See Table 130 on page 456 for valid characters.
Track_2 input output character set (required if the **clear_dtrack2_data_length** variable is greater than 0. Otherwise, it is not valid.)	
TK28BITA	Specifies that the track 2 discretionary data character set is ASCII represented in binary format, one character per byte. Valid ASCII values are in the range 0 - 9 (X'30' - X'39') and A - F (X'41' - X'46').
PIN encryption key output selection (one, optional)	
NOPINKEY	Do not return a DUKPT PIN encryption key. This is the default.
PINKEY	Return a DUKPT PIN encryption key.
PAN check digit compliance (one required if mode VFPE and the pan input output character set keyword is present. Otherwise, it is not allowed.)	
CMPCKDGT	Last digit of the PAN contains a compliant check digit per ISO/IEC 7812-1.
NONCKDGT	Last digit of the PAN does not contain a compliant check digit per ISO/IEC 7812-1.

clear_PAN_length

Direction: Input
Type: Integer

Specifies the number of digits in the **clear_PAN** parameter. This value must be 0 if **clear_PAN** is not to be enciphered. The value must be in the range 15 - 19 if PAN data is presented for encryption.

clear_PAN

Direction: Input
Type: String

Contains the account number with which the PIN is associated. The full account number, including check digit, should be included. The data for this parameter is in binary format. It is the binary representation of 4-bit hex (keyword PAN4BITX) or ASCII (keyword PAN8BITA). If the PAN contains an odd number of 4-bit digits, the data must be left justified in the PAN variable and the right-most 4 bits are ignored.

If the **clear_PAN_length** parameter is zero, this parameter is ignored.

clear_cardholder_name_length

Direction: Input
Type: Integer

Specifies the length of the **clear_cardholder_name** parameter in bytes. This value must be 0 if the cardholder name is not to be enciphered. The value must be in the range 2 - 32 if the cardholder name data is presented for encryption.

clear_cardholder_name

Direction: Input
Type: String

Contains the card holder's full name. The data for this parameter is in binary format. It is the binary representation of ASCII characters as defined in Table 130 on page 456. Only characters from this table are valid.

clear_dtrack1_data_length

Direction: Input
Type: Integer

Specifies the length of the **clear_dtrack1_data** parameter in bytes. This value must be 0 if the track 1 discretionary data is not to be enciphered. The value must be in the range 1 - 56 if the track 1 discretionary data is presented for encryption.

clear_dtrack1_data

Direction: Input
Type: String

Contains the discretionary data that is stored on track 1 of a magnetic stripe card. This data does not include the PAN, cardholder name, expiration date, or service code. The data for this parameter is in binary format. It is the binary representation of ASCII characters as defined in Table 130 on page 456. Only characters defined in this table are valid.

clear_dtrack2_data_length

Direction: Input
Type: Integer

Specifies the length of the **clear_dtrack2_data** parameter in bytes. This value must be 0 if the track 2 discretionary data is not to be enciphered. The value must be in the range 1 - 19 if the track 2 discretionary data is presented for encryption.

clear_dtrack2_data

Direction: Input
Type: String

Contains the discretionary data that is stored on track 2 of a magnetic stripe card. This data does not include the PAN, expiration date, or service code. The data for this parameter is in binary format. It is the binary representation of ASCII characters. The data for this parameter is in BASE-16 binary format. It is the binary representation of ASCII in the range X'30' - X'39' and X'41' - X'46' (ASCII: 0 - 9 and A - F).

key_identifier_length

Direction: Input
Type: Integer

Specifies the length of the **key_identifier** parameter in bytes. The value must be 64.

key_identifier

Direction: Input/Output
Type: String

The identifier of the key that is used to either encrypt the card data (key management STATIC) or derive the **DUKPT_PIN_key_identifer** (key management DUKPT). The *key identifier* is an operational token or the key label of an operational token in key storage.

For key management DUKPT, the key type must be KEYGENKY. In addition, it must have a control vector with bit 18 equal to B'1' (UKPT). The base derivation key is the one from which the operational keys are derived using the DUKPT algorithm defined in ANS X9.24 Part 1.

For key management STATIC, (Zone Encryption Key in the VDSP specification), the key type must be either CIPHER or ENCIPHER. For production purposes, it is recommended that the key have left and right halves that are not equal.

Note: Data keys are not supported.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

derivation_data_length

Direction: Input
Type: Integer

Specifies the length of the **derivation_data** parameter in bytes. The value must be 10 if a DUKPT key is specified in the **key_identifier** parameter. If a data encryption key is specified in the **key_identifier** parameter, this value must be set to zero.

derivation_data

Direction: Input
Type: String

Contains the 80 bit (10 byte) derivation data that is used as input to the DUKPT derivation process. The derivation data contains the current key serial number (CKSN), which is composed of the 59 bit initial key serial number

concatenated with the 21 bit value of the current encryption counter, which the device increments for each new transaction. This field is in binary format.

enc_PAN_length

Direction: Input
Type: Integer

Specifies the number of PAN digits in the **enc_PAN** parameter. This value must either be 0 or in the range 15 - 19, inclusive on output.

enc_PAN

Direction: Output
Type: String

This parameter returns the enciphered primary account number. For VFPE mode, if the PAN contains an odd number of 4-bit digits, the data is left-justified in the PAN variable and the right-most 4 bits can be ignored.

enc_cardholder_name_length

Direction: Input/Output
Type: Integer

Specifies the length of the **enc_cardholder_name** parameter in bytes. This output value is either 0 or in the range 2 - 32 for VFPE. For CBC mode, the output value is in the range 16 - 40 and a multiple of 8 if the service is successful and cardholder name data is enciphered. The parameter can be larger on input. However, on output, this length is updated to indicate the actual number of bytes returned by the service.

enc_cardholder_name

Direction: Output
Type: String

The field where the enciphered cardholder full name is returned.

enc_dtrack1_data_length

Direction: Input/Output
Type: Integer

Specifies the length of the **enc_dtrack1_data** parameter in bytes. The output value is either 0 or in the range 1 - 56 for mode VFPE. For mode CBC, the output value is in the range16 - 64 and a multiple of 8 if the service is successful and the track 1 discretionary data is enciphered. The parameter can be larger on input. However, on output, this length is updated to indicate the actual number of bytes returned by the service.

enc_dtrack1_data

Direction: Output
Type: String

This parameter returns the enciphered discretionary track 1 data.

enc_dtrack2_data_length

Direction: Input/Output
Type: Integer

Specifies the length of the **enc_dtrack2_data** parameter in bytes. The output

value is either 0 or in the range 1 - 19 for mode VFPE. For mode CBC, the output value is 8 or 16 if the service is successful and the data is enciphered. The parameter can be larger on input. However, on output, this length is updated to indicate the actual number of bytes returned by the service.

enc_dtrack2_data

Direction: Output
Type: String

This parameter returns the enciphered discretionary track 2 data.

DUKPT_PIN_key_identifier_length

Direction: Input/Output
Type: Integer

Specifies the length of the **DUKPT_PIN_key_identifier** parameter in bytes. If the PINKEY rule-array keyword is specified, set this value to 64. Otherwise, set this value to 0. On output, the variable is updated with the length of the data returned in the **DUKPT_PIN_key_identifier** variable.

DUKPT_PIN_key_identifier

Direction: Input/Output
Type: String

On input, this parameter must contain a DES OPINENC or IPINENC skeleton token. On output, it contains the DES token with the derived DES OPINENC or IPINENC key.

reserved1_length

Direction: Input
Type: Integer

Specifies the length of the **reserved1** parameter in bytes. The value must be 0.

reserved1

Direction: Input
Type: String

This parameter is ignored.

reserved2_length

Direction: Input
Type: Integer

Length of the **reserved2** parameter in bytes. The value must be 0.

reserved2

Direction: Input
Type: String

This parameter is ignored.

Restrictions

The restrictions for CSNBFPEE.

None.

Required commands

The required commands for CSNBFPEE.

This verb requires the **FPE Encrypt** command (offset X'02CF') to be enabled in the active role. This ACP is ON by default in z/OS. If DUKPT is specified, the verb also requires the **UKPT - PIN Verify, PIN Translate** command (offset X'00E1') to be enabled if you employ DUKPT processing.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBFPEE.

See "Usage notes for FPE Encipher (CSNBFPEE) and FPE Decipher (CSNBFPED) services" on page 457.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBFPEEJ .

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBFPEEJ(
           hikmNativeInteger   return_code,
           hikmNativeInteger   reason_code,
           hikmNativeInteger   exit_data_length,
           byte[]              exit_data,
           hikmNativeInteger   rule_array_count,
           byte[]              rule_array,
           hikmNativeInteger   clear_PAN_length,
           byte[]              clear_PAN,
           hikmNativeInteger   clear_cardholder_name_length,
           byte[]              clear_cardholder_name,
           hikmNativeInteger   clear_dtrack1_data_length,
           byte[]              clear_dtrack1_data,
           hikmNativeInteger   clear_dtrack2_data_length,
           byte[]              clear_dtrack2_data,
           hikmNativeInteger   key_identifier_length,
           byte[]              key_identifier,
           hikmNativeInteger   derivation_data_length,
           byte[]              derivation_data,
           hikmNativeInteger   enc_PAN_length,
           byte[]              enc_PAN,
           hikmNativeInteger   enc_cardholder_name_length,
           byte[]              enc_cardholder_name,
           hikmNativeInteger   enc_dtrack1_data_length,
           byte[]              enc_dtrack1_data,
           hikmNativeInteger   enc_dtrack2_data_length,
           byte[]              enc_dtrack2_data,
           hikmNativeInteger   DUKPT_PIN_key_identifier_length,
           byte[]              DUKPT_PIN_key_identifier,
           hikmNativeInteger   reserved1_length,
           byte[]              reserved1,
           hikmNativeInteger   reserved2_length,
           byte[]              reserved2);
```

FPE Translate (CSNBFPET)

The FPE Translate verb is used to translate payment data from encryption under one key to encryption under another key with a possibly different format.

You should avoid having plain text payment data in your environment. Translations can be performed with data that has been encrypted using the standard encryption option or with data that has been encrypted using the VFPE option. However, the target translation uses double length static TDES keys and the standard encryption option.

This service can be used to translate one or all of the following fields: the primary account number (PAN), the cardholder name, the track 1 discretionary data, or the track 2 discretionary data.

The following translation options are supported:

1. Translate standard option with CBC mode TDES and DUKPT keys.
2. Translate VFPE option with VFPE mode TDES andDUKPT keys.
3. Translate standard option with CBC mode TDES and static TDES keys.

To use this service, you must specify the following:

- the processing method, which is limited to Visa Data Secure Platform (VDSP)
- the key management method, either STATIC or DUKPT
- the algorithm, which is limited to TDES
- the mode, either CBC or Visa Format Preserving Encryption (VFPE) for the inbound data
- the ciphertext to be translated
- the character set of each field to be translated using rule-array keywords
- the base derivation key and key serial number if DUKPT key management is used, or a double-length TDES key if standard key management is used to recover the plain text
- the double length static TDES key used to re-encrypt the data
- Optionally, a check digit compliance indicator if VFPE is specified.

The service returns the translated fields and optionally, the DUKPT PIN encryption key, if the DUKPT key management is selected and the PINKEY rule is specified.

Format

The format of CSNBFPET.

```
CSNBFPET(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             input_PAN_length,
             input_PAN,
             input_cardholder_name_length,
             input_cardholder_name,
             input_dtrack1_data_length,
             input_dtrack1_data,
             input_dtrack2_data_length,
             input_dtrack2_data,
             input_key_identifier_length,
             input_key_identifier,
             output_key_identifier_length,
             output_key_identifier,
             derivation_data_length,
             derivation_data,
             output_PAN_length,
             output_PAN,
             output_cardholder_name_length,
             output_cardholder_name,
             output_dtrack1_data_length,
             output_dtrack1_data,
             output_dtrack2_data_length,
             output_dtrack2_data,
             DUKPT_PIN_key_identifier_length,
             DUKPT_PIN_key_identifier,
             reserved1_length,
             reserved1,
             reserved2_length,
             reserved2)
```

Parameters

The parameters for CSNBFPET.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. The minimum value is 4. The maximum value is 10.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. The **rule_array** keywords are described in Table 155 on page 526.

Note: At least one character set keyword is required.

Table 155. Keywords for FPE Translate control information

Keyword	Description
Processing method (required)	
VMDS	Specifies that the VDSP method (Visa Data Secure Platform method, formally known as the Visa Merchant Data Secure (VMDS) method) is to be used for processing.
Key management method (one required)	
STATIC	Specifies the use of double length (2-key) triple-DES symmetric keys. This is a non-DUKPT key.
DUKPT	Specifies the use of the transaction unique general purpose Data Encryption Keys generated by the DUKPT process at the point of service for data encryption. This is required if VFPE mode is specified. Otherwise, this is optional.
Algorithm (required)	
TDES	Specifies the use of CBC mode triple-DES encryption.
Mode (one required)	
CBC	Specifies the use of CBC mode. This is the mode for the standard encryption option.
VFPE	Specifies the use of Visa format preserving encryption.
PAN input output character set (one required if the **clear_PAN_length** variable is greater than 0. Otherwise, it is not allowed.)	
PAN8BITA	Specifies that the PAN data character set is ASCII represented in binary form. Valid only for VFPE mode.
PAN4BITX	Specifies that the PAN data character set is 4-bit hex with two digits per byte. Valid only for VFPE mode.
PAN-EBLK	Specifies that the PAN data is in a CBC encrypted block. Valid only for CBC mode.
Cardholder name input output character set (required if the **clear_cardholder_name_length** variable is greater than 0.)	
CN8BITA	Specifies that the cardholder name character set is ASCII represented in binary format, one character per byte. See Table 130 on page 456 for valid characters. Valid only for VFPE mode.
CN-EBLK	Specifies that the cardholder name data is in a CBC-encrypted block.
Track_1 input output character set (required if the **clear_dtrack1_data_length** variable is greater than 0. Otherwise, it is not valid.)	
TK18BITA	Specifies that the track 1 discretionary data character set is ASCII represented in binary format, one character per byte. See Table 130 on page 456 for valid characters.
TK1-EBLK	Specifies that the track 1 discretionary data is in a CBC-encrypted block. Valid only for CBC mode.
Track_2 input output character set (required if the **clear_dtrack2_data_length** variable is greater than 0. Otherwise, it is not valid.)	
TK28BITA	Specifies that the track 2 discretionary data character set is ASCII represented in binary format. Valid only for VFPE mode.
TK2-EBLK	Specifies that the track 2 discretionary data is in a CBC encrypted block. Valid only for CBC mode.
PIN encryption key output selection (one, optional, if DUKPT is specified. Otherwise, it is not valid.)	
NOPINKEY	Do not return a DUKPT PIN encryption key. This is the default.
PINKEY	Return a DUKPT PIN encryption key.
PAN check digit compliance (one required if mode VFPE and the PAN input character set keyword is present. Otherwise, it is not allowed.)	
CMPCKDGT	Last digit of the PAN contains a compliant check digit per ISO/IEC 7812-1.
NONCKDGT	Last digit of the PAN does not contain a compliant check digit per ISO/IEC 7812-1.

input_PAN_length

Direction: Input
Type: Integer

Specifies the length of the **input_PAN** parameter in bytes if the mode is CBC. Specifies the number of PAN digits if the mode is VFPE. The value is 0 if PAN data has not been presented for translation. Otherwise, the value is in the range 15 - 19 for VFPE. The value must be 16 if the standard option with CBC mode is selected.

input_PAN

Direction: Input
Type: String

The enciphered primary account number (PAN) that is to be translated. For VFPE mode, if the PAN contains an odd number of 4-bit digits, the data is left justified in the PAN variable and the right-most 4 bits are ignored.

input_cardholder_name_length

Direction: Input
Type: Integer

Specifies the length of the **input_cardholder_name** parameter in bytes. This value must be 0 if cardholder name is not presented for translation. Otherwise, the value is in the range 1 - 32 for VFPE. For CBC mode, the input value is either 16, 24, 32, or 40.

input_cardholder_name

Direction: Input
Type: String

The enciphered cardholder full name that is to be translated. Only characters in Table 130 on page 456 are valid.

input_dtrack1_data_length

Direction: Input
Type: Integer

Specifies the length of the **input_dtrack1_data** parameter in bytes. This value must be 0 if track 1 discretionary data is not presented for translation. Otherwise, the value is in the range 1 - 56 for VFPE. For CBC mode, the input value is either 16, 24, 32, 40, 48, 56, or 64.

input_dtrack1_data

Direction: Input
Type: String

The encrypted track 1 data that is to be translated. Only characters in Table 130 on page 456 are valid.

input_dtrack2_data_length

Direction: Input
Type: Integer

Specifies the length of the **input_dtrack2_data** parameter in bytes. This value must be 0 if track 2 discretionary data is not presented for translation. Otherwise, the value is in the range 1 - 19 for VFPE. For CBC mode, the input value is either 8 or 16.

input_dtrack2_data

Direction: Input
Type: String

The encrypted track 2 data that is to be translated.

input_key_identifier_length

Direction: Input
Type: Integer

Specifies the length of the **input_key_identifier** parameter in bytes. The value must be 64, because only fixed length DES tokens are supported as the key identifier.

input_key_identifier

Direction: Input/Output
Type: String

The identifier of the key that is used to either decrypt the card data (key management STATIC) or derive the **DUKPT_PIN_key_identifer** (key management DUKPT). The *key identifier* is an operational token or the key label of an operational token in key storage.

For key management DUKPT, the key type must be KEYGENKY. In addition, it must have a control vector with bit 18 equal to B'1' (UKPT). The base derivation key is the one from which the operational keys are derived using the DUKPT algorithm defined in ANS X9.24 Part 1.

For key management STATIC, (Zone Encryption Key in the VDSP specification), the key type must be either CIPHER or ENCIPHER. For production purposes, it is recommended that the key have left and right halves that are not equal.

Note: Data keys are not supported.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

output_key_identifier_length

Direction: Input
Type: Integer

Specifies the length of the **output_key_identifier** parameter in bytes. The value must be 64, because only fixed length DES tokens are supported as the key identifier.

output_key_identifier

Direction: Input/Output
Type: String

The identifier of the key that is used to decrypt the output card data. The *key identifier* is an operational token or the key label of an operational token in key storage.

The key type must be either CIPHER or ENCIPHER. For production purposes, it is recommended that the key have left and right halves that are not equal.

Note: Data keys are not supported.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

derivation_data_length

Direction: Input
Type: Integer

Specifies the length of the **derivation_data** parameter in bytes. The value must be 10 if a DUKPT key is specified in the **key_identifier** parameter. If a data encryption key is specified in the **key_identifier** parameter, this value must be set to zero.

derivation_data

Direction: Input
Type: String

Contains the 80 bit (10 byte) derivation data that is used as input to the DUKPT derivation process. The derivation data contains the current key serial number (CKSN), which is composed of the 59 bit initial key serial number concatenated with the 21 bit value of the current encryption counter, which the device increments for each new transaction. This field is in binary format.

output_PAN_length

Direction: Input/Output
Type: Integer

Specifies the number of bytes of data in the **output_PAN** parameter. This value is 0 or 16 on output.

output_PAN

Direction: Output
Type: String

This parameter returns the translated primary account number with which the PIN is associated. The full account number, including check digit, is translated. The data for this parameter is returned as TDES-encrypted data in binary format. The 16 byte output is left justified in this field.

output_cardholder_name_length

Direction: Input/Output
Type: Integer

Specifies the length of the **output_cardholder_name** parameter in bytes. This output value is either 0 or 16, 24, 32, or 40 bytes on output. The variable can be larger on input. However, on output, this field is updated to indicate the actual number of bytes returned by the card.

output_cardholder_name

Direction: Output
Type: String

This parameter returns the translated cardholder full name. The data for this parameter is returned as TDES-encrypted data in binary format.

output_dtrack1_data_length

Direction: Input/Output
Type: Integer

Specifies the length of the **output_dtrack1_data** parameter in bytes. The output value is either 0 or 16, 24, 32, 40, 48, 56, or 64 bytes. The value can be larger on input. However, on output, this field is updated to indicate the actual number of bytes returned by the service.

output_dtrack1_data

Direction: Output
Type: String

This parameter returns the translated discretionary track 1 data. This is the discretionary data from track 1 of a magnetic stripe card. The data for this parameter is returned as TDES-encrypted data in binary format.

output_dtrack2_data_length

Direction: Input/Output
Type: Integer

Specifies the length of the **output_dtrack2_data** parameter in bytes. The output value is either 0, 8, or 16. The value can be larger on input. However, on output, this field is updated to indicate the actual number of bytes returned by the service.

output_dtrack2_data

Direction: Output
Type: String

This parameter returns the translated discretionary track 2 data. This is the discretionary data from track 2 of a magnetic stripe card. The data for this parameter is returned as TDES-encrypted data in binary format.

DUKPT_PIN_key_identifier_length

Direction: Input/Output
Type: Integer

Specifies the length of the **DUKPT_PIN_key_identifier** parameter in bytes. If the PINKEY rule-array keyword is specified, set this value to 64. Otherwise, set this value to 0. On output, the variable is updated with the length of the data returned in the **DUKPT_PIN_key_identifier** variable.

DUKPT_PIN_key_identifier

Direction: Input/Output
Type: String

On input, this parameter must contain a DES OPINENC or IPINENC skeleton token. On output, it contains the DES token with the derived DES OPINENC or IPINENC key.

reserved1_length

Direction: Input
Type: Integer

Specifies the length of the **reserved1** parameter in bytes. The value must be 0.

reserved1

Direction: Input
Type: String

This parameter is ignored.

reserved2_length

Direction: Input
Type: Integer

Specifies the length of the **reserved2** parameter in bytes. The value must be 0.

reserved2

Direction: Input
Type: String

This parameter is ignored.

Restrictions

The restrictions for CSNBFPET.

None.

Required commands

The required commands for CSNBFPET.

This verb requires the **FPE Translate** command (offset X'02D1') to be enabled in the active role. This ACP is ON by default in z/OS. If DUKPT is specified, the verb also requires the **UKPT - PIN Verify, PIN Translate** command (offset X'00E1') to be enabled if you employ DUKPT processing.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBFPET.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBFPETJ .

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBFPETJ(
           hikmNativeInteger   return_code,
           hikmNativeInteger   reason_code,
           hikmNativeInteger   exit_data_length,
           byte[]              exit_data,
           hikmNativeInteger   rule_array_count,
           byte[]              rule_array,
           hikmNativeInteger   input_PAN_length,
           byte[]              input_PAN,
           hikmNativeInteger   input_cardholder_name_length,
           byte[]              input_cardholder_name,
           hikmNativeInteger   input_dtrack1_data_length,
           byte[]              input_dtrack1_data,
           hikmNativeInteger   input_dtrack2_data_length,
```

```
byte[]              input_dtrack2_data,
hikmNativeInteger   input_key_identifier_length,
byte[]              input_key_identifier,
hikmNativeInteger   output_key_identifier_length,
byte[]              output_key_identifier,
hikmNativeInteger   derivation_data_length,
byte[]              derivation_data,
hikmNativeInteger   output_PAN_length,
byte[]              output_PAN,
hikmNativeInteger   output_cardholder_name_length,
byte[]              output_cardholder_name,
hikmNativeInteger   output_dtrack1_data_length,
byte[]              output_dtrack1_data,
hikmNativeInteger   output_dtrack2_data_length,
byte[]              output_dtrack2_data,
hikmNativeInteger   DUKPT_PIN_key_identifier_length,
byte[]              DUKPT_PIN_key_identifier,
hikmNativeInteger   reserved1_length,
byte[]              reserved1,
hikmNativeInteger   reserved2_length,
byte[]              reserved2);
```

PIN Change/Unblock (CSNBPCU)

The PIN Change/Unblock verb is used to generate a special PIN block to change the PIN accepted by an integrated circuit card (smartcard).

The PIN Change/Unblock verb prepares an encrypted message-portion for communicating an original or replacement PIN for an EMV smart card. The verb embeds the PINs in an encrypted PIN-block from information that you supply. You incorporate the information created with the verb in a message sent to the smart card.

The processing is consistent with the specifications provided in these documents:

u

- *EMV 2000 Integrated Circuit Card Specifications for Payment Systems Version 4.0 (EMV4.0)*
- *Visa Integrated Circuit Card Specification Manual, Version 1.4.0*
- *Integrated Circuit Card Specification (VIS) 1.4.0 Corrections*

You specify the following information:

- Through the optional choice of one rule-array keyword, the key-diversification process to employ in deriving the session key used to encrypt the PIN block. See “Working with Europay-Mastercard-Visa Smart cards” on page 444 for processing details.

  TDES-XOR
  An exclusive-OR process described in “Deriving the CCA TDES-XOR session key” on page 534. It is the default.

  TDESEMV2
  The tree-based-diversification process with a branch factor of 2.

  TDESEMV4
  The tree-based-diversification process with a branch factor of 4.
- Through the required choice of one rule-array keyword, if you are providing a PIN for a smart card:

AMEXPCU1
For a card with a current PIN, provide the existing PIN in an encrypted PIN-block in the **current_reference_PIN_block** variable, and supply the new PIN-value in an encrypted PIN-block in the **new_reference_PIN_block** variable.

AMEXPCU2
For a card without a PIN, provide the new PIN in an encrypted PIN-block in the **new_reference_PIN_block** variable. The contents of the five current_reference_PIN_x variables are ignored.

VISAPCU1
For a card without a PIN, provide the new PIN in an encrypted PIN-block in the new_reference_PIN_block variable. The contents of the five **current_reference_PIN_x** variables are ignored.

VISAPCU2
For a card with a current PIN, provide the existing PIN in an encrypted PIN-block in the **current_reference_PIN_block** variable, and supply the new PIN-value in an encrypted PIN-block in the **new_reference_PIN_block** variable.

- Issuer-provided master-derivation keys (MDK). The card-issuer provides two keys for diversifying the same data:
  - The MAC-MDK key that you incorporate in the variable specified by the authentication_key_identifier parameter. The verb uses this key to derive an authentication value incorporated in the PIN block. The control vector for the MAC-MDK key must specify a DKYGENKY key type with DKYL0 (level-0), and DMAC or DALL permissions.
  - The ENC-MDK key that you incorporate in the variable specified by the encryption_key_identifier parameter. The verb uses this key to derive the PIN-block encryption key. The control vector for the ENC-MDK key must specify a DKYGENKY key type with DKYL0 (level-0), and DMPIN or DALL permissions.
- The diversification_data_length to indicate the sum of the lengths of:
  - Data, 8 or 16 bytes, encrypted by the verb using the MDK keys to generate the ENC-UDK and Unique DES Key.
  - The 2-byte application transaction counter (ATC). You receive the ATC value from the EMV smart card.
  - The optional 16-byte initial value used in the TDESEMVn processes.

  Valid lengths are 10, 18, 26, and 34 bytes.
- The diversification_data variable. Concatenate the 8-byte or 16-byte data, the ATC, and optionally the Initial Value.

  The 16-bit ATC counter is processed as a two-byte string, not as an integer value.
- The new-reference PIN in an encrypted PIN block. You provide:
  - the key to decrypt the PIN block
  - the PIN block
  - the format information that defines how to parse the PIN block
  - when using an ISO-0 or ISO-3 (Release 3.30 or later) PIN-block format, primary account number (PAN) information to enable PIN recovery from the ISO-0 or ISO-3 PIN-block format.
- If you specified VISAPCU2 (because the target smart card already has a PIN), the current_reference_PIN in an encrypted PIN block with the associated

decrypting key, PIN-block format, and PAN data. In any case, you must declare the five **current_reference_PIN_x** variables.

- The output_PIN_message variable to receive the encrypted PIN block for the smart card, and the length in bytes of the PIN block (16). The PIN-block format you specify (VISAPCU1 or VISAPCU2) corresponds to the one or two PIN values to be communicated to the smart card.
- Two variables which are reserved for future use: output_PIN_data_length (valued to zero), and an output_PIN_data string variable (or set the associated parameter to a null pointer).

The PIN Change/Unblock verb:

- Decrypts the MDK keys and verifies the required control vector permissions.
- Diversifies the left-most eight bytes of data using the MAC-MDK key to obtain the authentication value for placement into the PIN block.
- Recovers the supplied PIN values provided that PIN-block encrypting keys are one of IPINENC or OPINENC type, and the use of the specific type is authorized with the appropriate access-control command.
- Constructs and pads the output PIN block to a 16-byte string.
- Generates the session key used to encrypt the output-PIN block using the ENC-MDK, the key_generation_data, the ATC counter value, and the optional Initial Value.
- Triple encrypts the 16-byte padded PIN-block in ECB mode.
- Returns the encrypted, padded PIN-block in the **output_PIN_message** variable.

The generating **DKYGENKY** cannot have replicated halves. The *encryption_master_key* is a **DKYGENKY** that permits generation of a SMPIN key. The *authentication_master_key* is also a DKYGENKY that permits generation of a double length MAC key.

Deriving the CCA TDES-XOR session key

In the Diversified_Key_Generate and PIN_Change/Unblock verbs, the TDES-XOR process first derives a smart-card-specific intermediate key from the issuer-supplied ENC-MDK key and card-specific data. (This intermediate key is also used in the TDESEMV2 and TDESEMV4 processes. See the next section.) The intermediate key is then modified using the application transaction counter (ATC) value supplied by the smart card.

The double-length session-key creation steps:

1. 2. 3. 4.

1. Obtain the left-half of an intermediate key by ECB-mode Triple-DES encrypting the (first) eight bytes of card specific data using the issuer-supplied ENC-MDK key.
2. Again using the ENC-MDK key, obtain the right-half of the intermediate key by ECB-mode Triple-DES encrypting with one of these methods:
   - The second 8 bytes of card-specific derivation data when 16 bytes have been supplied.
   - The exclusive-OR of the supplied 8 bytes of derivation data with X'FFFFFFFF FFFFFFFF'.

3. Pad the ATC value to the left with six bytes of X'00' and exclusive-OR the result with the left-half of the intermediate key to obtain the left-half of the session key.
4. Obtain the one's complement of the ATC by exclusive-ORing the ATC with X'FFFF'. Pad the result on the left with six bytes of X'00'. Exclusive-OR the 8-byte result with the right-half of the intermediate key to obtain the right-half of the session key.

Format

The format of CSNBPCU.

```
CSNBPCU(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              rule_array_count,
              rule_array,
              authentication_master_key_length,
              authentication_master_key,
              encryption_master_key_length,
              encryption_master_key,
              key_generation_data_length,
              key_generation_data,
              new_reference_PIN_key_length,
              new_reference_PIN_key,
              new_reference_PIN_block,
              new_reference_PIN_profile,
              new_reference_PAN_data,
              current_reference_PIN_key_length,
              current_reference_PIN_key,
              current_reference_PIN_block,
              current_reference_PIN_profile,
              current_reference_PAN__data,
              output_PIN_data_length,
              output_PIN_data,
              output_PIN_profile,
              output_PIN_message_length,
              output_PIN_message)
```

Parameters

The parameters for CSNBPCU.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1 or 2.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. The keywords are left-aligned in an 8-byte field and padded on the right with blanks. The

keywords must be in contiguous storage. The *rule_array* keywords are described in Table 156.

Table 156. Keywords for PIN Change/Unblock control information

Keyword	Description
Algorithm (One, optional)	
TDES-XOR	TDES encipher clear data to generate the intermediate (card-unique) key, followed by XOR of the final two bytes of each key with the ATC counter. This is the default.
TDESEMV2	Same processing as in the Diversified Key Generate verb.
TDESEMV4	Same processing as in the Diversified Key Generate verb.
PIN processing method (One, required)	
AMEXPCU1	Form the new PIN from the new reference PIN, the smart-card-unique, intermediate key, and the current reference PIN.
AMEXPCU2	Form the new PIN from the new reference PIN and the smart-card-unique, intermediate key.
VISAPCU1	Form the new PIN from the new reference PIN and the intermediate (card-unique) key only.
VISAPCU2	Form the new PIN from the new reference PIN, the intermediate (card-unique) key and the current reference PIN.

authentication_master_key_length

Direction: Input
Type: Integer

The length of the parameter. Currently, the value must be 64.

authentication_master_key

Direction: Input/Output
Type: String

The label name or internal token of a **DKYGENKY** key type that is to be used to generate the card-unique diversified key. The control vector of this key must be a **DKYL0** key that permits the generation of a double-length **MAC** key (**DMAC**). This **DKYGENKY** might not have replicated key halves.

encryption_master_key_length

Direction: Input
Type: Integer

The length of the **encryption_master_key** parameter. Currently, the value must be 64.

encryption_master_key

Direction: Input/Output
Type: String

The label name or internal token of a **DKYGENKY** key type that is to be used to generate the card-unique diversified key and the secure messaging session key for the protection of the output PIN block. The control vector of this key must be a **DKYL0** key that permits the generation of a SMPIN key type. This **DKYGENKY** might not have replicated key halves.

key_generation_data_length

Direction: Input
Type: Integer

The length of the **key_generation_data** parameter. This value must be 10, 18, 26, or 34 bytes.

key_generation_data

Direction: Input
Type: String

The data provided to generate the card-unique session key. For **TDES-XOR**, this consists of 8 or 16 bytes of data to be processed by TDES to generate the card-unique diversified key followed by a 16-bit ATC counter to offset the card-unique diversified key to form the session key. For **TDESEMV2** and **TDESEMV4**, this can be 10, 18, 26, or 34 bytes. See "Diversified Key Generate (CSNBDKG)" on page 145 for more information.

new_reference_PIN_key_length

Direction: Input
Type: Integer

The length of the **new_reference_PIN_key** parameter. Currently, the value must be 64.

new_reference_PIN_key

Direction: Input/Output
Type: String

The label name or internal token of a PIN encrypting key that is to be used to decrypt the *new_reference_PIN_block*. This must be an **IPINENC** or **OPINENC** key. If the label name is supplied, the name must be unique in the DES key storage file.

new_reference_PIN_block

Direction: Input
Type: String

This is an 8-byte field that contains the enciphered PIN block of the new PIN.

new_reference_PIN_profile

Direction: Input
Type: String

This is a 24-byte field that contains three 8-byte elements with a PIN block format keyword, a format control keyword (NONE), and a pad digit as required by certain formats.

new_reference_PAN_data

Direction: Input
Type: String

This is a 12-byte field containing the PAN in character format. This data might be needed to recover the new reference PIN if the format is **ISO-0**, **ISO-3**, or **VISA-4**. If neither is used, this parameter might be blanks.

For **ISO-0** or **ISO-3**, use the rightmost 12 digits of the PAN, excluding the check digit. For **VISA-4**, use the leftmost 12 digits of the PAN, excluding the check digit.

current_reference_PIN_key_length

Direction: Input
Type: Integer

The length of the **current_reference_PIN_key** parameter. For the current implementation, the value must be 64. If the *rule_array* contains **VISAPCU1**, this value must be 0.

current_reference_PIN_key

Direction: Input/Output
Type: String

The label name or internal token of a PIN encrypting key that is to be used to decrypt the **current_reference_PIN_block**. This must be an **IPINENC** or **OPINENC** key. If the label name is supplied, the name must be unique in the key storage. If the *rule_array* contains **VISAPCU1**, this value is ignored.

current_reference_PIN_block

Direction: Input
Type: String

This is an 8-byte field that contains the enciphered PIN block of the new PIN. If the *rule_array* contains **VISAPCU1**, this value is ignored.

current_reference_PIN_profile

Direction: Input
Type: String

This is a 24-byte field that contains three 8-byte elements with a PIN block format keyword, a format control keyword (NONE), and a pad digit as required by certain formats. If the **rule_array** contains **VISAPCU1**, this value is ignored.

current_reference_PAN_data

Direction: Input
Type: String

This is a 12-byte field containing PAN in character format. This data might be needed to recover the new reference PIN if the format is **ISO-0**, **ISO-3**, or **VISA-4**. If neither is used, this parameter might be blanks. If the *rule_array* contains **VISAPCU1**, this value is ignored.

For **ISO-0** or **ISO-3**, use the rightmost 12 digits of the PAN, excluding the check digit. For **VISA-4**, use the leftmost 12 digits of the PAN, excluding the check digit.

output_PIN_data_length

Direction: Input
Type: Integer

Currently this field is reserved. This value must be 0.

output_PIN_data

Direction: Input
Type: String

This parameter is ignored.

output_PIN_profile

Direction: Input
Type: String

This is a 24-byte field that contains three 8-byte elements with a PIN block format keyword (**VISAPCU1** or **VISCPU2**), a format control keyword (NONE), and eight bytes of spaces.

output_PIN_message_length

Direction: Input/Output
Type: Integer

The length of the **output_PIN_message** field. Currently the value must be a minimum of 16.

output_PIN_message

Direction: Output
Type: String

The reformatted PIN block with the new reference PIN enciphered under the SMPIN session key.

Restrictions

The restrictions for CSNBPCU.

None.

Required commands

The required commands for CSNBPCU.

This verb requires the following commands to be enabled in the active role based on the permissible key-type, **IPINENC** or **OPINENC**, used in the decryption of the input PIN blocks.

PIN-block encrypting key-type	Offset	Command	Comment
OPINENC	X'00BC'	**PIN Change/Unblock - change EMV PIN with OPINENC**	Required if either the *new_reference_PIN_key* or the *current_reference_PIN_key* are permitted to be an **OPINENC** key type.
IPINENC	X'00BD'	**PIN Change/Unblock - change EMV PIN with IPINENC**	Required if either the *new_reference_PIN_key* or the *current_reference_PIN_key* are permitted to be an **IPINENC** key type.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

When a MAC-MDK or an ENC-MDK of key type **DKYGENKY** is specified with control vector bits (19 - 22) of B'1111', the **Diversified Key Generate - DKYGENKY - DALL** command (offset X'0290') must also be enabled in the active role.

Note: A role with offset X'0290' enabled can also use the Diversified Key Generate verb with a **DALL** key.

An enhanced PIN security mode is available for extracting PINs from a 3621 or 3624 encrypted PIN-block using the **PADDIGIT** PIN-extraction method. This mode limits checking of the PIN to decimal digits, and a minimum PIN length of 4 is enforced; no other PIN-block consistency checking will occur. To activate this mode, enable the **Enhanced PIN Security** command (offset X'0313') in the active role.

Usage notes

The usage notes for CSNBPCU.

There are additional access points for this verb.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBPCUJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBPCUJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger authentication_master_key_length,
           byte[]            authentication_master_key,
           hikmNativeInteger encryption_master_key_length,
           byte[]            encryption_master_key,
           hikmNativeInteger key_generation_data_length,
           byte[]            key_generation_data,
           hikmNativeInteger new_reference_PIN_key_length,
           byte[]            new_reference_PIN_key,
           byte[]            nnew_reference_PIN_block,
           byte[]            new_reference_PIN_profile,
           byte[]            new_reference_PAN_data,
           hikmNativeInteger current_reference_PIN_key_length,
           byte[]            current_reference_PIN_key,
           byte[]            current_reference_PIN_block,
           byte[]            current_reference_PIN_profile,
           byte[]            current_reference_PAN__data,
           hikmNativeInteger output_PIN_data_length,
           byte[]            output_PIN_data,
           byte[]            output_PIN_profile,
           hikmNativeInteger output_PIN_message_length,
           byte[]            output_PIN_message);
```

Recover PIN from Offset (CSNBPFO)

Use the Recover PIN from Offset verb to calculate the encrypted customer-entered PIN from a PIN generating key, account information, and an IBM-PINO Offset.

The customer-entered PIN is returned in a PIN block formatted to the specifications of the **PIN_profile** and **PAN_data** parameters, and encrypted with the key supplied in parameter **PIN_encryption_key_identifier** as described in "Parameters" on page 541.

Format

The format of CSNBPFO.

```
CSNBPFO(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             PIN_encryption_key_identifier_length,
             PIN_encryption_key_identifier,
             PIN_generation_key_identifier_length,
             PIN_generation_key_identifier,
             PIN_profile,
             PAN_data,
             offset,
             reserved_1,
             data_array,
             encrypted_PIN_block_length,
             encrypted_PIN_block)
```

Parameters

The parameters for CSNBFPET.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. This value must be 0.

rule_array

Direction: Input
Type: String array

There are no keywords for this service. Therefore, this parameter is ignored.

PIN_encryption_key_identifier_length

Direction: Input
Type: Integer

Length of the **PIN_encryption_key_identifier** parameter in bytes. This value must be 64.

PIN_encryption_key_identifier

Direction: Input
Type: String

An internal key token or the label of the AES or DES key storage record containing an OPINENC key that is used to encrypt the **returned_encrypted_PIN_block**.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

PIN_generation_key_identifier_length

Direction: Input
Type: Integer

Length of the **PIN_generation_key_identifier** parameter in bytes. This value must be 64.

PIN_generation_key_identifier

Direction: Input
Type: String

An internal key token or the label of the AES or DES key storage record containing a PINGEN key that is used to generate the bank reference PIN.

If the token supplied was encrypted under the old master key, the token will be returned encrypted under the current master key.

PIN_profile

Direction: Input
Type: String array

This parameter consists of three 8-byte character elements that contain information necessary to format a PIN block. The pad digit is needed to format an IBM 3624 or 3621 PIN block. The format control constant must be NONE. The first element of the **PIN_profile** (PIN Block Format) determines the format of the output PIN block.

PAN_data

Direction: Input
Type: String

A 12-byte personal account number (PAN) in character format. The PAN is used in formatting the PIN block if the PIN profile specifies ISO-0, ISO-3, or VISA-4 block formats. Otherwise, ensure that this parameter is a 12-byte variable in application storage. The information in this variable is ignored, but the variable must be specified.

offset

Direction: Input
Type: String

A 16 byte area that contains the 4-byte PVV left-justified and padded with blanks. This is the value which was returned by a prior call to the **Clear PIN Generate Alternate** service (see “Clear PIN Generate Alternate (CSNBCPA)” on page 468).

data_array

Direction: Input
Type: String

A pointer to a string variable containing three 16-byte numeric character strings, which are equivalent to a single 48-byte string. The values in the **data_array** parameter depend on the keyword for the PIN-calculation method. Each element is not always used, but you must always declare a complete data array.

Array Element	Description
Decimalization_table	This element contains the decimalization table of 16 characters that are used to convert the hexadecimal digits X'0' to X'F' of the enciphered validation data to the decimal digits 0 to 9.
validation_data	This element contains 1 to 16 characters of account data, left justified and padded on the right with spaces.
reserved_field	Must be 16 bytes of blanks.

reserved_1

Direction: Input
Type: Integer

The **reserved_1** parameter must be zero.

encrypted_PIN_block_length

Direction: Input/Output
Type: Integer

Length of the **encrypted_PIN_block** parameter in bytes.

encrypted_PIN_block

Direction: Input
Type: String

This parameter is an 8-byte field that contains the encrypted customer PIN that was originally used in the **Clear PIN Generate Alternate** service (see "Clear PIN Generate Alternate (CSNBCPA)" on page 468).

Restrictions

The restrictions for CSNBPFO.

None.

Required commands

The required commands for CSNBPFO.

This verb requires the **Recover PIN From Offset** command (offset X'02B0') to be enabled in the active role.

When the **ANSI X9.8 PIN - Enforce PIN block restrictions** command (offset X'0350') is enabled in the active role, only a PIN-block format keyword of ISO-0 or ISO-3 is allowed in the input **PIN_profile** parameter. Note that offset X'0350' also affects access control of the Encrypted PIN Translate and the Secure Messaging for PINs verbs.

When the **ANSI X9.8 PIN - Use stored decimalization tables only** command (offset X'0356') is enabled in the active role, the **Decimalization_table** element of the **data_array** value must match one of the PIN decimalization tables that are in the active state on the coprocessor. Use of this command provides improved security and control for PIN decimalization tables.

An enhanced PIN security mode is available for formatting an encrypted PIN-block into IBM 3624 format. This mode limits checking of the PIN to decimal

digits; no other PIN-block consistency checking will occur. To activate this mode, enable the **Enhanced PIN Security** command (offset X'0313') in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBFPET.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBPFO .

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBPFO(
          hikmNativeInteger   return_code,
          hikmNativeInteger   reason_code,
          hikmNativeInteger   exit_data_length,
          byte[]              exit_data,
          hikmNativeInteger   rule_array_count,
          byte[]              rule_array,
          hikmNativeInteger   PIN_encryption_key_identifier_length,
          byte[]              PIN_encryption_key_identifier,
          hikmNativeInteger   PIN_generation_key_identifier_length,
          byte[]              PIN_generation_key_identifier,
          byte[]              PIN_profile,
          byte[]              PAN_data,
          byte[]              offset,
          hikmNativeInteger   reserved_1,
          byte[]              data_array,
          hikmNativeInteger   encrypted_PIN_block_length,
          byte[]              encrypted_PIN_block);
```

Secure Messaging for Keys (CSNBSKY)

The Secure Messaging for Keys verb will encrypt a text block including a clear key value decrypted from an internal or external DES token.

The text block is normally a "Value" field of a secure message TLV (Tag/Length/Value) element of a secure message. TLV is defined in ISO/IEC 7816-4.

Format

The format of CSNBSKY.

```
CSNBSKY(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              rule_array_count,
              rule_array,
              input_key_identifier,
              key_encrypting_key_identifier,
              secmsg_key_identifier,
              text_length,
              clear_text,
              initialization_vector,
              key_offset,
              key_offset_field_length,
              enciphered_text,
              output_chaining_vector)
```

Parameters

The parameters for CSNBSKY.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0 or 1.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. The processing method is the encryption mode used to encrypt the message. The *rule_array* keywords are described in Table 157.

Table 157. Keywords for Secure Messaging for Keys control information

Keyword	Description
Enciphering mode (One, optional)	
TDES-CBC	Use CBC mode to encipher the message. This is the default.
TDES-ECB	Use EBC mode to encipher the message.

input_key_identifier

Direction: Input/Output
Type: String

The internal token, external token, or key label of an internal token of a double length DES key. The key is recovered in the clear and placed in the text to be encrypted. The control vector of the DES key must not prohibit export.

key_encrypting_key_identifier

Direction: Input/Output
Type: String

If the *input_key_identifier* is an external token, this parameter is the internal token or the key label of the internal token of **IMPORTER** or **EXPORTER**. If it is not, it is a null token. If a key label is specified, the key label must be unique.

secmsg_key_identifier

Direction: Input/Output
Type: String

The internal token or key label of a secure message key for encrypting keys. This key is used to encrypt the updated *clear_text* containing the recovered DES key.

text_length

Direction: Input
Type: Integer

The length of the *clear_text* parameter. Length must be a multiple of eight. Maximum length is 4096.

clear_text

Direction: Input
Type: String

Clear text that contains the recovered DES key at the offset specified and is then encrypted. Any padding or formatting of the message must be done by the caller on input.

initialization_vector

Direction: Input
Type: String

The 8-byte supplied string for the **TDES-CBC** mode of encryption. The *initialization_vector* is XORed with the first eight bytes of *clear_text* before encryption. This field is ignored for **TDES-ECB** mode.

key_offset

Direction: Input
Type: Integer

The offset within the *clear_text* parameter at *key_offset* where the recovered clear *input_key_identifier* value is to be placed. The first byte of the *clear_text* field is offset 0.

key_offset_field_length

Direction: Input
Type: Integer

The length of the field within *clear_text* parameter at *key_offset* where the recovered clear *input_key_identifier* value is to be placed. Length must be a multiple of eight and is equal to the key length of the recovered key. The key must fit entirely within the *clear_text*.

enciphered_text

Direction: Output
Type: String

The field where the enciphered text is returned. The length of this field must be at least as long as the *clear_text* field.

output_chaining_vector

Direction: Output
Type: String

This field contains the last eight bytes of enciphered text and is used as the *initialization_vector* for the next encryption call if data needs to be chained for **TDES-CBC** mode. No data is returned for **TDES-ECB**.

Restrictions

The restrictions for CSNBSKY.

None.

Required commands

The required commands for CSNBSKY.

This verb requires the **Secure Messaging for Keys** command (offset X'0273') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBSKY.

Keys appear in the clear only within the secure boundary of the cryptographic coprocessor, and never in host storage.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBSKYJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBSKYJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           byte[]            input_key_indentifier,
           byte[]            key_encrypting_key,
           byte[]            session_key,
           hikmNativeInteger text_length,
           byte[]            clear_text,
           byte[]            initialization_vector,
           hikmNativeInteger key_offset,
```

```
hikmNativeInteger key_offset_field_length,
byte[]            cipher_text,
byte[]            output_chaining_value);
```

Secure Messaging for PINs (CSNBSPN)

The Secure Messaging for PINs verb will encrypt a text block including a clear PIN block recovered from an encrypted PIN block.

The input PIN block will be reformatted if the block format in the *input_PIN_profile* is different from the block format in the *output_PIN_profile*. The clear PIN block will only be self encrypted if the **SELFENC** keyword is specified in the *rule_array*. The text block is normally a "Value" field of a secure message TLV (Tag/Length/Value) element of a secure message. TLV is defined in ISO/IEC 7816-4.

Format

The format of CSNBSPN.

```
CSNBSPN(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             input_PIN_block,
             PIN_encrypting_key_identifier,
             input_PIN_profile,
             input_PAN_data,
             secmsg_key_identifier,
             output_PIN_profile,
             output_PAN_data,
             text_length,
             clear_text,
             initialization_vector,
             PIN_offset,
             PIN_offset_field_length,
             enciphered_text,
             output_chaining_vector)
```

Parameters

The parameters for CSNBSPN.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters common to all verbs” on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0, 1, or 2.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. The processing method is the algorithm used to create the generated key. The keywords are left-aligned and padded on the right with blanks. The *rule_array* keywords are described in Table 158.

Table 158. Keywords for Secure Messaging for PINs control information

Keyword	Description
Enciphering mode (One, optional)	
TDES-CBC	Use CBC mode to encipher the message. This is the default.
TDES-ECB	Use EBC mode to encipher the message.
PIN encryption (One, optional)	
CLEARPIN	Recovered clear input PIN block (might be reformatted) is placed in the clear in the message for encryption with the secure message key. This is the default.
SELFENC	Recovered clear input PIN block (might be reformatted) is self-encrypted and then placed in the message for encryption with the secure message key.

input_PIN_block

Direction: Input
Type: String

The 8-byte input PIN block that is to be recovered in the clear and, perhaps, reformatted and then placed in the *clear_text* to be encrypted.

PIN_encrypting_key_identifier

Direction: Input/Output
Type: String

The internal token or key label of the internal token of the PIN encrypting key used in encrypting the *input_PIN_block*. The key must be an **IPINENC** key.

input_PIN_profile

Direction: Input
Type: String

The three 8-byte character elements that contain information necessary to extract the PIN from a formatted PIN block. The valid input PIN formats are **ISO-0**, **ISO-1**, **ISO-2**, and **ISO-3**. See "The PIN profile" on page 449 for additional information.

input_PAN_data

Direction: Input
Type: String

The 12 digit personal account number (PAN) if the input PIN format is **ISO-0** or **ISO-3**. Otherwise, the parameter is ignored.

For **ISO-0** or **ISO-3**, use the rightmost 12 digits of the PAN, excluding the check digit.

secmsg_key_identifier

Direction: Input/Output
Type: String

The internal token or key label of an internal token of a secure message key for encrypting PINs. This key is used to encrypt the updated *clear_text*.

output_PIN_profile

Direction: Input
Type: String

The three 8-byte character elements that contain information necessary to create a formatted PIN block. If reformatting is not required, the *input_PIN_profile* and the *output_PIN_profile* must specify the same PIN block format. Output PIN block formats supported are **ISO-0**, **ISO-1**, **ISO-2**, and **ISO-3**.

output_PAN_data

Direction: Input
Type: String

The 12 digit personal account number (PAN) if the output PIN format is **ISO-0** or **ISO-3**. Otherwise, this parameter is ignored.

For **ISO-0** or **ISO-3**, use the rightmost 12 digits of the PAN, excluding the check digit.

text_length

Direction: Input
Type: Integer

The length of the *clear_text* parameter that follows. Length must be a multiple of eight. Maximum length is 4096.

clear_text

Direction: Input
Type: String

Clear text that contains the recovered and/or reformatted/encrypted PIN at offset specified and then encrypted. Any padding or formatting of the message must be done by the caller on input.

initialization_vector

Direction: Input
Type: String

The 8-byte supplied string for the **TDES-CBC** mode of encryption. The *initialization_vector* is XORed with the first eight bytes of *clear_text* before encryption. This field is ignored for **TDES-ECB** mode.

PIN_offset

Direction: Input
Type: Integer

The offset within the *clear_text* parameter where the reformatted PIN block is to be placed. The first byte of the *clear_text* field is offset 0.

PIN_offset_field_length

Direction: Input
Type: Integer

The length of the field within *clear_text* parameter at *PIN_offset* where the recovered clear *input_PIN_block* value is to be placed. The PIN block might be self-encrypted if requested by the *rule_array*. Length must be eight. The PIN

block must fit entirely within the *clear_text*.

enciphered_text

Direction: Output
Type: String

The field where the enciphered text is returned. The length of this field must be at least as long as the *clear_text* field.

output_chaining_vector

Direction: Output
Type: String

This field contains the last eight bytes of enciphered text and is used as the *initialization_vector* for the next encryption call if data needs to be chained for **TDES-CBC** mode. No data is returned for **TDES-ECB**.

Restrictions

The restrictions for CSNBSPN.

None.

Required commands

The required commands for CSNBSPN.

This verb requires the **Secure Messaging for PINs** command (offset X'0274') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Beginning with Release 4.1.0, three new commands are added (offsets X'0350', X'0351', and X'0352'). These three commands affect how PIN processing is performed as described below:

1. Enable the **ANSI X9.8 PIN - Enforce PIN block restrictions** command (offset X'0350') in the active role to apply additional restrictions to PIN processing implemented in CCA 4.1.0, as follows:
   - Constrain use of **ISO-2** PIN blocks to offline PIN verification and PIN change operations in integrated circuit card environments only. Specifically, do not allow **ISO-2** input or output PIN blocks.
   - Do not reformat a PIN-block format that includes a PAN into a PIN-block format that does not include a PAN.
   - Do not allow a change of PAN data. Specifically, when performing translations between PIN block formats that both include PAN data, do not allow the *input_PAN_data* and *output_PAN_data* variables to be different from the PAN data enciphered in the input PIN block.

   Note: A role with offset X'0350' enabled also affects access control of the Clear PIN Generate Alternate and the Encrypted PIN Translate verbs.
2. Enable the **ANSI X9.8 PIN - Allow modification of PAN** command (offset X'0351') in the active role to override the restriction to not allow a change of PAN data. This override is applicable only when either the **ANSI X9.8 PIN - Enforce PIN block restrictions** command (offset X'0350') or the **ANSI X9.8 PIN - Allow only ANSI PIN blocks** command (offset X'0352') or both are

enabled in the active role. This override is to support account number changes in issuing environments. Offset X'0351' has no effect if neither offset X'0350' nor offset X'0352' is enabled in the active role.

Note: A role with offset X'0351' enabled also affects access control of the Encrypted PIN Translate verbs.

3. Enable the **ANSI X9.8 PIN - Allow only ANSI PIN blocks** command (offset X'0352') in the active role to apply a more restrictive variation of the **ANSI X9.8 PIN - Enforce PIN block restrictions** command (offset X'0350'). In addition to the previously described restrictions of offset X'0350', this command also restricts the *input_PIN_profile* and the *output_PIN_profile* to contain only ISO-0, ISO-1, and ISO-3 PIN block formats. Specifically, the IBM 3624 PIN-block format is not allowed with this command. Offset X'0352' overrides offset X'0350'.

   Note: A role with offset X'0352' enabled also affects access control of the Encrypted PIN Translate verbs.

For more information, see "ANSI X9.8 PIN restrictions" on page 447.

Usage notes

The usage notes for CSNBSPN.

Keys appear in the clear only within the secure boundary of the cryptographic coprocessors, and never in host storage.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBSPNJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBSPNJ(
          hikmNativeInteger return_code,
          hikmNativeInteger reason_code,
          hikmNativeInteger exit_data_length,
          byte[]            exit_data,
          hikmNativeInteger rule_array_count,
          byte[]            rule_array,
          byte[]            in_PIN_blk,
          byte[]            in_PIN_enc_key_id,
          byte[]            in_PIN_profile,
          byte[]            in_PAN_data,
          byte[]            secmsg_key,
          byte[]            out_PIN_profile,
          byte[]            out_PAN_data,
          hikmNativeInteger text_length,
          byte[]            clear_text,
          byte[]            initialization_vector,
          hikmNativeInteger PIN_offset,
          hikmNativeInteger  PIN_offset_field_length,
          byte[]            cipher_text,
          byte[]            output_chaining_value);
```

Transaction Validation (CSNBTRV)

The Transaction Validation verb supports the generation and validation of American Express card security codes (CSC).

This verb generates and verifies transaction values based on information from the transaction and a cryptographic key. You select the validation method, and either the generate or verify mode, through *rule_array* keywords.

For the American Express process, the control vector supplied with the cryptographic key must indicate a MAC or MACVER class key. The key can be single or double length. **DATAM** and **DATAMV** keys are not supported. The MAC generate control vector bit must be on (bit 20) if you request CSC generation and MAC verify bit (bit 21) must be on if you request verification.

Format

The format of CSNBTRV.

```
CSNBTRV(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              rule_array_count,
              rule_array,
              transaction_key_length,
              transaction_key,
              transaction_info_length,
              transaction_info,
              validation_values_length,
              validation_values)
```

Parameters

The parameters for CSNBTRV.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1 or 2.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. The keywords are left-aligned in an 8-byte field and padded on the right with blanks. The keywords must be in contiguous storage. The *rule_array* keywords are described in Table 159.

Table 159. Keywords for Transaction Validation control information

Keyword	Description
Card security code algorithm (One, optional)	
CSC-V1	Specifies use of CSC version 1.0 algorithm for generating or verifying the validation values.
CSC-V2	Specifies use of CSC version 2.0 algorithm for generating or verifying the validation values.
American Express card security codes (One, required)	

Table 159. Keywords for Transaction Validation control information (continued)

Keyword	Description
CSC-3	3-digit card security code (CSC) located on the signature panel. **VERIFY** implied. This is the default.
CSC-4	4-digit card security code (CSC) located on the signature panel. **VERIFY** implied.
CSC-5	5-digit card security code (CSC) located on the signature panel. **VERIFY** implied.
CSC-345	Generate 5-byte, 4-byte, or 3-byte values when given an account number and an expiration date. **GENERATE** implied.
Operation (One, optional)	
VERIFY	Specifies verification of the value presented in the validation values variable.
GENERATE	Specifies generation of the value presented in the validation values variable.

transaction_key_length

Direction: Input
Type: Integer

The length of the **transaction_key** parameter.

transaction_key

Direction: Input
Type: String

The label name or internal token of a **MAC** or **MACVER** class key. The key can be single or double length.

transaction_info_length

Direction: Input
Type: Integer

The length of the *transaction_info* parameter. For the American Express CSC codes, the length must be 19.

transaction_info

Direction: Input
Type: String

For American Express, this is a 19-byte field containing the concatenation of the 4-byte expiration data (in the format YYMM) and the 15-byte American Express account number. Provide the information in character format.

validation_values_length

Direction: Input/Output
Type: Integer

The length of the *validation_values* parameter. Maximum value for this field is 64.

validation_values

Direction: Input
Type: String

This variable contains American Express CSC values. The data is output for **GENERATE** and input for **VERIFY**. See Table 160 on page 555.

Table 160. Values for Transaction Validation validation_values parameter

Operation	Element description
GENERATE and **CSC-345**	5555544444333 where: 55555 = CSC 5 value 4444 = CSC 4 value 333 = CSC 3 value
VERIFY and **CSC-3**	333 = CSC 3 value
VERIFY and **CSC-4**	4444 = CSC 4 value
VERIFY and **CSC-5**	55555 = CSC 5 value

Restrictions

The restrictions for CSNBTRV.

None.

Required commands

The required commands for CSNBTRV.

This verb requires the listed commands to be enabled in the active role, depending on the operation and card security code specified:

Operation keyword	Card security code keyword	Offset	Command
GENERATE	**CSC-345**	X'0291'	**Transaction Validation - Generate**
VERIFY	**CSC-3**	X'0292'	**Transaction Validation - Verify CSC-3**
	CSC-4	X'0293'	**Transaction Validation - Verify CSC-4**
	CSC-5	X'0294'	**Transaction Validation - Verify CSC-5**

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBTRV.

There are additional access control points for this verb.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBTRVJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBTRVJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger transaction_key_length,
```

```
byte[]            transaction_key,
hikmNativeInteger transaction_info_length,
byte[]            transaction_info,
hikmNativeInteger validation_values_length,
byte[]            validation_values);
```

Chapter 11. Financial services for DK PIN methods

The German Banking Industry Committee, *Deutsche Kreditwirtschaft* (DK) specifies PIN methods and requirements for financial services.

DK is an association of the German banking industry. The intellectual property rights regarding the methods and specification belong to the German Banking Industry Committee.

Note: All cryptographic coprocessors must be loaded with the same level of code. There are several licensed internal code (LIC) releases in support of the DK PIN methods. Ensure that all of the coprocessors have the same LIC level to support the function you want to use.

The following financial services (verbs) are described in this topic:

Weak PIN table

The DK PIN methods support the use of a table of weak PINs.

Services that generate PINs compare the generated PIN against the table and if the PIN is in the table, the service generates a different PIN. Services that change PINs compare the new PIN against the table and if the new PIN is in the table, the service fails.

Weak PIN tables can be stored in the cryptographic coprocessors for use by callable services. Only tables that have been activated can be used. A TKE Workstation is required to manage the tables in the coprocessors.

A Trusted Key Entry workstation (TKE) is required to administer the weak PIN tables for each adapter. In the TKE documentation and user interface, each domain has a restricted PIN table. The corresponding tab is called **Domain Restricted PINs**. The user may activate, load, and remove PINs from the weak PIN tables on a per-domain basis.

Note: All coprocessors must have installed the same table of weak PINs.

DK PIN methods

The DK PIN methods use a PIN reference value or word (PRW) to verify PINs rather than regenerating the PIN from customer account data.

The PRW is generated by concatenating the customer PAN data, the issuer card data, the PIN length, the PIN, and a 4-byte random number. It is encrypted using a PRW key with the GENONLY key usage. The PRW and random number are the output of the generation. The PIN is verified by generating the PRW using a PRW key with the VERIFY key usage and comparing it against the supplied PRW and random number.

DK Deterministic PIN Generate (CSNBDDPG)

Use the DK Deterministic PIN Generate service to generate a PIN and PIN reference value or word (PRW) using an AES PIN calculation key. The PIN reference value is used to verify the PIN in other services.

Note: The PIN generation process uses the account information to calculate the PIN. If the process generates a PIN that appears in the weak PIN table, it rejects that PIN, increments the rightmost byte of the account information by 1, and generates another PIN. The PIN generation process continues to increment the byte until an acceptable PIN is generated. For additional information, see "Weak PIN table" on page 557.

You can use this service to perform the following tasks:

- Generate an encrypted PIN block in PBF-1 format with a PIN print key to be printed on a PIN mailer.
- Generate a PRW which can be used to verify the PIN.
- Optionally, generate an encrypted PIN block in PBF-1 format to be stored for later use in personalizing replacement cards.

Weak PINs: The PIN generation process uses the account information to calculate the PIN. During the PIN generation process, if a generated PIN is found in the weak PIN table, it is rejected; the process increments the rightmost byte of the account information by 1 and generates another PIN. The process repeats itself until a suitable PIN is generated.

Format

The format of CSNBDDPG.

```
CSNBDDPG(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             account_info_ER_length,
             account_info_ER,
             PAN_data_length,
             PAN_data,
             card_p_data_length,
             card_p_data,
             card_t_data_length,
             card_t_data,
             PIN_length,
             PIN_generation_key_identifier_length,
             PIN_generation_key_identifier,
             PRW_key_identifier_length,
             PRW_key_identifier,
             PIN_print_key_identifier_length,
             PIN_print_key_identifier,
             OPIN_encryption_key_identifier_length,
             OPIN_encryption_key_identifier,
             OEPB_MAC_key_identifier_length,
             OEPB_MAC_key_identifier,
             PIN_reference_value_length,
             PIN_reference_value,
             PRW_random_number_length,
             PRW_random_number,
             PIN_print_block_length,
             PIN_print_block,
             encrypted_PIN_block_length,
             encrypted_PIN_block,
             PIN_block_MAC_length,
             PIN_block_MAC)
```

Parameters

The parameters for CSNBDDPG.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. This value must be 0 or 1.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. The keywords are left-aligned in an 8-byte field and padded on the right with blanks. The

keywords must be in contiguous storage. The *rule_array* keywords are described in Table 161.

Table 161. Keywords for DK Deterministic PIN Generate control information

Keyword	Description
PIN Block output selection keyword (One, optional)	
EPB	Return an encrypted PIN block (EPB) and a MAC of the encrypted PIN block.
NOEPB	Do not return an encrypted PIN block. This is the default value.

account_info_ER_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **account_info_ER** parameter. The value must be 16.

account_info_ER

Direction: Input
Type: String

The 16-byte account information used to generate the PIN, right-aligned and padded on the left with binary zeros.

PAN_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PAN_data** parameter. The value must be in the range 10 - 19.

PAN_data

Direction: Input
Type: String

The PAN data to which the PIN is associated. The full account number, including check digit, should be included.

card_p_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **card_p_data** parameter. The value must be in the range 2 - 256.

card_p_data

Direction: Input
Type: String

The time-invariant card data (CDp), determined by the card issuer, which is used to differentiate between multiple cards for one account.

card_t_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **card_t_data** parameter. The value must be in the range 2 - 256.

card_t_data

Direction: Input
Type: String

The time-sensitive card data, determined by the card issuer, which, together with the account number and the **card_p_data** value, specifies an individual card.

PIN_length

Direction: Input
Type: Integer

Specifies the length of the PIN to be generated. This value must be in the range 4 - 12.

PIN_generation_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PIN_generation_key_identifier** parameter. If the **PIN_generation_key_identifier** contains a label, the value must be 64. Otherwise, the value must be at least the actual token length, up to 725.

PIN_generation_key_identifier

Direction: Input
Type: String

The identifier of the PIN generating key. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINCALC, the key usage fields must indicate GENONLY, CBC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

PRW_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PRW_key_identifier** parameter. If the **PRW_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

PRW_key_identifier

Direction: Input/Output
Type: String

The identifier of the PRW generating key. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINPRW, the key usage fields must indicate GENONLY, CMAC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

PIN_print_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PIN_print_key_identifier** parameter. If the **PIN_print_key_identifier** contains a label, the value must be 64. Otherwise, the value must be at least the actual token length, up to 725.

PIN_print_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to wrap the PIN for printing. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINPROT, and the key usage fields must indicate ENCRYPT, CBC, and DKPINOPP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

OPIN_encryption_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **OPIN_encryption_key_identifier** parameter. If the rule array indicates that no encrypted PIN block is to be returned, this value must be 0. If the **OPIN_encryption_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

OPIN_encryption_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to wrap the PIN block. The key identifier is an operational token or the key label of an operational token in key storage. If the rule array indicates that no encrypted PIN block is to be returned, this parameter is ignored. The key algorithm of this key must be AES, the key type must be PINPROT, and the key usage fields must indicate ENCRYPT, CBC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

OEPB_MAC_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **OEPB_MAC_key_identifier** parameter. If the rule array indicates that no encrypted PIN block MAC is to be returned, this value must be 0. If the **OEPB_MAC_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

OEPB_MAC_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to generate the MAC of the PIN block. The key identifier is an operational token or the key label of an operational token in

key storage. If the rule array indicates that no encrypted PIN block is to be returned, this parameter is ignored. The key algorithm of this key must be AES, the key type must be MAC, and the key usage fields must indicate CMAC, GENONLY, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

PIN_reference_value_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **PIN_reference_value** parameter. The value must be at least 16. On output, it is set to 16.

PIN_reference_value

Direction: Output
Type: String

The 16-byte calculated PIN reference value.

PRW_random_number_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **PRW_random_number** parameter. The value must be at least 4. On output, it is set to 4.

PRW_random_number

Direction: Output
Type: String

The 4-byte random number associated with the PIN reference value.

PIN_print_block_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **PIN_print_block** parameter. The value must be at least 32. On output, it is set to 32.

PIN_print_block

Direction: Output
Type: String

The 32-byte encrypted PIN block to be passed to the PIN mailer function.

encrypted_PIN_block_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **encrypted_PIN_block** parameter. If the rule array indicates that no encrypted PIN block should be returned, this value must be 0. Otherwise, it should be at least 32.

encrypted_PIN_block

Direction: Output

Type: String

The 32-byte encrypted PIN block in PBF-1 format. This parameter is ignored if no encrypted PIN block is returned.

PIN_block_MAC_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **PIN_block_MAC** parameter. If the rule_array indicates that no PIN block MAC should be returned, this value must be 0. Otherwise, it must be at least 8.

PIN_block_MAC

Direction: Output
Type: String

The 8-byte CMAC of the encrypted PIN block. This parameter is ignored if no encrypted PIN block is returned.

Restrictions

The restrictions for CSNBDDPG.

- Use of the Visa PVV PIN-calculation method always produces a four-digit output rather than padding the output with binary zeros to the length of the PIN.
- A key identifier length must be 64 for a key label.

Required commands

The required commands for CSNBDDPG.

The DK Deterministic PIN Generate verb requires the **DK Deterministic PIN Generate** command (offset X'02C6') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBDDPG.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBDDPGJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBDDPGJ(
           hikmNativeInteger    return_code,
           hikmNativeInteger    reason_code,
           hikmNativeInteger    exit_data_length,
           byte[]               exit_data,
           hikmNativeInteger    rule_array_count,
           byte[]               rule_array,
           hikmNativeInteger    account_info_ER_length,
```

```
                        byte[]              account_info_ER,
                        hikmNativeInteger   PAN_data_length,
                        byte[]              PAN_data,
                        hikmNativeInteger   card_p_data_length,
                        byte[]              card_p_data,
                        hikmNativeInteger   card_t_data_length,
                        byte[]              card_t_data,
                        hikmNativeInteger   PIN_length,
                        hikmNativeInteger   PIN_generation_key_identifier_length,
                        byte[]              PIN_generation_key_identifier,
                        hikmNativeInteger   PRW_key_identifier_length,
                        byte[]              PRW_key_identifier,
                        hikmNativeInteger   PIN_print_key_identifier_length,
                        byte[]              PIN_print_key_identifier,
                        hikmNativeInteger   OPIN_encryption_key_identifier_length,
                        byte[]              OPIN_encryption_key_identifier,
                        hikmNativeInteger   OEPB_MAC_key_identifier_length,
                        byte[]              OEPB_MAC_key_identifier,
                        hikmNativeInteger   PIN_reference_value_length,
                        byte[]              PIN_reference_value,
                        hikmNativeInteger   PRW_random_number_length,
                        byte[]              PRW_random_number,
                        hikmNativeInteger   PIN_print_block_length,
                        byte[]              PIN_print_block,
                        hikmNativeInteger   encrypted_PIN_block_length,
                        byte[]              encrypted_PIN_block,
                        hikmNativeInteger   PIN_block_MAC_length,
                        byte[]              PIN_block_MAC);
```

DK Migrate PIN (CSNBDMP)

Use the DK Migrate PIN verb to generate the PIN reference value (PRW) for a specified user account. The PIN reference value is used to verify the PIN in other services.

An ISO-1 formatted PIN block is input to determine the value of the PIN for the account. The PIN is reformatted into a DK-defined PIN block and the PIN reference value is calculated using a PRW random value and other account information. The PIN reference value and associated PRW random value are returned to be used as input by other PIN processes to verify the PIN.

If validation of the PIN is desired to personalize smart cards, specify the EPB **rule-array** keyword. This keyword causes an output encrypted PIN block to be returned along with a PIN block MAC. The MAC is calculated over the output PIN block and additional card data using the block cipher-based MAC algorithm called CMAC (NIST SP 800-38B).

Note: This service does not test for weak PINs.

Format

The format of CSNBDMP.

```
CSNBDMP(
           return_code,
           reason_code,
           exit_data_length,
           exit_data,
           rule_array_count,
           rule_array,
           PAN_data_length,
           PAN_data,
           card_p_data_length,
           card_p_data,
           card_t_data_length,
           card_t_data,
           ISO1_PIN_block_length,
           ISO1_PIN_block,
           IPIN_encryption_key_identifier_length,
           IPIN_encryption_key_identifier,
           PRW_key_identifier_length,
           PRW_key_identifier,
           OPIN_encryption_key_identifier_length,
           OPIN_encryption_key_identifier,
           OEPB_MAC_key_identifier_length,
           OEPB_MAC_key_identifier,
           PIN_reference_value_length,
           PIN_reference_value,
           PRW_random_number_length,
           PRW_random_number,
           encrypted_PIN_block_length,
           encrypted_PIN_block,
           PIN_block_MAC_length,
           PIN_block_MAC)
```

Parameters

The parameters for CSNBDMP.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. This value must be 0 or 1.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the callable service. The keywords must be in contiguous storage with each of the keywords left-justified in its own 8-byte location and padded on the right with blanks. The *rule_array* keywords are described in Table 162 on page 567.

Table 162. Keywords for DK Migrate PIN control information

Keyword	Description
PIN Block output selection keyword (One, optional)	
EPB	Return an encrypted PIN block and a MAC of the encrypted PIN block.
NOEPB	Do not return an encrypted PIN block (EPB). This is the default value.

PAN_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PAN_data** parameter. The value must be in the range 10 - 19.

PAN_data

Direction: Input
Type: String

The personal account number in character form which the PIN is associated. The primary account number, including check digit, should be included.

card_p_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **card_p_data** parameter. The value must be in the range 2 - 256.

card_p_data

Direction: Input
Type: String

The time-invariant card data (CDp), determined by the card issuer, which is used to differentiate between multiple cards for one account.

card_t_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **card_t_data** parameter. The value must be in the range 2 - 256.

card_t_data

Direction: Input
Type: String

The time-sensitive card data, determined by the card issuer, which, together with the account number and the **card_p_data**, specifies an individual card.

ISO1_PIN_block_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **ISO1_PIN_block** parameter. This value must be 8.

ISO1_PIN_block

Direction: Input
Type: String

The 8-byte encrypted PIN block with the current PIN in ISO-1 format with the customer chosen PIN. This PIN is used to generate the PIN reference value.

IPIN_encryption_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **IPIN_encryption_key_identifier** parameter. If the **IPIN_encryption_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

IPIN_encryption_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to decrypt the PIN block containing the IOS-1 PIN. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be DES and the key type must be IPINENC.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

PRW_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PRW_key_identifier** parameter. If **PRW_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

PRW_key_identifier

Direction: Input/Output
Type: String

The identifier of the PRW generating key. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINPRW, and the key usage fields must indicate GENONLY, CMAC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

OPIN_encryption_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **OPIN_encryption_key_identifier** parameter. If the rule array indicates that no encrypted PIN block is to be returned, this value must be 0. If **OPIN_encryption_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

OPIN_encryption_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to wrap the PIN block. The key identifier is an operational token or the key label of an operational token in key storage. If the rule array indicates that no encrypted PIN block is to be returned, this parameter is ignored. The key algorithm of this key must be AES, the key type must be PINPROT, and the key usage fields must indicate ENCRYPT, CBC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

OEPB_MAC_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **OEPB_MAC_key_identifier** parameter. If the rule array indicates that no encrypted PIN block MAC is to be returned, this value must be 0. If **OEPB_MAC_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

OEPB_MAC_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to generate the MAC of the PIN block. The key identifier is an operational token or the key label of an operational token in key storage. If the rule array indicates that no encrypted PIN block is to be returned, this parameter is ignored. The key algorithm of this key must be AES, the key type must be MAC, and the key usage fields must indicate GENONLY, CMAC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

PIN_reference_value_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **PIN_reference_value** parameter. This value must be 16. On output, **PIN_reference_value_length** is set to 16.

PIN_reference_value

Direction: Output
Type: String

The 16-byte calculated PIN reference value.

PRW_random_number_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **PRW_random_number** parameter. The value must be 4. On output, **PRW_random_number_length** is set to 4.

PRW_random_number

Direction: Output

Type: String

The 4-byte random number associated with the PIN reference value.

encrypted_PIN_block_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the encrypted_PIN_block parameter. If the rule array indicates that no encrypted PIN block should be returned, this value must be 0. Otherwise, it should be at least 32.

encrypted_PIN_block

Direction: Output
Type: String

The 32-byte encrypted PIN block in PBF-1 format. This parameter is ignored if no encrypted PIN block is returned.

PIN_block_MAC_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the PIN_block_MAC parameter. If the rule_array indicates that no PIN block MAC should be returned, this value must be 0. Otherwise, it must be at least 8.

PIN_block_MAC

Direction: Output
Type: String

The 8-byte CMAC of the encrypted PIN block. This parameter is ignored if no encrypted PIN block is returned.

Restrictions

The restrictions for CSNBDMP.

A key identifier length must be 64 for a key label.

Required commands

The required commands for CSNBDMP.

The DK Migrate PIN verb requires the **DK Migrate PIN** command (offset X'02CE') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBDMP.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBDMPJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBDMPJ(
           hikmNativeInteger    return_code,
           hikmNativeInteger    reason_code,
           hikmNativeInteger    exit_data_length,
           byte[]               exit_data,
           hikmNativeInteger    rule_array_count,
           byte[]               rule_array,
           hikmNativeInteger    PAN_data_length,
           byte[]               PAN_data,
           hikmNativeInteger    card_p_data_length,
           byte[]               card_p_data,
           hikmNativeInteger    card_t_data_length,
           byte[]               card_t_data,
           hikmNativeInteger    ISO1_PIN_block_length,
           byte[]               ISO1_PIN_block,
           hikmNativeInteger    IPIN_encryption_key_identifier_length,
           byte[]               IPIN_encryption_key_identifier,
           hikmNativeInteger    PRW_key_identifier_length,
           byte[]               PRW_key_identifier,
           hikmNativeInteger    OPIN_encryption_key_identifier_length,
           byte[]               OPIN_encryption_key_identifier,
           hikmNativeInteger    OEPB_MAC_key_identifier_length,
           byte[]               OEPB_MAC_key_identifier,
           hikmNativeInteger    PIN_reference_value_length,
           byte[]               PIN_reference_value,
           hikmNativeInteger    PRW_random_number_length,
           byte[]               PRW_random_number,
           hikmNativeInteger    output_encrypted_PIN_block_length,
           byte[]               output_encrypted_PIN_block,
           hikmNativeInteger    PIN_block_MAC_length,
           byte[]               PIN_block_MAC);
```

DK PAN Modify in Transaction (CSNBDPMT)

Use the DK PAN Modify in Transaction verb to obtain a new PIN reference value (PRW) for an existing PIN when a merger has occurred and the account information has changed.

The input includes the current PIN, the account information (PAN and card data) for the current, and the new account.

The DK PRW CMAC Generate service is called prior to this service to generate the MAC of the changed account information. If the MAC associated with the account information does not verify, the service fails.

Format

The format of CSNBDPMT.

```
CSNBDPMT(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             current_PAN_data_length,
             current_PAN_data,
             new_PAN_data_length,
             new_PAN_data,
             current_card_p_data_length,
             current_card_p_data,
             current_card_t_data_length,
             current_card_t_data,
             new_card_p_data_length,
             new_card_p_data,
             new_card_t_data_length,
             new_card_t_data,
             CMAC_FUS_length,
             CMAC_FUS,
             ISO_encrypted_PIN_block_length,
             ISO_encrypted_PIN_block,
             current_PIN_reference_value_length,
             current_PIN_reference_value,
             current_PRW_random_number_length,
             current_PRW_random_number,
             CMAC_FUS_key_identifier_length,
             CMAC_FUS_key_identifier,
             IPIN_encryption_key_identifier_length,
             IPIN_encryption_key_identifier,
             PRW_key_identifier_length,
             PRW_key_identifier,
             new_PRW_key_identifier_length,
             new_PRW_key_identifier,
             new_PIN_reference_value_length,
             new_PIN_reference_value,
             new_PRW_random_number_length,
             new_PRW_random_number)
```

Parameters

The parameters for CSNBDPMT.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. This value must be 0.

rule_array

Direction: Input
Type: String array

There are no keywords for this service.

current_PAN_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **current_PAN_data** parameter. The value must be in the range 10 - 19.

current_PAN_data

Direction: Input
Type: String

The current PAN data associated with the PIN. The full account number, including check digit, should be included.

new_PAN_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **new_PAN_data** parameter. The value must be in the range 10 - 19.

new_PAN_data

Direction: Output
Type: Integer

The new PAN data to be associated with the PIN. The full account number, including check digit, should be included.

current_card_p_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **current_card_p_data** parameter. The value must be in the range 2 - 256.

current_card_p_data

Direction: Input
Type: String

The time-invariant card data (CDp) of the current account, determined by the card issuer.

current_card_t_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **current_card_t_data** parameter. The value must be in the range 2 - 256.

current_card_t_data

Direction: Input
Type: String

The time-sensitive card data of the current account, determined by the card issuer.

new_card_p_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **new_card_p_data** parameter. The value must be in the range 2 - 256.

new_card_p_data

Direction: Input
Type: String

The time-invariant card data (CDp) of the current account, determined by the card issuer.

new_card_t_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **new_card_t_data** parameter. The value must be in the range 2 - 256.

new_card_t_data

Direction: Input
Type: String

The time-sensitive card data of the current account, determined by the card issuer.

CMAC_FUS_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **CMAC_FUS** parameter. The value must be in the range 8 - 16.

CMAC_FUS

Direction: Input
Type: String

The 8-byte to 16-byte MAC that was generated from the current and new PANs and card data strings and PIN reference values. The MAC is generated using the DK PRW CMAC Generate service.

ISO_encrypted_PIN_block_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **ISO_encrypted_PIN_block** parameter. The value must be 8.

ISO_encrypted_PIN_block

Direction: Input
Type: String

The 8-byte encrypted PIN block with the PIN in ISO-1 format.

current_PIN_reference_value_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **current_PIN_reference_value** parameter. The value must be 16.

current_PIN_reference_value

Direction: Input
Type: String

The 16-byte PIN reference value for comparison to the calculated value.

current_PRW_random_number_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **current_PRW_random_number** parameter. The value must be 4.

current_PRW_random_number

Direction: Input
Type: String

The 4-byte random number associated with the PIN reference value.

CMAC_FUS_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **CMAC_FUS_key_identifier** parameter. If **CMAC_FUS_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

CMAC_FUS_key_identifier

Direction: Input
Type: String

The identifier of the key to verify the **CMAC_FUS** value. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be MAC, and the key usage fields must indicate VERIFY, CMAC, and DKPINAD2.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

IPIN_encryption_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **IPIN_encryption_key_identifier** parameter. If **IPIN_encryption_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

IPIN_encryption_key_identifier

Direction: Input
Type: String

The identifier of the key to decrypt the **ISO_encrypted_PIN_block**. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be DES and the key type must be IPINENC.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

PRW_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PRW_key_identifier** parameter. If **PRW_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

PRW_key_identifier

Direction: Input
Type: String

The identifier of the key to verify the input PRW. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINPRW, and the key usage fields must indicate VERIFY, CMAC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

new_PRW_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **new_PRW_key_identifier** parameter. If **new_PRW_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

new_PRW_key_identifier

Direction: Input
Type: String

The identifier of the key to generate the new PRW. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINPRW, and the key usage fields must indicate GENONLY, CMAC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

new_PIN_reference_value_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **new_PIN_reference_value** parameter. The value must be at least 16. On output, it is set to 16.

new_PIN_reference_value

Direction: Input
Type: String

The 16-byte new PIN reference value.

new_PRW_random_number_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **new_PRW_random_number** parameter. The value must be at least 4. On output, it is set to 4.

new_PRW_random_number

Direction: Input
Type: String

The 4-byte random number associated with the new PIN reference value.

Restrictions

The restrictions for CSNBDPMT.

None.

Required commands

The required commands for CSNBDPMT.

The DK PAN Modify in Transaction verb requires the **DK PAN Modify in Transaction** command (offset X'02C5') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBDPMT.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBDPMTJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBDPMTJ(
           hikmNativeInteger    return_code,
           hikmNativeInteger    reason_code,
           hikmNativeInteger    exit_data_length,
           byte[]               exit_data,
           hikmNativeInteger    rule_array_count,
           byte[]               rule_array,
           hikmNativeInteger    current_PAN_data_length,
           byte[]               current_PAN_data,
           hikmNativeInteger    new_PAN_data_length,
           byte[]               new_PAN_data,
           hikmNativeInteger    current_card_p_data_length,
           byte[]               current_card_p_data,
           hikmNativeInteger    current_card_t_data_length,
           byte[]               current_card_t_data,
           hikmNativeInteger    new_card_p_data_length,
```

```
byte[]              new_card_p_data,
hikmNativeInteger   new_card_t_data_length,
byte[]              new_card_t_data,
hikmNativeInteger   CMAC_FUS_length,
byte[]              CMAC_FUS,
hikmNativeInteger   ISO_encrypted_PIN_block_length,
byte[]              ISO_encrypted_PIN_block,
hikmNativeInteger   current_PIN_reference_value_length,
byte[]              current_PIN_reference_value,
hikmNativeInteger   current_PRW_random_number_length,
byte[]              current_PRW_random_number,
hikmNativeInteger   CMAC_FUS_key_identifier_length,
byte[]              CMAC_FUS_key_identifier,
hikmNativeInteger   IPIN_encryption_key_identifier_length,
byte[]              IPIN_encryption_key_identifier,
hikmNativeInteger   PRW_key_identifier_length,
byte[]              PRW_key_identifier,
hikmNativeInteger   new_PRW_key_identifier_length,
byte[]              new_PRW_key_identifier,
hikmNativeInteger   new_PIN_reference_value_length,
byte[]              new_PIN_reference_value,
hikmNativeInteger   new_PRW_random_number_length,
byte[]              new_PRW_random_number);
```

DK PAN Translate (CSNBDPT)

Use the DK PAN Translate verb to create an encrypted PIN block with the same PIN and a different PAN. The account data may change, but changing the PIN is to be avoided. This service creates a new encrypted PIN block and MAC on the encrypted PIN block that is used to accept the PAN change at an authorization node.

You can use this service to perform the following tasks:

- Generate an encrypted PIN block in PBF-1 format with a changed PAN to be used at the authorization node to create a PIN reference value.
- Generate a CMAC over the encrypted PIN block for validation.

Format

The format of CSNBDPT.

```
CSNBDPT(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             card_p_data_length,
             card_p_data,
             card_t_data_length,
             card_t_data,
             new_PAN_data_length,
             new_PAN_data,
             new_card_p_data_length,
             new_card_p_data,
             PIN_reference_value_length,
             PIN_reference_value,
             PRW_random_number_length,
             PRW_random_number,
             current_encrypted_PIN_block_length,
             current_encrypted_PIN_block,
             current_PIN_block_MAC_length,
             current_PIN_block_MAC,
             PRW_key_identifier_length,
             PRW_key_identifier,
             IPIN_encryption_key_identifier_length,
             IPIN_encryption_key_identifier,
             IEPB_MAC_key_identifier_length,
             IEPB_MAC_key_identifier,
             OPIN_encryption_key_identifier_length,
             OPIN_encryption_key_identifier,
             OEPB_MAC_key_identifier_length,
             OEPB_MAC_key_identifier,
             new_encrypted_PIN_block_length,
             new_encrypted_PIN_block,
             new_PIN_block_MAC_length,
             new_PIN_block_MAC)
```

Parameters

The parameters for CSNBDPT.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. This value must be 0.

rule_array

Direction: Input
Type: String array

There are no keywords for this service.

card_p_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **card_p_data** parameter. The value must be in the range 2 - 256.

card_p_data

Direction: Input
Type: String

The time-invariant card data (CDp), determined by the card issuer, which is used to differentiate between multiple cards for one account.

card_t_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **card_t_data** parameter. The value must be in the range 2 - 256.

card_t_data

Direction: Input
Type: String

The time-sensitive card data, determined by the card issuer, which, together with the account number and the **card_p_data**, specifies an individual card.

new_PAN_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **new_PAN_data** parameter. The value must be in the range 10 - 19.

new_PAN_data

Direction: Input
Type: String

The new personal account number (in character string form) to which the PIN is associated. The full account number, including check digit, should be included.

new_card_p_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **new_card_p_data** parameter. The value must be in the range 2 - 256.

new_card_p_data

Direction: Input
Type: String

The time-invariant card data (CDp), determined by the card issuer, which is used to differentiate between multiple cards for one account.

PIN_reference_value_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PIN_reference_value** parameter. The value must be 16.

PIN_reference_value

Direction: Input
Type: String

The 16-byte PIN reference value for comparison to the calculated value.

PRW_random_number_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PRW_random_number** parameter. The value must be 4.

PRW_random_number

Direction: Input
Type: String

The 4-byte random number associated with the PIN reference value.

current_encrypted_PIN_block_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **current_encrypted_PIN_block** parameter. The value must be 32.

current_encrypted_PIN_block

Direction: Input
Type: Integer

The 32-byte encrypted PIN block in PBF-1 format of the current PIN.

current_PIN_block_MAC_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **current_PIN_block_MAC** parameter. The value must be 8.

current_PIN_block_MAC

Direction: Input
Type: String

The 8-byte MAC of the current encrypted PIN block and the **card_p_data**.

PRW_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PRW_key_identifier** parameter. If

PRW_key_identifier contains a label, the length must be 64. Otherwise, the value must at least the actual token length, up to 725.

PRW_key_identifier

Direction: Input
Type: String

The identifier of the PRW verifying key. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINPRW, the key usage fields must indicate VERIFY, CMAC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

IPIN_encryption_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **IPIN_encryption_key_identifier** parameter. If the **IPIN_encryption_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

IPIN_encryption_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to decrypt the PIN block containing the current PIN. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINPROT, and the key usage fields must indicate DECRYPT, CBC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

IEPB_MAC_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **IEPB_MAC_key_identifier** parameter. If the **IEPB_MAC_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

IEPB_MAC_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to verify MAC of the inbound encrypted PIN block. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be MAC, and the key usage fields must indicate CMAC. VERIFY, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

OPIN_encryption_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **OPIN_encryption_key_identifier** parameter. If the **OPIN_encryption_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

OPIN_encryption_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to encrypt the new PIN block. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINPROT, and the key usage fields must indicate ENCRYPT, CBC, and DKPINAD1.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

OEPB_MAC_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **OEPB_MAC_key_identifier** parameter. If the **OEPB_MAC_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

OEPB_MAC_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to generate the MAC of the new encrypted PIN block. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be MAC, and the key usage fields must indicate CMAC, GENONLY, and DKPINAD1.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

new_encrypted_PIN_block_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **new_encrypted_PIN_block** parameter. The value must be at least 32. On output, it is set to 32.

new_encrypted_PIN_block

Direction: Output
Type: String

The 32-byte encrypted new PIN block.

new_PIN_block_MAC_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **new_PIN_block_MAC** parameter. The value must be at least 8.

new_PIN_block_MAC

Direction: Output
Type: String

The 8-byte MAC of the new encrypted PIN block.

Restrictions

The restrictions for CSNBDPT.

None.

Required commands

The required commands for CSNBDPT.

The DK PAN Translate verb requires the **DK PAN Translate** command (offset X'02C7') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBDPT.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBDPTJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBDPTJ(
          hikmNativeInteger    return_code,
          hikmNativeInteger    reason_code,
          hikmNativeInteger    exit_data_length,
          byte[]               exit_data,
          hikmNativeInteger    rule_array_count
          byte[]               rule_array,
          hikmNativeInteger    card_p_data_length,
          byte[]               card_p_data,
          hikmNativeInteger    card_t_data_length,
          byte[]               card_t_data,
          hikmNativeInteger    new_PAN_data_length,
          byte[]               new_PAN_data,
          hikmNativeInteger    new_card_p_data_length,
          byte[]               new_card_p_data,
          hikmNativeInteger    PIN_reference_value_length,
          byte[]               PIN_reference_value,
          hikmNativeInteger    PRW_random_number_length,
          byte[]               PRW_random_number,
          hikmNativeInteger    current_encrypted_PIN_block_length,
          byte[]               current_encrypted_PIN_block,
          hikmNativeInteger    current_PIN_block_MAC_length,
          byte[]               current_PIN_block_MAC,
          hikmNativeInteger    PRW_MAC_key_identifier_length,
```

```
byte[]              PRW_MAC_key_identifier,
hikmNativeInteger   IPIN_encryption_key_identifier_length,
byte[]              IPIN_encryption_key_identifier,
hikmNativeInteger   IEPB_MAC_key_identifier_length,
byte[]              IEPB_MAC_key_identifier,
hikmNativeInteger   OPIN_encryption_key_identifier_length,
byte[]              OPIN_encryption_key_identifier,
hikmNativeInteger   OEPB_MAC_key_identifier_length,
byte[]              OEPB_MAC_key_identifier,
hikmNativeInteger   new_encrypted_PIN_block_length,
byte[]              new_encrypted_PIN_block,
hikmNativeInteger   new_PIN_block_MAC_length,
byte[]              new_PIN_block_MAC);
```

DK PIN Change (CSNBDPC)

Use the DK PIN Change verb to update the personal identification number (PIN) reference value or word (PRW) for a specified account when a cardholder uses a bank or credit card at an ATM or point-of-sale (POS) terminal to update a card with a new PIN, in other words, to change the current PIN to a customer-selected PIN.

When cardholders use a terminal to update a card with a new PIN, they enter the current PIN and the new PIN. At the terminal, the current PIN is formatted into an 8-byte ISO-1 PIN block and enciphered using a PIN encrypting key for the current PIN. Likewise, the new PIN is formatted into an 8-byte ISO-1 PIN block and enciphered using a PIN encrypting key for the new PIN.

The account of a cardholder is uniquely identified by a 10 - 19 digit primary account number (PAN) of the bank or credit card. Additional information normally available on the card is a time-sensitive card expiry date and a time-invariant (permanent) card sequence number. The verb verifies the current PIN by deciphering the current ISO-1 PIN block and uses the recovered PIN and other additional information to verify the current PIN. If the current PIN does not verify, the process is aborted and an error is returned. If it does verify, the new PIN is recovered from the new ISO-1 PIN block and is reformatted into a DK-defined PIN block that is used with a new PRW random value and other information to calculate a new PRW. The new PRW and associated new PRW random value are returned to be used as input later by other PIN processes for PIN verification.

A card script can be created and encrypted for use later to update a customer smart (chip) card. To create a TDES-encrypted card script, specify either the TDES-CBC or TDES-ECB script selection keyword in the rule array. Beginning with Release 4.4, to create an AES-encrypted card script, specify AES-CBC.

If validation of the PIN is desired to personalize a smart card, specify the EPB PIN block output selection rule-array keyword. This keyword causes an output encrypted PIN block to be returned along with a PIN block MAC. The MAC is calculated over the output PIN block and additional card data using the block cipher-based MAC algorithm, called CMAC (refer to *NIST SP 800-38B*).

Note: If the PIN recovered from the **new_ISO1_PIN_block** variable is found in the weak PIN table, it is rejected and an error is returned indicating that the selected PIN was in the weak PIN table.

Format

The format of CSNBDPC.

```
CSNBDPC(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              rule_array_count,
              rule_array,
              PAN_data_length,
              PAN_data,
              card_p_data_length,
              card_p_data,
              card_t_data_length,
              card_t_data,
              cur_ISO1_PIN_block_length,
              cur_ISO1_PIN_block,
              new_ISO1_PIN_block_length,
              new_ISO1_PIN_block,
              card_script_data_length,
              card_script_data,
              script_offset,
              script_offset_field_length,
              script_initialization_vector_length,
              script_initialization_vector,
              output_PIN_profile,
              PIN_reference_value_length,
              PIN_reference_value,
              PRW_random_number_length,
              PRW_random_number,
              PRW_key_identifier_length,
              PRW_key_identifier,
              cur_IPIN_encryption_key_identifier_length,
              cur_IPIN_encryption_key_identifier,
              new_IPIN_encryption_key_identifier_length,
              new_IPIN_encryption_key_identifier,
              script_key_identifier_length,
              script_key_identifier,
              script_MAC_key_identifier_length,
              script_MAC_key_identifier,
              new_PRW_key_identifier_length,
              new_PRW_key_identifier,
              OPIN_encryption_key_identifier_length,
              OPIN_encryption_key_identifier,
              OEPB_MAC_key_identifier_length,
              OEPB_MAC_key_identifier,
              script_length,
              script,
              script_MAC_length,
              script_MAC,
              new_PIN_reference_value_length,
              new_PIN_reference_value,
              new_PRW_random_number_length,
              new_PRW_random_number,
              output_encrypted_PIN_block_length,
              output_encrypted_PIN_block,
              PIN_block_MAC_length,
              PIN_block_MAC)
```

Parameters

The parameters for CSNBDPC.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. The value must be 0, 1, 2, 3, 4, or 5.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. The keywords must be in contiguous storage with each of the keywords left-justified in its own 8-byte location and padded on the right with blanks. The **rule_array** keywords are described in Table 163.

Table 163. Keywords for DK PIN Change control information

Keyword	Description
PIN Block output selection keyword (One, optional)	
EPB	Return an encrypted PIN block and a MAC to verify the encrypted PIN block.
NOEPB	Do not return an encrypted PIN block (EPB). This is the default value.
Script selection algorithm and method (One, optional)	
AES-CBC (Release 4.4 or later)	Use CBC mode to AES encrypt the PIN block in the script.
NOSCRIPT	Do not return an encrypted SMPIN message with a MAC. This is the default value.
TDES-CBC	Use CBC mode to TDES encrypt the PIN block in the script.
TDES-ECB	Use ECB mode to TDES encrypt the PIN block in the script.
PIN encryption keyword (One, optional) Only valid if TDES-CBC or TDES-ECB is selected.	
CLEARPIN	Do not encrypt the PIN prior to inserting in the script block. This is the default value.
SELF-ENC	Copy the PIN-block self-encrypted to the clear PIN block within the clear output message. Use this rule array keyword to specify that the 8-byte PIN block shall be used as a DES key to encrypt the PIN block. The service copies the self-encrypted PIN block to the clear PIN block in the output message.
MAC Ciphering Method (One required for AES-CBC, one optional for TDES-CBC or TDES-ECB, otherwise not allowed.)	
CMAC (Release 4.4 or later)	Specifies to use the cipher-based MAC algorithm block cipher mode of operation for authentication, recommended in NIST SP 800-38B. Required for AES-CBC. Only valid with AES-CBC.
EMVMACD	Specifies the EMV-related message-padding and calculation method.
TDES-MAC	Specifies the ANS X9.9 Option 1 (binary data) procedure and a CBC Triple-DES encryption of the data.
X9.19OPT	Specifies the ANS X9.19 Optional Procedure. A double-length key is required. This is the default value.

Table 163. Keywords for DK PIN Change control information (continued)

Keyword	Description
MAC Length and presentation (one optional, with keyword AES-CBC, TDES-CBC or TDES-ECB, otherwise not allowed.)	
MACLEN8	Specifies an 8-byte MAC. This is the default for TDES-CBC and TDES-ECB.
MACLEN16 (Release 4.4 or later)	Specifies a 16-byte MAC. Only valid with CMAC. This is the default for AES-CBC.

PAN_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PAN_data** parameter. The value must be in the range 10 - 19.

PAN_data

Direction: Input
Type: String

The PAN data to which the PIN is associated. The full account number, including check digit, should be included.

card_p_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **card_p_data** parameter. The value must be in the range 2 - 256.

card_p_data

Direction: Input
Type: String

The time-invariant card data (CDp), determined by the card issuer, which is used to differentiate between multiple cards for one account.

card_t_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **card_t_data** parameter. The value must be in the range 2 - 256.

card_t_data

Direction: Input
Type: String

The time-sensitive card data, determined by the card issuer, which, together with the account number and the **card_p_data**, specifies an individual card.

cur_ISO1_PIN_block_length

Direction: Input

Type: Integer

Specifies the length in bytes of the **cur_ISO1_PIN_block** parameter. The value must be 8.

cur_ISO1_PIN_block

Direction: Input
Type: String

The 8-byte encrypted PIN block with the current PIN in ISO-1 format.

new_ISO1_PIN_block_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **new_ISO1_PIN_block** parameter. The value must be 8.

new_ISO1_PIN_block

Direction: Input
Type: String

The new encrypted PIN block with the customer chosen PIN. The PIN block must be in ISO-1 format.

card_script_data_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the **card_script_data** variable. If the script selection of the rule array specifies to not return an encrypted SMPIN message with a PIN block MAC (that is, AES-CBC, TDES-CBC, or TDES-ECB is not specified), the value must be 0. Otherwise, set the value to a multiple of 16 and less than or equal to 4096 for AES_CBC, and set the value to a multiple of 8 and less than or equal to 4096 for TDES-CBC or TDES-ECB.

card_script_data

Direction: Input
Type: String

The clear text string to be updated with the clear PIN block and encrypted.

script_offset

Direction: Input
Type: Integer

The offset to the location for the PIN block in the script. Specify the first byte of the clear text as offset 0. This offset plus the value of the **script_offset_field_length** must be less than or equal to the **card_script_data_length**. If NOSCRIPT is specified in the rule array, this parameter is ignored.

script_offset_field_length

Direction: Input
Type: Integer

The length of the field within the **card_script_data** parameter at **script_offset** where the new PIN value is to be placed. Length must be 8. The PIN block must fit entirely within the **card_script_data**. If NOSCRIPT is specified in the rule array, this parameter is ignored.

script_initialization_vector_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the **script_initialization_vector** parameter. For script selection algorithm and method keyword AES-CBC the value must be 16, and for TDES-CBC the value must be 8. Otherwise, set the value to 0.

script_initialization_vector

Direction: Input
Type: String

a pointer to a string variable containing the initialization vector to use when encrypting the script in CBC mode. If the **script_initialization_vector_length** variable is 0 or if keyword TDES-ECB is specified, this parameter is ignored but must be declared. Otherwise, this parameter must point to a string of hexadecimal zeros.

output_PIN_profile

Direction: Input
Type: String

A 24-byte string containing the PIN profile, including the PIN block format for the script. See "The PIN profile" on page 449 for additional information. You can use PIN-block formats ISO-0, ISO-1, ISO-2, and ISO-3 with this service. If NOSCRIPT is specified in the rule array, this parameter is ignored.

PIN_reference_value_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PIN_reference_value** parameter. This value must be 16.

PIN_reference_value

Direction: Input
Type: String

The 16-byte PIN reference value of the current PIN for comparison to the calculated value.

PRW_random_number_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PRW_random_number** parameter. The value must be 4.

PRW_random_number

Direction: Input
Type: String

The 4-byte random number associated with the PIN reference value of the current PIN.

PRW_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PRW_key_identifier** parameter. If the **PRW_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

PRW_key_identifier

Direction: Input
Type: String

The identifier of the key to verify the PRW of the current PIN block. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINPRW, and the key usage fields must indicate VERIFY, CMAC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

cur_IPIN_encryption_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **cur_IPIN_encryption_key_identifier** parameter. If the **cur_IPIN_encryption_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

cur_IPIN_encryption_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to decrypt the PIN_block containing the current PIN. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be DES and the key type must be IPINENC.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

new_IPIN_encryption_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **new_IPIN_encryption_key_identifier** parameter. If the **new_IPIN_encryption_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

new_IPIN_encryption_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to decrypt the PIN_block containing the new PIN. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be DES and the key type must be IPINENC.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

script_key_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the **script_key_identifier** variable. If the **script_key_identifier** parameter identifies a key label, the length must be 64. Otherwise, for script selection algorithm and method keyword NOSCRIPT or its default, set the length to 0 or the length of a null key token, for AES-CBC set a maximum length of 725, and for TDES-CBC or TDES-ECB set the length to 64.

script_key_identifier

Direction: Input/Output
Type: String

A pointer to a string variable containing an operational fixed-length DES key-token or the key label of such a record in DES key-storage. Beginning with Release 4.4, it can be a pointer to a string variable containing an operational variable-length AES key-token or the key label of such a record in AES key-storage. The type of key depends on the script selection algorithm and method of the rule array:

- If AES-CBC is specified to return an AES-enciphered SMPIN message with a PIN block MAC, the key must be contained in a variable-length symmetric key-token. The key must have a token algorithm of AES and a key type of SECMSG. In addition, the key usage fields must enable the encryption of PINs in an EMV secure message (SMPIN), and must allow the key to be used by the CSNBDPC verb (ANY-USE or DPC-ONLY).
- If TDES-ECB or TDES-CBC) is specified to return a DES-enciphered SMPIN message with a PIN block MAC, the key must be contained in a fixed-length DES key token and have a key type of SECMSG. In addition, the control vector must enable the encryption of PINs (SMPIN bit 19 = B'1') .
- If NOSCRIPT or its default is specified to not return an enciphered SMPIN message with a PIN block MAC, the script_key_identifier_length variable should be set to 0. If the length is greater than 0, this parameter must identify a valid DES or, beginning with release 4.4, a valid AES key-token that is otherwise ignored.

script_MAC_key_identifier_length

Direction: Input
Type: Integer

The script_MAC_key_identifier_length parameter is a pointer to an integer variable containing the number of bytes of data in the **script_MAC_key_identifier** variable. If the **script_MAC_key_identifier** parameter identifies a key label, the length must be 64. Otherwise, for script selection algorithm and method keyword NOSCRIPT or its default, set the value to 0 or the length of a null key token, for AES-CBC set a maximum length of 725, and for TDES-CBC or TDES-ECB set the length to 64.

script_MAC_key_identifier

Direction: Input/Output
Type: String

A pointer to a string variable containing an operational fixed-length DES key-token or the key label of such a record in DES key-storage. Beginning with Release 4.4, it can be a pointer to a string variable containing an operational variable-length AES key-token or the key label of such a record in AES key-storage. The type of key depends on the script selection algorithm and method of the rule array:

- If AES-CBC is specified to return an AES-enciphered SMPIN message with a PIN block MAC, the key must be contained in a variable-length symmetric key-token. The key must have a token algorithm of AES and a key type of MAC. In addition, the key usage fields must have the MAC operation set so that the key can be used for generate (GENERATE or GENONLY), and the MAC mode must be CMAC.
- If TDES-ECB or TDES-CBC is specified to return a DES-enciphered SMPIN message with a PIN block MAC, the key must be double length and have a key type of MAC (generate is allowed). In addition, the control vector must have a subtype of ANY-MAC (bits 0-3 = B'0000').
- If NOSCRIPT or its default is specified to not return an enciphered SMPIN message with a PIN block MAC, the **script_key_identifier_length** variable should be set to 0. If the length is greater than 0, this parameter must identify a valid DES or, beginning with Release 4.4, a valid AES key-token that is otherwise ignored.

new_PRW_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **new_PRW_key_identifier** parameter. If the **new_PRW_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

new_PRW_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to verify the new PRW. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINPRW, and the key usage fields must indicate GENONLY, CMAC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

OPIN_encryption_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **OPIN_encryption_key_identifier** parameter. If the rule array indicates that no encrypted PIN block is to be returned, this value must be 0. If the **OPIN_encryption_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

OPIN_encryption_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to encrypt the new PIN block. The key identifier is an operational token or the key label of an operational token in key storage. If the OPIN_encryption_key_identifier_length is 0, this parameter is ignored. The key algorithm of this key must be AES, the key type must be PINPROT, and the key usage fields must indicate ENCRYPT, CBC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

OEPB_MAC_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **OEPB_MAC_key_identifier** parameter. If the rule array indicates that no encrypted PIN block is to be returned, this value must be 0. If the **OEPB_MAC_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

OEPB_MAC_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to generate the MAC of new PIN block. The key identifier is an operational token or the key label of an operational token in key storage. If the **OEPB_MAC_key_identifier_length** is 0, this parameter is ignored. The key algorithm of this key must be AES, the key type must be MAC, and the key usage fields must indicate CMAC, GENONLY, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

script_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the **script** variable. The value must be 0 if the script selection algorithm and method of the rule array specifies NOSCRIPT or its default. Otherwise, the value must be at least as long as the **card_script_data_length**.

script

Direction: Output
Type: String

The encrypted output script. The length of the field must be at least as long as the input script.

script_MAC_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the **script_MAC** variable. Set to 0 if script selection algorithm and method of the rule array specifies NOSCRIPT or its default. Otherwise, on input set the value to at least 8 for MAC length and presentation keyword MACLEN8, or at least

16 for MACLEN16. On output, the value is updated with the length of data returned in the **script_MAC** variable.

script_MAC

Direction: Output
Type: String

A pointer to a string variable containing the MAC calculated on the script returned in the **script** variable. The value is left-aligned in the variable and is truncated on the right as needed to the length specified by the MAC length and presentation keyword.

new_PIN_reference_value_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **new_PIN_reference_value** parameter. The value must be at least 16. On output, it is set to 16.

new_PIN_reference_value

Direction: Output
Type: String

The 16-byte new PIN reference value of the new PIN block.

new_PRW_random_number_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **new_PRW_random_number** parameter. The value must be at least 4. On output, it is set to 4.

new_PRW_random_number

Direction: Output
Type: String

The 4-byte random number associated with the new PIN reference value.

output_encrypted_PIN_block_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **output_encrypted_PIN_block** parameter. If the rule array indicates that no encrypted PIN block should be returned, this value must be 0. Otherwise, it should be at least 32. On output it is set to 32.

output_encrypted_PIN_block

Direction: Output
Type: String

The 32-byte encrypted new PIN block. If the **output_encrypted_PIN_block_length** is 0, this parameter is ignored.

PIN_block_MAC_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **PIN_block_MAC** parameter. If the rule_array indicates that no PIN block MAC should be returned, this value must be 0. Otherwise, it must be at least 8.

PIN_block_MAC

Direction: Output
Type: String

The 8-byte MAC of the new encrypted PIN block. If the PIN_block_MAC_length is 0, this parameter is ignored.

Restrictions

The restrictions for CSNBDPC.

The following rule array keywords are not supported in releases before Release 4.4:

- Script selection algorithm and method keyword AES-CBC
- MAC ciphering method keyword CMAC
- MAC length and presentation keyword MACLEN16

The **script_MAC_key_identifier** parameter and the **script_key_identifier** parameter cannot identify an AES variable-length symmetric key-token in releases before Release 4.4.

Required commands

The required commands for CSNBDPC.

The DK PIN Change verb requires the **DK PIN Change** command (offset X'02C2') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBDPC.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBDPCJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBDPCJ(
          hikmNativeInteger    return_code,
          hikmNativeInteger    reason_code,
          hikmNativeInteger    exit_data_length,
          byte[]               exit_data,
          hikmNativeInteger    rule_array_count,
          byte[]               rule_array,
          hikmNativeInteger    PAN_data_length,
          byte[]               PAN_data,
          hikmNativeInteger    card_p_data_length,
          byte[]               card_p_data,
          hikmNativeInteger    card_t_data_length,
```

```
byte[]              card_t_data,
hikmNativeInteger   cur_ISO1_PIN_block_length,
byte[]              cur_ISO1_PIN_block,
hikmNativeInteger   new_ISO1_PIN_block_length,
byte[]              new_ISO1_PIN_block,
hikmNativeInteger   card_script_data_length,
byte[]              card_script_data,
hikmNativeInteger   script_offset,
hikmNativeInteger   script_offset_field_length,
hikmNativeInteger   script_initialization_vector_length,
byte[]              script_initialization_vector,
byte[]              output_PIN_profile,
hikmNativeInteger   PIN_reference_value_length,
byte[]              PIN_reference_value,
hikmNativeInteger   PRW_random_number_length,
byte[]              PRW_random_number,
hikmNativeInteger   PRW_key_identifier_length,
byte[]              PRW_key_identifier,
hikmNativeInteger   current_IPIN_encryption_key_identifier_length,
byte[]              current_IPIN_encryption_key_identifier,
hikmNativeInteger   new_IPIN_encryption_key_identifier_length,
byte[]              new_IPIN_encryption_key_identifier,
hikmNativeInteger   script_key_identifier_length,
byte[]              script_key_identifier,
hikmNativeInteger   script_MAC_key_identifier_length,
byte[]              script_MAC_key_identifier,
hikmNativeInteger   new_PRW_key_identifier_length,
byte[]              new_PRW_key_identifier,
hikmNativeInteger   OPIN_encryption_key_identifier_length,
byte[]              OPIN_encryption_key_identifier,
hikmNativeInteger   OEPB_MAC_key_identifier_length,
byte[]              OEPB_MAC_key_identifier,
hikmNativeInteger   script_length,
byte[]              script,
hikmNativeInteger   script_MAC_length,
byte[]              script_MAC,
hikmNativeInteger   new_PIN_reference_value_length,
byte[]              new_PIN_reference_value,
hikmNativeInteger   new_PRW_random_number_length,
byte[]              new_PRW_random_number,
hikmNativeInteger   output_encrypted_PIN_block_length,
byte[]              output_encrypted_PIN_block,
hikmNativeInteger   PIN_block_MAC_length,
byte[]              PIN_block_MAC);
```

DK PIN Verify (CSNBDPV)

Use the DK PIN Verify verb to verify an ISO-1 format PIN in a transaction. The account, the card data, and the PRW are used to verify the PIN.

The input PIN is converted to PBF-0 format. A test PIN reference value (PRW) is created and that value is compared bit by bit to the input PRW.

Format

The format of CSNBDPV.

```
CSNBDPV(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             PAN_data_length,
             PAN_data,
             card_data_length,
             card_data,
             PIN_reference_value_length,
             PIN_reference_value,
             PRW_random_number_length,
             PRW_random_number,
             ISO_encrypted_PIN_block_length,
             ISO_encrypted_PIN_block,
             PRW_key_identifier_length,
             PRW_key_identifier,
             IPIN_encryption_key_identifier_length,
             IPIN_encryption_key_identifier)
```

Parameters

The parameters for CSNBDPV.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. This value must be 0.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the callable service. The keywords must be in contiguous storage with each of the keywords left-justified in its own 8-byte location and padded on the right with blanks. There are no keywords for this service.

PAN_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PAN_data** parameter. The value must be in the range 10 - 19.

PAN_data

Direction: Input
Type: String

The PAN data which the PIN is associated. The full account number, including check digit, should be included.

card_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **card_data** parameter. The value must be in the range 4 - 512.

card_data

Direction: Input
Type: String

The time-invariant card data (CDp) and the time-sensitive card data (CDt) which, together with the account number, specifies an individual card.

PIN_reference_value_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PIN_reference_value** parameter. This value must be 16.

PIN_reference_value

Direction: Input
Type: String

The 16-byte PIN reference value for comparison to the calculated value.

PRW_random_number_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PRW_random_number** parameter. The value must be 4.

PRW_random_number

Direction: Input
Type: String

The 4-byte random number associated with the PIN reference value.

ISO_encrypted_PIN_block_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **ISO_encrypted_PIN_block** parameter. This value must be 8.

ISO_encrypted_PIN_block

Direction: Input
Type: String

The 8-byte encrypted PIN block in ISO-1 format.

PRW_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PRW_key_identifier** parameter. If the **PRW_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

PRW_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to verify the PIN reference value. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINPRW, and the key usage fields must indicate VERIFY, CMAC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is be returned encrypted under the current master key.

IPIN_encryption_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **IPIN_encryption_key_identifier** parameter. If the **IPIN_encryption_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

IPIN_encryption_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to decrypt the PIN_block. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be DES and the key type must be IPINENC.

If the token supplied was encrypted under the old master key, the token is be returned encrypted under the current master key.

Restrictions

The restrictions for CSNBDPV.

None.

Required commands

The required commands for CSNBDPV.

The DK PIN Verify verb requires the **DK PIN Verify** command (offset X'02C1') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBDPV.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBDPVJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBDPVJ(
           hikmNativeInteger    return_code,
           hikmNativeInteger    reason_code,
           hikmNativeInteger    exit_data_length,
           byte[]               exit_data,
           hikmNativeInteger    rule_array_count,
           byte[]               rule_array,
           hikmNativeInteger    PAN_data_length,
           byte[]               PAN_data,
           hikmNativeInteger    card_data_length,
           byte[]               card_data,
           hikmNativeInteger    PIN_reference_value_length,
           byte[]               PIN_reference_value,
           hikmNativeInteger    PRW_random_number_length,
           byte[]               PRW_random_number,
           hikmNativeInteger    ISO_encrypted_PIN_block_length,
           byte[]               ISO_encrypted_PIN_block,
           hikmNativeInteger    PRW_key_identifier_length,
           byte[]               PRW_key_identifier,
           hikmNativeInteger    IPIN_encryption_key_identifier_length,
           byte[]               IPIN_encryption_key_identifier);
```

DK PRW Card Number Update (CSNBDPNU)

Use the DK PRW Card Number Update verb to generate a PIN reference value (PRW) when a replacement card is being issued. The original primary account number (PAN) and PIN are used with new time-sensitive card data to generate the new PRW.

You can use this service to perform the following tasks:

- Generate a PRW that can be used to verify the PIN.
- Optionally, generate an encrypted PIN block in PBF-1 format to be stored for later use in personalizing replacement cards.

Format

The format of CSNBDPNU.

```
CSNBDPNU(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             card_p_data_length,
             card_p_data,
             card_t_data_length,
             card_t_data,
             encrypted_PIN_block_length,
             encrypted_PIN_block,
             PIN_block_MAC_length,
             PIN_block_MAC,
             PRW_key_identifier_length,
             PRW_key_identifier,
             IPIN_encryption_key_identifier_length,
             IPIN_encryption_key_identifier,
             IEPB_MAC_key_identifier_length,
             IEPB_MAC_key_identifier,
             OPIN_encryption_key_identifier_length,
             OPIN_encryption_key_identifier,
             OEPB_MAC_key_identifier_length,
             OEPB_MAC_key_identifier,
             PIN_reference_value_length,
             PIN_reference_value,
             PRW_random_number_length,
             PRW_random_number,
             new_encrypted_PIN_block_length,
             new_encrypted_PIN_block,
             new_PIN_block_MAC_length,
             new_PIN_block_MAC)
```

Parameters

The parameters for CSNBDPNU.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. This value must be 0 or 1.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. The keywords are left-aligned in an 8-byte field and padded on the right with blanks. The keywords must be in contiguous storage. The **rule_array** keywords are described in Table 164 on page 603.

Table 164. Keywords for DK PRW Card Number Update control information

Keyword	Description
PIN Block output selection keyword (One, optional)	
EPB	Return an encrypted PIN block.
NOEPB	Do not return an encrypted PIN block (EPB). This is the default.

card_p_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **card_p_data** parameter. The value must be in the range 2 - 256.

card_p_data

Direction: Input
Type: String

The time-invariant card data (CDp), determined by the card issuer, which is used to differentiate between multiple cards for one account.

card_t_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **card_t_data** parameter. The value must be in the range 2 - 256.

card_t_data

Direction: Input
Type: String

The time-sensitive card data, determined by the card issuer, which, together with the account number and the **card_p_data**, specifies an individual card.

encrypted_PIN_block_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **encrypted_PIN_block** parameter. The value must be 32.

encrypted_PIN_block

Direction: Input
Type: String

The 32-byte input encrypted PIN block in PBF-1 format.

PIN_block_MAC_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PIN_block_MAC** parameter. The value must be 8.

PIN_block_MAC

Direction: Input
Type: String

The 8-byte CMAC of the encrypted PIN block.

PRW_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PRW_key_identifier** parameter. If the **PRW_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

PRW_key_identifier

Direction: Input/Output
Type: String

The identifier of the PRW generating key. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINPRW, the key usage fields must indicate GENONLY, CMAC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

IPIN_encryption_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **IPIN_encryption_key_identifier** parameter. If the **IPIN_encryption_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

IPIN_encryption_key_identifier

Direction: Input/Output
Type: String

The identifier of the key that encrypts the input PIN block. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINPROT, and the key usage fields must indicate DECRYPT, CBC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

IEPB_MAC_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **IEPB_MAC_key_identifier** parameter. If the **IEPB_MAC_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

IEPB_MAC_key_identifier

Direction: Input
Type: String

The identifier of the CMAC verification key. The key identifier is an

operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be MAC, and the key usage fields must indicate CMAC, VERIFY, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

OPIN_encryption_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **OPIN_encryption_key_identifier** parameter. If the **OPIN_encryption_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

OPIN_encryption_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to wrap the new PIN block. The key identifier is an operational token or the key label of an operational token in key storage. If the rule array indicates that no encrypted PIN block is to be returned, this value is ignored. The key algorithm of this key must be AES, the key type must be PINPROT, and the key usage fields must indicate ENCRYPT, CBC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

OEPB_MAC_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **OEPB_MAC_key_identifier** parameter. If the rule array indicates that no encrypted PIN block MAC is to be returned, this value must be 0.

If the **OEPB_MAC_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

OEPB_MAC_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to generate the CMAC of the new PRW. The key identifier is an operational token or the key label of an operational token in key storage. If the rule array indicates that no encrypted PIN block MAC is to be returned, this parameter is ignored. The key algorithm of this key must be AES, the key type must be MAC, and the key usage fields must indicate GENONLY, CMAC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

PIN_reference_value_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **PIN_reference_value** parameter. This value must be 16. On output, it is set to 16.

PIN_reference_value

Direction: Output
Type: String

The calculated 16-byte PIN reference value.

PRW_random_number_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **PRW_random_number** parameter. The value must be 4. On output, it is set to 4.

PRW_random_number

Direction: Output
Type: String

The 4-byte random number associated with the PIN reference value.

new_encrypted_PIN_block_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **new_encrypted_PIN_block** parameter. If the rule array indicates that no new encrypted PIN block should be returned, this parameter must be zero. Otherwise, the parameter should be at least 32.

new_encrypted_PIN_block

Direction: Output
Type: String

The new 32-byte encrypted PIN block. If the rule array indicates that no new encrypted PIN block should be returned, this parameter is ignored.

new_PIN_block_MAC_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **new_PIN_block_MAC** parameter. If the rule_array indicates that no new_PIN_block_MAC should be returned, this value must be zero. Otherwise, it must be at least 8.

new_PIN_block_MAC

Direction: Output
Type: String

The new 8-byte encrypted MAC of the new PIN block. If the rule array indicates that no new encrypted PIN block should be returned, this parameter is ignored.

Restrictions

The restrictions for CSNBDPNU.

None.

Required commands

The required commands for CSNBDPNU.

The DK PRW Card Number Update verb requires the **DK PRW Card Number Update** command (offset X'02C3') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBDPNU.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBDPNUJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBDPNUJ(
           hikmNativeInteger    return_code,
           hikmNativeInteger    reason_code,
           hikmNativeInteger    exit_data_length,
           byte[]               exit_data,
           hikmNativeInteger    rule_array_count,
           byte[]               rule_array,
           hikmNativeInteger    card_p_data_length,
           byte[]               card_p_data,
           hikmNativeInteger    card_t_data_length,
           byte[]               card_t_data,
           hikmNativeInteger    encrypted_PIN_block_length,
           byte[]               encrypted_PIN_block,
           hikmNativeInteger    PIN_block_MAC_length,
           byte[]               PIN_block_MAC,
           hikmNativeInteger    PRW_key_identifier_length,
           byte[]               PRW_key_identifier,
           hikmNativeInteger    IPIN_encryption_key_identifier_length,
           byte[]               IPIN_encryption_key_identifier,
           hikmNativeInteger    IEPB_MAC_key_identifier_length,
           byte[]               IEPB_MAC_key_identifier,
           hikmNativeInteger    OPIN_encryption_key_identifier_length,
           byte[]               OPIN_encryption_key_identifier,
           hikmNativeInteger    OEPB_MAC_key_identifier_length,
           byte[]               OEPB_MAC_key_identifier,
           hikmNativeInteger    PIN_reference_value_length,
           byte[]               PIN_reference_value,
           hikmNativeInteger    PRW_random_number_length,
           byte[]               PRW_random_number,
           hikmNativeInteger    new_encrypted_PIN_block_length,
           byte[]               new_encrypted_PIN_block,
           hikmNativeInteger    new_PIN_block_MAC_length,
           byte[]               new_PIN_block_MAC);
```

DK PRW CMAC Generate (CSNBDPCG)

Use the DK PRW CMAC Generate verb to generate a message authentication code (MAC) over specific values involved in an account number change transaction. The input includes the current and new PAN and card data and the PIN reference value.

The output of this service is used as input to the DK PAN Modify in Transaction callable service, which will create the new PIN reference value (PRW) to be used to verify the PIN.

Format

The format of CSNBDPCG.

```
CSNBDPCG(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             current_PAN_data_length,
             current_PAN_data,
             new_PAN_data_length,
             new_PAN_data,
             current_card_data_length,
             current_card_data,
             new_card_data_length,
             new_card_data,
             PIN_reference_value_length,
             PIN_reference_value,
             CMAC_FUS_key_identifier_length,
             CMAC_FUS_key_identifier,
             CMAC_FUS_length,
             CMAC_FUS)
```

Parameters

The parameters for CSNBDPCG.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. This value must be 0.

rule_array

Direction: Input
Type: String array

There are no keywords for this service.

current_PAN_data_length

Direction: Input

Type: Integer

A pointer to an integer variable containing the length in bytes of the **current_PAN_data** variable. This value must be in the range 10 - 19.

current_PAN_data

Direction: Input
Type: String

The current PAN data. The full account number, including check digit, should be included.

new_PAN_data_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the length in bytes of the **new_PAN_data** variable. This value must be in the range 10 - 19.

new_PAN_data

Direction: Input
Type: String

The new PAN data. The full account number, including check digit, should be included.

current_card_data_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the length in bytes of the **current_card_data** variable. This value must be in the range 4 - 512.

current_card_data

Direction: Input/Output
Type: String

The current card data, determined by the card issuer.

new_card_data_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the length in bytes of the **new_card_data** variable. This value must be in the range 4 - 512.

new_card_data

Direction: Input/Output
Type: String

The new card data, determined by the card issuer.

PIN_reference_value_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PIN_reference_value** parameter. This value must be 16. On output, **PIN_reference_value_length** is set to 16.

PIN_reference_value

Direction: Input
Type: String

The 16-byte PIN reference value of the current PIN.

CMAC_FUS_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **CMAC_FUS_key_identifier** variable. If the **CMAC_FUS_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

CMAC_FUS_key_identifier

Direction: Input
Type: String

The identifier of the key to generate the MAC. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be MAC, and the key usage fields must indicate GENONLY, CMAC, and DKPINAD2.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

CMAC_FUS_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable that specifies the length in bytes of the **CMAC_FUS** variable. This value must be in the range 8 - 16.

CMAC_FUS

Direction: Output
Type: String

The MAC of the current and new PANs and card data strings.

Restrictions

The restrictions for CSNBDPCG.

None.

Required commands

The required commands for CSNBDPCG.

The DK PRW CMAC Generate verb requires the **DK PRW CMAC Generate** command (offset X'02C4') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBDPCG.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBDPCGJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBDPCGJ(
           hikmNativeInteger    return_code,
           hikmNativeInteger    reason_code,
           hikmNativeInteger    exit_data_length,
           byte[]               exit_data,
           hikmNativeInteger    rule_array_count,
           byte[]               rule_array,
           hikmNativeInteger    current_PAN_data_length,
           byte[]               current_PAN_data,
           hikmNativeInteger    new_PAN_data_length,
           byte[]               new_PAN_data,
           hikmNativeInteger    current_card_data_length,
           byte[]               current_card_data,
           hikmNativeInteger    new_card_data_length,
           byte[]               new_card_data,
           hikmNativeInteger    PIN_reference_value_length,
           byte[]               PIN_reference_value,
           hikmNativeInteger    CMAC_FUS_key_identifier_length,
           byte[]               CMAC_FUS_key_identifier,
           hikmNativeInteger    CMAC_FUS_length,
           byte[]               CMAC_FUS);
```

DK Random PIN Generate (CSNBDRPG)

Use the DK Random PIN Generate verb to generate a PIN and a PIN reference value using the random process. After the PIN is generated, a PIN reference value (PRW) is created. The PIN reference value is used to verify the PIN in other processes. An optional encrypted PIN block is generated for printing.

Note: If the generated PIN appears in the weak PIN table, the generation process is modified and re-tried until a valid PIN is generated.

You can use this service to perform the following tasks:

- Generate an encrypted PIN block in PBF-1 format with a PIN print key to be printed on a PIN mailer.
- Generate a PIN reference value which can be used to verify the PIN.
- Optionally, generate an encrypted PIN block in PBF-1 format to be stored for later use in personalizing replacement cards, along with a verifying CMAC over the encrypted block and additional card data.

Format

The format of CSNBDRPG.

```
CSNBDRPG(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             PAN_data_length,
             PAN_data,
             card_p_data_length,
             card_p_data,
             card_t_data_length,
             card_t_data,
             PIN_length,
             PRW_key_identifier_length,
             PRW_key_identifier,
             PIN_print_key_identifier_length,
             PIN_print_key_identifier,
             OPIN_encryption_key_identifier_length,
             OPIN_encryption_key_identifier,
             OEPB_MAC_key_identifier_length,
             OEPB_MAC_key_identifier,
             PIN_reference_value_length,
             PIN_reference_value,
             PRW_random_number_length,
             PRW_random_number,
             PIN_print_block_length,
             PIN_print_block,
             encrypted_PIN_block_length,
             encrypted_PIN_block,
             PIN_block_MAC_length,
             PIN_block_MAC)
```

Parameters

The parameters for CSNBDRPG.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. This value must be 0 or 1.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. The keywords are left-aligned in an 8-byte field and padded on the right with blanks. The keywords must be in contiguous storage. The **rule_array** keywords are described in Table 165 on page 613.

Table 165. Keywords for DK Random PIN Generate control information

Keyword	Description
PIN Block output selection keyword (One, optional)	
EPB	Return an encrypted PIN block.
NOEPB	Do not return an encrypted PIN block (EPB). This is the default.

PAN_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PAN_data** parameter. The value must be in the range 10 - 19.

PAN_data

Direction: Input
Type: String

The personal account number in character form to which the PIN is associated. The primary account number, including check digit, should be included.

card_p_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **card_p_data** parameter. The value must be in the range 2 - 256.

card_p_data

Direction: Input
Type: String

The time-invariant card data (CDp), determined by the card issuer, which is used to differentiate between multiple cards for one account.

card_t_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **card_t_data** parameter. The value must be in the range 2 - 256.

card_t_data

Direction: Input
Type: String

The time-sensitive card data, determined by the card issuer, which, together with the account number and the **card_p_data**, specifies an individual card.

PIN_length

Direction: Input
Type: Integer

Specifies the length of the PIN to be generated. This value must be in the range 4 - 12.

PRW_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PRW_key_identifier** parameter. If the **PRW_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

PRW_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to verify the PRW of the current PIN block. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINPRW, and the key usage fields must indicate VERIFY, CMAC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

PIN_print_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PIN_print_key_identifier** parameter. If the **PIN_print_key_identifier** contains a label, the value must be 64. Otherwise, the value must be at least the actual token length, up to 725.

PIN_print_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to wrap the PIN for printing. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINPROT, and the key usage fields must indicate ENCRYPT, CBC, and DKPINOPP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

OPIN_encryption_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **OPIN_encryption_key_identifier** parameter. If the **OPIN_encryption_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

OPIN_encryption_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to wrap the new PIN block. The key identifier is an operational token or the key label of an operational token in key storage. If the rule array indicates that no encrypted PIN block is to be returned, this value is ignored. The key algorithm of this key must be AES, the key type must be

PINPROT, and the key usage fields must indicate ENCRYPT, CBC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

OEPB_MAC_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **OEPB_MAC_key_identifier** parameter. If the rule array indicates that no encrypted PIN block MAC is to be returned, this value must be 0.

If the **OEPB_MAC_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

OEPB_MAC_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to generate the CMAC of the new PRW. The key identifier is an operational token or the key label of an operational token in key storage. If the rule array indicates that no encrypted PIN block MAC is to be returned, this parameter is ignored. The key algorithm of this key must be AES, the key type must be MAC, and the key usage fields must indicate GENONLY, CMAC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

PIN_reference_value_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **PIN_reference_value** parameter. This value must be 16. On output, it is set to 16.

PIN_reference_value

Direction: Output
Type: String

The calculated 16-byte PIN reference value.

PRW_random_number_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **PRW_random_number** parameter. The value must be 4. On output, it is set to 4.

PRW_random_number

Direction: Output
Type: String

The 4-byte random number associated with the PIN reference value.

PIN_print_block_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **PIN_print_block** parameter. The value must be at least 32. On output, it is set to 32.

PIN_print_block

Direction: Output
Type: String

The 32-byte encrypted PIN block to be passed to the PIN mailer function.

encrypted_PIN_block_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **encrypted_PIN_block** parameter. If the rule array indicates that no encrypted PIN block should be returned, this value must be 0. Otherwise, it should be at least 32.

encrypted_PIN_block

Direction: Output
Type: String

The 32-byte encrypted PIN block in PBF-1 format. This parameter is ignored if no encrypted PIN block is returned.

PIN_block_MAC_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **PIN_block_MAC** parameter. If the rule_array indicates that no PIN block MAC should be returned, this value must be 0. Otherwise, it must be at least 8.

PIN_block_MAC

Direction: Output
Type: String

The 8-byte CMAC of the encrypted PIN block. This parameter is ignored if no encrypted PIN block is returned.

Restrictions

The restrictions for CSNBDRPG.

None.

Required commands

The required commands for CSNBDRPG.

The DK Random PIN Generate verb requires the **DK Random PIN Generate** command (offset X'02C0') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBDRPG.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBDRPGJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBDRPGJ(
           hikmNativeInteger    return_code,
           hikmNativeInteger    reason_code,
           hikmNativeInteger    exit_data_length,
           byte[]               exit_data,
           hikmNativeInteger    rule_array_count,
           byte[]               rule_array,
           hikmNativeInteger    PAN_data_length,
           byte[]               PAN_data,
           hikmNativeInteger    card_p_data_length,
           byte[]               card_p_data,
           hikmNativeInteger    card_t_data_length,
           byte[]               card_t_data,
           hikmNativeInteger    PIN_length,
           hikmNativeInteger    PRW_key_identifier_length,
           byte[]               PRW_key_identifier,
           hikmNativeInteger    PIN_print_key_identifier_length,
           byte[]               PIN_print_key_identifier,
           hikmNativeInteger    OPIN_encryption_key_identifier_length,
           byte[]               OPIN_encryption_key_identifier,
           hikmNativeInteger    OEPB_MAC_key_identifier_length,
           byte[]               OEPB_MAC_key_identifier,
           hikmNativeInteger    PIN_reference_value_length,
           byte[]               PIN_reference_value,
           hikmNativeInteger    PRW_random_number_length,
           byte[]               PRW_random_number,
           hikmNativeInteger    PIN_print_block_length,
           byte[]               PIN_print_block,
           hikmNativeInteger    encrypted_PIN_block_length,
           byte[]               encrypted_PIN_block,
           hikmNativeInteger    PIN_block_MAC_length,
           byte[]               PIN_block_MAC);
```

DK Regenerate PRW (CSNBDRP)

The DK Regenerate PRW verb generates a new PIN reference value for a changed account number.

You can use this service to perform the following tasks:

- Generate a PIN reference value over the existing PIN and new PAN, which can be used to verify transactions.
- Generate an encrypted PIN block in PBF-1 format to be stored for later use in personalization of smart cards.

Format

The format of CSNBDRP.

```
CSNBDRP(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             card_p_data_length,
             card_p_data,
             card_t_data_length,
             card_t_data,
             encrypted_PIN_block_length,
             encrypted_PIN_block,
             PIN_block_MAC_length,
             PIN_block_MAC,
             PRW_key_identifier_length,
             PRW_key_identifier,
             IPIN_encryption_key_identifier_length,
             IPIN_encryption_key_identifier,
             IEPB_MAC_key_identifier_length,
             IEPB_MAC_key_identifier,
             OPIN_encryption_key_identifier_length,
             OPIN_encryption_key_identifier,
             OEPB_MAC_key_identifier_length,
             OEPB_MAC_key_identifier,
             PIN_reference_value_length,
             PIN_reference_value,
             PRW_random_number_length,
             PRW_random_number,
             new_encrypted_PIN_block_length,
             new_encrypted_PIN_block,
             new_PIN_block_MAC_length,
             new_PIN_block_MAC)
```

Parameters

The parameters for CSNBDRP.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: String array

A pointer to an integer variable containing the number of elements in the **rule_array** variable. This value must be 0.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. The keywords are left-aligned in an 8-byte field and padded on the right with blanks. The keywords must be in contiguous storage. There are no keywords for this service.

card_p_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **card_p_data** parameter. The value must be in the range 2 - 256.

card_p_data

Direction: Input
Type: String

The time-invariant card data (CDp), determined by the card issuer, which is used to differentiate between multiple cards for one account.

card_t_data_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **card_t_data** parameter. The value must be in the range 2 - 256.

card_t_data

Direction: Input
Type: String

The time-sensitive card data, determined by the card issuer, which, together with the account number and the **card_p_data**, specifies an individual card.

encrypted_PIN_block_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **encrypted_PIN_block** parameter. If the rule array indicates that no encrypted PIN block should be returned, this value must be 0. Otherwise, it should be at least 32.

encrypted_PIN_block

Direction: Input
Type: String

The 32-byte encrypted PIN block in PBF-1 format. This parameter is ignored if no encrypted PIN block is returned.

PIN_block_MAC_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **PIN_block_MAC** parameter. If the rule array indicates that no PIN block MAC should be returned, this value must be 0. Otherwise, it must be at least 8.

PIN_block_MAC

Direction: Input
Type: String

The 8-byte CMAC of the encrypted PIN block. This parameter is ignored if no encrypted PIN block is returned.

PRW_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PRW_key_identifier** parameter. If **PRW_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

PRW_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to verify the PRW of the current PIN block. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINPRW, and the key usage fields must indicate VERIFY, CMAC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

IPIN_encryption_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **IPIN_encryption_key_identifier** parameter. If **IPIN_encryption_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

IPIN_encryption_key_identifier

Direction: Input
Type: String

The identifier of the key to decrypt the PIN block containing the current PIN. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINPROT, and the key usage fields must indicate DECRYPT, CBC, and DKPINAD1.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

IEPB_MAC_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **IEPB_MAC_key_identifier** parameter. If **IEPB_MAC_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

IEPB_MAC_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to verify MAC of the inbound encrypted PIN block. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type

must be MAC, and the key usage fields must indicate CMAC. VERIFY, and DKPINAD1.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

OPIN_encryption_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **OPIN_encryption_key_identifier** parameter. If **OPIN_encryption_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

OPIN_encryption_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to encrypt the new PIN block. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be PINPROT, and the key usage fields must indicate ENCRYPT, CBC, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

OEPB_MAC_key_identifier_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **OEPB_MAC_key_identifier** parameter. If **OEPB_MAC_key_identifier** contains a label, the length must be 64. Otherwise, the value must be at least the actual token length, up to 725.

OEPB_MAC_key_identifier

Direction: Input/Output
Type: String

The identifier of the key to generate the MAC of the new encrypted PIN block. The key identifier is an operational token or the key label of an operational token in key storage. The key algorithm of this key must be AES, the key type must be MAC, and the key usage fields must indicate CMAC, GENONLY, and DKPINOP.

If the token supplied was encrypted under the old master key, the token is returned encrypted under the current master key.

PIN_reference_value_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PIN_reference_value** parameter. The value must be 16. On output, it is set to 16.

PIN_reference_value

Direction: Input
Type: String

The 16-byte PIN reference value for comparison to the calculated value.

PRW_random_number_length

Direction: Input
Type: Integer

Specifies the length in bytes of the **PRW_random_number** parameter. The value must be 4. On output, it is set to 4.

PRW_random_number

Direction: Input
Type: String

The 4-byte random number associated with the PIN reference value.

new_encrypted_PIN_block_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **new_encrypted_PIN_block** parameter. The value must be at least 32. On output, it is set to 32.

new_encrypted_PIN_block

Direction: Output
Type: String

The 32-byte encrypted new PIN block.

new_PIN_block_MAC_length

Direction: Input/Output
Type: Integer

Specifies the length in bytes of the **new_PIN_block_MAC** parameter. The value must be at least 8.

new_PIN_block_MAC

Direction: Output
Type: String

The 8-byte MAC of the new encrypted PIN block.

Restrictions

The restrictions for CSNBDRP.

None.

Required commands

The required commands for CSNBDRP.

The DK Regenerate PRW verb requires the **DK Regenerate PRW** command (offset X'02C8') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBDRP.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBDRPJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBDRPJ(
           hikmNativeInteger    return_code,
           hikmNativeInteger    reason_code,
           hikmNativeInteger    exit_data_length,
           byte[]               exit_data,
           hikmNativeInteger    rule_array_count,
           byte[]               rule_array,
           hikmNativeInteger    card_p_data_length,
           byte[]               card_p_data,
           hikmNativeInteger    card_t_data_length,
           byte[]               card_t_data,
           hikmNativeInteger    encrypted_PIN_block_length,
           byte[]               encrypted_PIN_block,
           hikmNativeInteger    PIN_block_MAC_length,
           byte[]               PIN_block_MAC,
           hikmNativeInteger    PRW_key_identifier_length,
           byte[]               PRW_key_identifier,
           hikmNativeInteger    IPIN_encryption_key_identifier_length,
           byte[]               IPIN_encryption_key_identifier,
           hikmNativeInteger    IEPB_MAC_key_identifier_length,
           byte[]               IEPB_MAC_key_identifier,
           hikmNativeInteger    OPIN_encryption_key_identifier_length,
           byte[]               OPIN_encryption_key_identifier,
           hikmNativeInteger    OEPB_MAC_key_identifier_length,
           byte[]               OEPB_MAC_key_identifier,
           hikmNativeInteger    PIN_reference_value_length,
           byte[]               PIN_reference_value,
           hikmNativeInteger    PRW_random_number_length,
           byte[]               PRW_random_number,
           hikmNativeInteger    new_encrypted_PIN_block_length,
           byte[]               new_encrypted_PIN_block,
           hikmNativeInteger    new_PIN_block_MAC_length,
           byte[]               new_PIN_block_MAC);
```

Chapter 12. Using digital signatures

Use the CCA verbs described in this topic to support digital signatures to authenticate messages.

Digital Signature Generate (CSNDDSG)

This verb generates a digital signature using an RSA or ECC private key.

This verb supports the following methods:

- ANSI X9.30 (ECDSA)
- ANSI X9.31 (RSA)
- ISO 9796-1 (RSA)
- RSA DSI PKCS 1.0 and 1.1 (RSA)
- Padding on the left with zeros (RSA)

Note: The maximum signature length is 512 bytes (4096 bits).

The input text should have been previously hashed using either the One-Way Hash verb or the MDC Generate verb. If the signature formatting algorithm specifies ANSI X9.31, you must specify the hash algorithm used to hash the text (**SHA-1** or **RPMD-160**). See "Formatting hashes and keys in public-key cryptography" on page 950.

You select the method of formatting the text through the *rule_array* parameter.

If the *PKA_private_key_identifier* specifies an RSA private key, you select the method of formatting the text through the *rule_array* parameter. If the *PKA_private_key_identifier* specifies an ECC private key, the ECC signature generated is according to ANSI X9.30.

Note: For PKCS the message digest and the message-digest algorithm identifier are combined into an ASN.1 value of type DigestInfo, which is BER-encoded to give an octet string D (see Table 166 on page 626). D is the text string supplied in the *hash* variable.

Format

The format of CSNDDSG.

```
CSNDDSG(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             PKA_private_key_identifier_length,
             PKA_private_key_identifier,
             hash_length,
             hash,
             signature_field_length,
             signature_bit_length,
             signature_field)
```

Parameters

The parameter definitions for CSNDDSG.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0, 1, 2, or 3.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. A keyword specifies the method for calculating the digital signature. Each keyword is left-aligned in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage. The *rule_array* keywords are described in Table 166.

Table 166. Keywords for Digital Signature Generate control information

Keyword	Description
Digital signature formatting method (One, optional and not valid with **ECDSA** keyword.)	
ISO-9796	Calculate the digital signature on the *hash* according to ISO-9796-1. Any hash method is allowed. This is the default.
PKCS-1.0	Calculate the digital signature on the BER-encoded ASN.1 value of the type DigestInfo containing the hash according to the RSA Data Security, Inc. Public Key Cryptography Standards #1 block type 00. The text must have been hashed and BER-encoded before input to this service.
PKCS-1.1	Calculate the digital signature on the BER-encoded ASN.1 value of the type DigestInfo containing the hash according to the RSA Data Security, Inc. Public Key Cryptography Standards #1 block type 01. The text must have been hashed and BER-encoded before input to this service.
ZERO-PAD	Format the hash by padding it on the left with binary zeros to the length of the RSA key modulus. Any supported hash function is allowed.

Table 166. Keywords for Digital Signature Generate control information (continued)

Keyword	Description
X9.31	Format according to the ANSI X9.31 standard. The input text must have been previously hashed with one of the hash algorithms specified below.
Hash method specification (One, optional. Valid only with **X9.31** digital-signature hash formatting method.)	
RPMD-160	Hash the input text using the RIPEMD-160 hash method.
SHA-1	Hash the input text using the SHA-1 hash method.
SHA-256	Hash the input text using the SHA-256 hash method.
SHA-384	Hash the input text using the SHA-384 hash method.
SHA-512	Hash the input text using the SHA-512 hash method.
Token algorithm (One, optional)	
ECDSA	Generate an ECC digital signature. This keyword was introduced with CCA 4.1.0. When specified, this is the only keyword permitted in the *rule_array*.
RSA	Generate an RSA digital signature. This is the default.

PKA_private_key_identifier_length

Direction: Input
Type: Integer

The length of the *PKA_private_key_identifier* field. The maximum size is 3500 bytes.

PKA_private_key_identifier

Direction: Input
Type: String

An internal token or label of the RSA private key or retained key. If the signature format is **X9.31**, the modulus of the RSA key must have a minimum length of 1024 bits or greater. If the signature algorithm is **ECDSA**, this parameter must be a token or label of an ECC private key.

hash_length

Direction: Input
Type: Integer

The length of the *hash* parameter in bytes. It must be the exact length of the text to sign. The maximum size is 512 bytes. If you specify **ZERO-PAD** in the *rule_array* parameter, the length is restricted to 36 bytes unless the RSA key is a signature only key, then the maximum length is 512 bytes.

On the IBM eServer zSeries 990 and subsequent releases, the hash length limit is controlled by a new access control point. Only RSA key management keys are affected by this access control point. The limit for RSA signature use only keys is 512 bytes. This new access control point is always disabled in the default role. You must have a TKE workstation to enable it.

hash

Direction: Input
Type: String

The application-supplied text on which to generate the signature. The input text must have been previously hashed, and for PKCS formatting, it must be BER-encoded as previously described. For **X9.31**, the hash algorithms must

have been either **SHA-1** or **RPMD-160**. See the *rule_array* parameter for more information.

signature_field_length

Direction: Input/Output
Type: Integer

The length in bytes of the *signature_field* to contain the generated digital signature. The maximum size is 512 bytes.

For RSA, this must be at least the RSA modulus size (rounded up to a multiple of 32 bytes for the **X9.31** signature format, or one byte for all other signature formats).

For RSA, this field is updated with the minimum byte length of the digital signature.

For the **ECDSA** signature algorithm, R concatenated with S is the digital signature. The maximum output value will be 1042 bits (131 bytes). The size of the signature is determined by the size of P. Both R and S will have size P. For prime curves, the maximum size is 2 * 521 bits. For Brainpool curves, the maximum size is 2 * 512 bits.

signature_bit_length

Direction: Output
Type: Integer

The bit length of the digital signature generated. For **ISO-9796** this is 1 less than the modulus length. For other RSA processing methods, this is the modulus length.

signature_field

Direction: Output
Type: String

The digital signature generated is returned in this field. The digital signature is in the low-order bits (right-aligned) of a string whose length is the minimum number of bytes that can contain the digital signature. This string is left-aligned within the *signature_field*. Any unused bytes to the right are undefined.

Restrictions

The restrictions for CSNDDSG.

Although **ISO-9796** does not require the input hash to be an integral number of bytes in length, this verb requires you to specify the *hash_length* in bytes.

X9.31 requires the RSA token to have a minimum modulus bit length of 1024 bits, and the length must also be a multiple of 256 bits (or 32 bytes).

The length of the *hash* parameter in bytes must be the exact length of the text to sign. The maximum size is 256 bytes. If you specify **ZERO-PAD** in the *rule_array* parameter, the length is restricted to 36 bytes unless the RSA key is a signature only key, then the maximum length is 256 bytes.

The hash length limit is controlled by an access control point. If OFF (disabled), the maximum hash length limit for **ZERO-PAD** is the modulus length of the PKA private key. If ON (enabled), the maximum hash length limit for **ZERO-PAD** is 36

bytes. Only RSA key management keys are affected by this access control point. The limit for RSA signature use only keys is 256 bytes. This new access control point is always disabled in the default role. You must have a TKE workstation to enable it.

Required commands

The required commands for CSNDDSG.

This verb requires the **Digital Signature Generate** command (offset X'0100') to be enabled in the active role.

With the use of the **DSG ZERO-PAD unrestricted hash length** command (offset X'030C'), the hash-length restriction does not apply when using **ZERO-PAD** formatting.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNDDSG.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDDSGJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNDDSGJ)
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger PKA_private_key_identifier_length,
           byte[]            PKA_private_key_identifier,
           hikmNativeInteger hash_length,
           byte[]            hash,
           hikmNativeInteger signature_field_length,
           hikmNativeInteger signature_bit_length,
           byte[]            signature_field);
```

Digital Signature Verify (CSNDDSV)

This verb verifies a digital signature using an RSA or ECC public key.

This verb verifies digital signatures generated with these methods:

- ANSI X9.30 (ECDSA)
- ANSI X9.31 (RSA)
- ISO 9796-1 (RSA)
- RSA DSI PKCS 1.0 and 1.1 (RSA)
- Padding on the left with zeros (RSA)

This verb can use the RSA or ECC public key, depending on the digital signature algorithm used to generate the signature.

This verb can also use the public keys that are contained in trusted blocks, regardless of whether the block also contains rules to govern its use when generating or exporting keys with the Remote Key Export verb. The format of the trusted block enables Digital Signature Verify to distinguish it from other RSA key tokens, and therefore no special rule array keyword or other parameters are required in order to indicate that the trusted block is being used. However, if the Digital Signature Generate verb is used with the **TPK-ONLY** keyword in the *rule_array*, an error will occur if the *PKA_public_key_identifier* does not contain a trusted block.

Input text should have been previously hashed. You can use the One-Way Hash verb. See also "Formatting hashes and keys in public-key cryptography" on page 950.

Note: The maximum signature length is 256 bytes (2048 bits).

EC signature verification update

Beginning with CCA Release 5.0 of CSNDDSV for the CEX5C, checking of EC signatures is strengthened with new hardware support. For most invalid signatures the typical response of return code 4, reason code 429 is still returned. However, for some cases where the signature value ***r*** or value ***s*** is mathematically off the curve described by the ***q*** value from the EC key, that is, when `r > (q-1) or s > (q-1)`, then return code 12 with reason code 769 is returned, indicating a rejection from the EC hardware layer. The decision to return the different error for the more serious case is reached when the host library loads (at application startup) by the value of a new environment variable: `CSU_EC_CHECKCURVE`

For appropriate failure cases, if the value of `CSU_EC_CHECKCURVE` is 1, then the new return/reason code 12/769 is returned. Otherwise, 4/429 is returned. The default value of `CSU_EC_CHECKCURVE` is 0, to maintain compatibility with prior releases. IBM recommends that customers prepare their applications and set the new environment variable to 1. Use the following command to set the variable (also to set it in a profile):

```
export CSU_EC_CHECKCURVE=1
```

Some error path test cases with pre-figured values may need to be updated, and the new return/reason code handling may need to be added to your application as a different type of signature verification failure. The adapter is functioning normally and no service action is required. Note that this change does not narrow valid verification cases or add restrictions. All valid signatures when passed to CSNDDSV with the correct key still verify with return code of 0, reason code of 0.

Advantages of the new verification granularity

The new use of return code/reason code 12/769 can help you with problem determination for the following practical cases:

- Check the bytes of the ECDSA signature. The ***r*** value is mathematically determined from the *order of G* of the curve and a random number. If the value of ***r*** or the value of ***s*** equals or exceeds the *order of G* of the curve, then the signature has been corrupted. No value or range of values can be statically ruled out for any bytes of ***r*** or ***s*** because of the random components. However you

may be able to detect a corruption pattern by inspection. The ***s*** value is the same byte length as ***r*** (which is the same length as the curve *order of G*). The ***s*** value immediately follows ***r*** in the signature.

- Certain types of key corruption may also cause this error - forcing the key to be off the curve. Since public keys are passed in the clear, this is hard to detect until the public key is actually used, but again inspection of the key data may help.
- While using the wrong key type on the same curve size (example: substituting a P384 key for a BP384 key) will probably not generate 12/769 instead of 4/429, using a key for a smaller curve may generate 12/769 (example: substituting a P224 key for a P256 key).

Format

The format of CSNDDSV.

```
CSNDDSV(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              rule_array_count,
              rule_array,
              PKA_public_key_identifier_length,
              PKA_public_key_identifier,
              hash_length,
              hash,
              signature_field_length,
              signature_field)
```

Parameters

The parameters for CSNDDSV.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0, 1, 2, or 3.

rule_array

Direction: Input
Type: String array

Keywords that provide control information to the verb. A keyword specifies the method to use to verify the digital signature. Each keyword is left-aligned in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage. The *rule_array* keywords are described in Table 167.

Table 167. Keywords for Digital Signature Verify control information

Keyword	Description
Digital signature formatting method (Optional and not valid with **ECDSA** keyword.)	

Table 167. Keywords for Digital Signature Verify control information (continued)

Keyword	Description
ISO-9796	Verify the digital signature on the hash according to ISO-9796-1. Any hash method is allowed. This is the default.
PKCS-1.0	Verify the digital signature on the BER-encoded ASN.1 value of the type DigestInfo as specified in the RSA Data Security, Inc. Public Key Cryptography Standards #1 block type 00. The text must specify BER encoded hash text.
PKCS-1.1	Verify the digital signature on the BER-encoded ASN.1 value of the type DigestInfo as specified in the RSA Data Security, Inc. Public Key Cryptography Standards #1 block type 01. The text must specify BER encoded hash text.
ZERO-PAD	Format the hash by padding it on the left with binary zeros to the length of the PKA key modulus. Any supported hash function is allowed.
X9.31	Format according to ANSI X9.31 standard.
Trusted public key restriction (Optional. Not valid with **ECDSA** keyword. Valid only with trusted blocks. See "Trusted blocks" on page 877.)	
TPK-ONLY	Permits the use of only public keys contained in trusted blocks. By specifying this keyword, the use of regular CCA RSA key tokens is rejected and only the use of a (trusted) public key supplied by the *PKA_public_key_identifier* parameter can be used to verify the digital signature, thus assuring a sensitive signature verification operation is limited to trusted public keys. If **TPK-ONLY** is specified, the *PKA_public_key_identifier* parameter must identify a trusted block that contains two sections after the trusted block token header: (1) trusted block trusted RSA public key (section X'11'), and (2) trusted block information (section X'14'). Section X'14' is required for all trusted blocks. Section X'11' contains the trusted public key, and its usage rules must indicate it can be used in digital signature operations.
Token algorithm (One, optional)	
ECDSA	Verify an ECC digital signature. This keyword was introduced with CCA 4.1.0. When specified, this is the only keyword permitted in the *rule_array*.
RSA	Verify an RSA digital signature. This is the default. This keyword was introduced with CCA 4.1.0.

PKA_public_key_identifier_length

Direction: Input
Type: Integer

The length of the *PKA_public_key_identifier* field containing the public key token or label. The maximum size is 3500 bytes.

PKA_public_key_identifier

Direction: Input
Type: String

A token or label of the RSA public key or internal trusted block. If this parameter contains a token or the label of an internal trusted block, the *rule_array* parameter must specify **TPK-ONLY**. If the signature algorithm is **ECDSA**, this must be a token label or an ECC public key.

hash_length

Direction: Input
Type: Integer

The length of the *hash* parameter in bytes. It must be the exact length of the text that was signed. The maximum size is 512 bytes.

hash

Direction: Input
Type: String

The application-supplied text on which the supplied signature was generated. The text must have been previously hashed and, for PKCS formatting, BER-encoded as previously described.

signature_field_length

Direction: Input
Type: Integer

The length in bytes of the *signature_field* parameter. The maximum size is 512 bytes.

signature_field

Direction: Input
Type: String

This field contains the digital signature to verify. The digital signature is in the low-order bits (right-aligned) of a string whose length is the minimum number of bytes that can contain the digital signature. This string is left-aligned within the *signature_field*.

Restrictions

The restrictions for CSNDDSV.

The ability to recover a message from a signature (which ISO-9796 allows but does not require) is **not** supported.

The exponent of the RSA public key must be odd.

Although ISO-9796 does not require the input hash to be an integral number of bytes in length, this service requires you to specify the *hash_length* in bytes.

X9.31 requires the RSA token to have a minimum modulus bit length of 1024, and the length must also be a multiple of 256 bits (or 32 bytes).

Required commands

The required commands for CSNDDSV.

This verb requires the **Digital Signature Verify** command (offset X'0101') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNDDSV.

None.

Related information

Additional information for CSNDDSV.

Trusted Block Create (CSNDTBC), Remote Key Export (CSNDRKX)

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDDSVJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNDDSVJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger PKA_public_key_identifier_length,
           byte[]            PKA_public_key_identifier,
           hikmNativeInteger hash_length,
           byte[]            hash,
           hikmNativeInteger signature_field_length,
           byte[]            signature_field);
```

Chapter 13. Managing PKA cryptographic keys

Use these verbs to generate and manage PKA keys.

PKA Key Generate (CSNDPKG)

Use the PKA Key Generate verb to generate RSA keys for use on the cryptographic coprocessor or other CCA systems, or ECC keys for use starting with CEX3C.

Input to the PKA Key Generate verb is either a skeleton key token that has been built by the PKA Key Token Build verb, or a valid internal token. In the case of a valid internal token, the verb will generate a key with the same modulus length and the same exponent. In the case of a valid internal ECC token, PKA Key Generate will generate a key based on the curve type and size. Internal tokens with a X'09' section are not supported.

RSA key generation requires the following information in the input skeleton token:

- Size of the modulus in bits. The modulus for Modulus-Exponent format keys is between 512 and 1024 bits in length. The CRT modulus is between 512 and 4096 bits in length. The modulus for the variable-length Modulus-Exponent format is between 512 and 4096 bits in length.

RSA key generation has the following restrictions:

- For Modulus-Exponent, there are restrictions on the modulus, public exponent, and private exponent.
- For CRT, there are restrictions on *dp*, *dq*, *U*, and the public exponent.

See the Key value structure in "PKA Key Token Build (CSNDPKB)" on page 644 for a summary of restrictions.

ECC key generation requires this information in the skeleton token:

- The key type: ECC
- The type of curve: Prime or Brainpool
- The size of *p* in bits: 192, 224, 256, 384 or 521 for Prime curves and 160, 192, 224, 256, 320, 384, or 512 for Brainpool curves
- Key usage information
- Optionally, application associated data

The generated ECC private key will be returned in one of the following forms:

- Clear key
- Encrypted key enciphered under the APKA-MK

- Encrypted key enciphered by an AES transport key

Format

The format of CSNDPKG.

```
CSNDPKG(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             regeneration_data_length,
             regeneration_data,
             skeleton_key_identifier_length,
             skeleton_key_identifier,
             transport_key_identifier,
             generated_key_token_length,
             generated_key_token)
```

Parameters

The parameter definitions for CSNDPKG.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1, 2, or 3.

rule_array

Direction: Input
Type: String array

A keyword that provides control information to the verb. A keyword is left-aligned in an 8-byte field and padded on the right with blanks. The *rule_array* keywords are described in Table 168.

Table 168. Keywords for PKA Key Generate control information

Keyword	Description
Private key encryption (One, required)	
CLEAR	Return the private key in clear text. The private key in clear text is an external token. This keyword is valid only for RSA and ECC keys.
MASTER	Encipher the private key or OPK using the PKA master-key for an RSA key, or the OPK using the APKA master-key for an ECC key. The *transport_key_identifier* parameter should specify a null key-token. The keyword is not supported if a skeleton token with a **09** section is provided.

Table 168. Keywords for PKA Key Generate control information (continued)

Keyword	Description
RETAIN	Retains the private key within the cryptographic engine and returns the public key. This is only valid for RSA signature keys. Because of this, the RETAIN keyword is not supported for: • A skeleton token with a X'09' section provided. • An ECC token. Before using this keyword, see the information about retained keys in "Using retained keys" on page 438. **Note:** Take special notice on the types of skeleton key tokens that can be passed. The PKA Key Token Build verb will, of course, let you create many more types of skeleton key tokens than can be used to generate retained keys, because this is the minority of supported function.
XPORT	Enciphers the private key under the IMPORTER or EXPORTER key-encrypting-key identified by the *transport_key_identifier* parameter. For an RSA key, this is an EXPORTER or IMPORTER transport key in a fixed-length operational DES key-token. For an ECC key, this is an EXPORTER or IMPORTER transport key in an operational variable-length AES key-token. This keyword is valid only for RSA and ECC keys.
RETAIN option (one, optional). Valid only with the **RETAIN** keyword.	
CLONE	Mark a generated and retained private key as usable in cryptographic engine cloning process. This keyword is supported only if **RETAIN** is also specified. Only valid for RSA keys. The keyword is not supported for: • A skeleton token with a X'09' section provided • An ECC token
Regeneration data option (One, optional)	
ITER-38	Force 38 iterations of tests for primality, as required by ANSI X9.31 for the Miller-Rabin primality tests. This option produces a more secure key, but it is labor intensive. This keyword is invalid for ECC key generation. This keyword was introduced with CCA 4.1.0.
Transport key-type (one, optional; one required if *transport_key_identifier* is a label). If this keyword is specified, it must match the type of key to be transported, whether the identifier is a label or not.	
OKEK-AES	The outbound key-encrypting key represents an AES key-token.
OKEK-DES	The outbound key-encrypting key represents a DES key-token. This is the default.

regeneration_data_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the **regeneration_data** variable. This parameter must be 0 for ECC tokens. For RSA tokens, the value must be 8 - 512.

If the value is 0, the generated keys are based on a random-seed value. If this value is between 8 - 256, the regeneration data is hashed to form a seed value used in the key generation process to provide a means for recreating a public-private key pair.

regeneration_data

Direction: Input
Type: String

This field points to a string variable containing a string used as the basis for creating a particular public-private key pair in a repeatable manner. The regeneration data is hashed to form a seed value used in the key generation process and provides a means for recreating a public-private key pair.

skeleton_key_identifier_length

Direction: Input
Type: Integer

The length of the **skeleton_key_identifier** parameter in bytes. The maximum allowed value is 3500 bytes.

skeleton_key_identifier

Direction: Input
Type: String

A pointer to the application-supplied skeleton key token generated by PKA Key Token Build, or the label of the token that contains the required modulus length and public exponent for RSA key generation, or the required curve type and bit length for ECC key generation.

If RETAIN was specified and the *skeleton_key_identifier* is a label, the label must match the private key name of the key. For RSA keys, the *skeleton_key_identifier* parameter must contain a token that specifies a modulus length in the range 512 - 4096 bits.

transport_key_identifier

Direction: Input
Type: String

A pointer to a string variable containing an operational AES or DES key-encrypting-key token, a null key-token, or a key label of such a key. Use an **IMPORTER** key to encipher a private key to be used at this node. Use an **EXPORTER** key to encipher a private key to be used at another node. Choose one of the following:

- When generating an ECC key with the **XPORT** rule-array keyword, provide the variable-length symmetric **IMPORTER** or **EXPORTER** key-token to be used to wrap the generated ECC key. Key bit lengths of 128, 192, and 256 are supported. If this parameter points to a key label, specify rule-array keyword **OKEK-AES** to indicate that the AES key-storage dataset contains the key token.
- When generating an RSA key with the **XPORT** rule-array keyword, provide the fixed-length DES **IMPORTER** or **EXPORTER** key-token to be used to wrap the generated RSA key. If this parameter points to a key label, specify rule-array keyword **OKEK-DES** to indicate that the DES key-storage dataset contains the key token.
- If the **XPORT** rule-array keyword is not specified, specify a null key-token. If this parameter points to a key label, specify keyword **OKEK-AES** for an ECC key or keyword **OKEK-DES** for an RSA key.

generated_key_token_length

Direction: Input/Output
Type: Integer

The length of the generated key token. The field is checked to ensure that it is at least equal to the size of the token being returned. The maximum size is 3500 bytes. On output, this field is updated with the actual token length.

generated_key_token

Direction: Input/Output
Type: String

The internal token or label of the generated RSA or ECC key. When generating

an RSA retained key, on output the verb returns the public token in this variable.

If the key label identifies a key record in PKA key-storage:

- A record must already exist in the PKA key storage file with this same label or the verb will fail
- The generated key token replaces any key token associated with the label.
- the *generated_key_token_length* returned to the application will be the same as the input length.

If the first byte of the identified string does not indicate a key label (that is, not in the range X'20' - X'FE'), and the variable is of sufficient length to receive the result, then the generated key token is returned in the identified variable.

Restrictions

The restrictions for CSNDPKG.

- The maximum public exponent is 17 bits for any key that has a modulus greater than 2048 bits.
- Not all IBM implementations of CCA support a CRT form of the RSA private key; check the product-specific literature. The IBM implementations support an optimized RSA private key (a key in Chinese Remainder Theorem format). The formats vary between versions.
- See “PKA key tokens” on page 66 for the formats used when generating the various forms of key tokens.
- When generating a key for use with ANSI X9.31 digital signatures, the modulus length must be: 1024, 1280, 1536, 1792, 2048, or 4096 bits.
- The key label used for a retained key must not exist in the external PKA key-storage held on the hard disk drive.
- Due to potential loss of a retained private key within the cryptographic engine, retained keys should be avoided for key management purposes.
- 2048-bit RSA keys may have a public exponent in the range of 1 - 256 bytes.
- 4096-bit RSA key public exponents are restricted to the values 3 and 65537.

Required commands

The required commands for CSNDPKG.

- This verb requires the **PKA Key Generate** command (offset X'0103') to be enabled in the active role.
- With the **CLONE** rule-array keyword, enable the **PKA Key Generate - Clone** command (offset X'0204').
- With the **CLEAR** rule-array keyword, enable the **PKA Key Generate - Clear RSA Key** command (offset X'0205') in the hardware.
- To generate ECC keys with the **CLEAR** rule-array keyword, this verb requires the **PKA Key Generate - Clear ECC keys** command (offset X'0326') to be enabled in the active role.
- To generate keys based on the value supplied in the *regeneration_data* variable, you must enable one of these commands:
  - When not using the RETAIN keyword, enable the **PKA Key Generate - Permit Regeneration Data** command (offset X'027D').
  - When using the RETAIN keyword, enable the **PKA Key Generate - Permit Regeneration Data Retain** command (offset X'027E').

- To disallow the wrapping of a key with a weaker key-encrypting key, enable the **Prohibit weak wrapping - Transport keys** command (offset X'0328') in the active role. This command affects multiple verbs. See Chapter 22, "Access control points and verbs," on page 953.
- To receive a warning when wrapping a key with a weaker key-encrypting key, enable the **Warn when weak wrap - Transport keys** command (offset X'032C') in the active role. The **Prohibit weak wrapping - Transport keys** command (offset X'0328') overrides this command.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

Usage notes for CSNDPKG.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDPKGJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNDPKGJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger regeneration_data_length,
           byte[]            regeneration_data,
           hikmNativeInteger skeleton_key_token_length,
           byte[]            skeleton_key_token,
           byte[]            transport_key_identifier,
           hikmNativeInteger generated_key_identifier_length,
           byte[]            generated_key_identifier);
```

PKA Key Import (CSNDPKI)

This verb imports an external PKA or ECC private key token. (This consists of a PKA or ECC private key and public key.)

The secret values of the key can be:

- Clear
- Encrypted under a limited-authority DES importer key if the *source_key_identifier* is an RSA token
- Encrypted under an AES Key Encryption Key if the *source_key_identifier* is an ECC token

This verb can also import a clear PKA key. The PKA Key Token Build verb creates a clear PKA key token.

This verb can also import an external trusted block token for use with the Remote Key Export verb.

Output of this verb is a CCA internal token of the RSA or ECC private key or trusted block.

Format

The format of CSNDPKI.

```
CSNDPKI(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              rule_array_count,
              rule_array,
              source_key_identifier_length,
              source_key_identifier,
              importer_key_identifier,
              target_key_identifier_length,
              target_key_identifier)
```

Parameters

The parameter definitions for CSNDPKI.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0 or 1.

rule_array

Direction: Input
Type: String array

The **rule_array** parameter is a pointer to a string variable containing a keyword. The keyword is 8 bytes in length and must be left-aligned and padded on the right with space characters. The **rule_array** keywords are described in Table 169.

Table 169. Keywords for PKA Key Import control information

Keyword	Description
Token type (One, optional)	
ECC	Specifies that the key being imported is an ECC key.
RSA	Specifies that the key being imported is an RSA key or a trusted block. This is the default.
Transport key type (optional)	
IKEK-AES	The **importer_key_identifier** is an AES key.
IKEK-DES	The **importer_key_identifier** is a DES key. This is the default.

source_key_identifier_length

Direction: Input
Type: Integer

The length of the **source_key_identifier** parameter. The maximum size is 3500 bytes.

source_key_identifier

Direction: Input
Type: String

Contains an external token or label of a PKA private key, without section identifier X'14' (Trusted Block Information), or the trusted block in external form as produced by the Trusted Block Create verb with the **ACTIVATE** keyword.

If a PKA private key without the section identifier X'14' is passed in:

- There are no qualifiers. A retained key can not be used.
- The key token must contain both public-key and private-key information. The private key can be in clear text or it can be enciphered. ECC tokens must contain a private key in cleartext.
- This is the output of the PKA Key Generate (CSNDPKG) verb or the PKA Key Token Build (CSNDPKB) verb.
- If encrypted, the key was created on another platform.

If a PKA private key with the section identifier X'14' is passed in:

- This verb is used to encipher the **MAC** key within the trusted block under the PKA master key instead of the **IMP-PKA** key-encrypting key.
- The **importer_key_identifier** must contain an **IMP-PKA** KEK.

importer_key_identifier

Direction: Input/Output
Type: String

A variable-length field containing an AES or DES key identifier used to wrap the imported key. For RSA keys and trusted blocks, this must be a DES limited authority transport key (**IMP-PKA**). For ECC keys, this must be an AES transport key.

This parameter contains one of the following:

- 64-byte label of a key storage record that contains the transport key.
- 64-byte DES internal key token containing the transport key.
- A variable-length AES internal key token containing the transport key.

This parameter is ignored for clear tokens.

target_key_identifier_length

Direction: Input/Output
Type: Integer

The length of the **target_key_identifier** parameter. The maximum size is 3500 bytes. On output, and if the size is of sufficient length, the variable is updated with the actual length of the **target_key_identifier** field.

target_key_identifier

Direction: Input/Output
Type: String

This field contains the internal token or label of the imported PKA private key or a trusted block. If a label is specified on input, a PKA key storage record with this label must exist. The PKA key storage record with this label will be

overwritten with the imported key unless the existing record is a retained key. If the record is a retained key, the import will fail. A retained key record cannot be overwritten. If no label is specified on input, this field is ignored and should be set to binary zeros on input.

Restrictions

The restrictions for CSNDPKI.

This verb imports RSA keys of up to 4096 bits. However, the hardware configuration sets the limits on the modulus size of keys for digital signatures and key management; thus, the key can be successfully imported but fail when used if the limits are exceeded.

The *importer_key_identifier* parameter is a limited-authority key-encrypting key.

CRT form tokens with a private section ID of X'05' cannot be imported.

Required commands

The required commands for CSNDPKI.

This verb requires the **PKA Key Import** command (offset X'0104') to be enabled in the active role. If the *source_key_token* parameter points to a trusted block, also enable the **PKA Key Import - Import an external trusted block** command (offset X'0311').

To disable the wrapping of a key with a weaker master key, the **Prohibit weak wrapping - Master keys** command (offset X'0333') must be enabled in the active role.

To receive a warning when wrapping a key with a weaker master key, enable the **Warn when weak wrap - Master keys** command (offset X'0332') in the active role. The **Prohibit weak wrapping - Master keys** command overrides this command.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNDPKI.

This verb imports keys of any modulus size up to 2048 bits. However, the hardware configuration sets the limits on the modulus size of keys for digital signatures and key management; thus, the key can be successfully imported but fail when used if the limits are exceeded.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDPKIJ.

This verb has a Java Native Interface (JNI) version, which is named CSNDPKIJ. See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNDPKIJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
```

```
byte[]            exit_data,
hikmNativeInteger rule_array_count,
byte[]            rule_array,
hikmNativeInteger source_key_token_length,
byte[]            source_key_token,
byte[]            transport_key_identifier,
hikmNativeInteger target_key_identifier_length,
byte[]            target_key_identifier);
```

PKA Key Token Build (CSNDPKB)

Use this verb to build external PKA key tokens containing unenciphered private RSA or ECC keys.

You can use this token as input to the PKA Key Import verb to obtain an operational internal token containing an enciphered private key. This verb builds a skeleton token that you can use as input to the PKA Key Generate verb (see Table 168 on page 636). You can also input to this verb a clear unenciphered public RSA or ECC key and return the public key in a token format that other PKA verbs can use directly.

This verb is used to create the following:

- A *skeleton_key_token* for use with the PKA Key Generate verb.
- A key token with a public key that has been obtained from another source.
- A key token with a clear private-key and the associated public key.
- A key token for an RSA private key in optimized Chinese Remainder Theorem (CRT) format.
- An RSA token with X'09' section identifier using the **RSAMEVAR** keyword to obtain a token for a key in Modulus-Exponent format that is variable length.

ECC key generation requires this information in the skeleton token:

- The key type: **ECC**
- The type of curve: Prime or Brainpool
- The size of *p* in bits: 192, 224, 256, 384 or 521 for Prime curves and 160, 192, 224, 256, 320, 384, or 521 for Brainpool curves
- Key usage information
- Optionally, application associated data

Format

The format of CSNDPKB.

```
CSNDPKB(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_value_structure_length,
             key_value_structure,
             private_key_name_length,
             private_key_name,
             user_definable_associated_data_length,
             user_definable_associated_data,
             reserved_2_length,
             reserved_2,
             reserved_3_length,
             reserved_3,
             reserved_4_length,
             reserved_4,
             reserved_5_length,
             reserved_5,
             key_token_length,
             key_token)
```

Parameters

The parameters for CSNDPKB.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1, 2, or 3.

rule_array

Direction: Input
Type: String array

One or two keywords that provide control information to the verb. The keywords must be in contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. The *rule_array* keywords are described in Table 170.

Table 170. Keywords for PKA Key Token Build control information

Keyword	Description
Key type (One, required)	
ECC-PAIR	This keyword indicates building a token containing both public and private ECC key information. The parameter *key_value_structure* identifies the input key values, if supplied.
ECC-PUBL	This keyword indicates building a token containing public ECC key information. The parameter *key_value_structure* identifies the input values, if supplied.

Table 170. Keywords for PKA Key Token Build control information (continued)

Keyword	Description
RSA-AESC (Release 4.3 or later)	Create a key token for an RSA public key and an RSA private key in Chinese-Remainder Theorem (CRT) format with an AES-encrypted OPK.
RSA-AESM (Release 4.3 or later)	Create a key token for an RSA public key and an RSA private key in Modulus-Exponent format with an AES-encrypted OPK.
RSA-CRT	This keyword indicates building a token containing an RSA private key in the optimized Chinese Remainder Theorem (CRT) format. The parameter *key_value_structure* identifies the input key values, if supplied.
RSA-PRIV	This keyword indicates building a token containing both public and private RSA key information. The parameter *key_value_structure* identifies the input key values, if supplied.
RSA-PUBL	This keyword indicates building a token containing public RSA key information. The parameter *key_value_structure* identifies the input values, if supplied.
RSAMEVAR	This keyword indicates RSA-Modulus Exponent-Variant (RSAMEVAR), a type X'09' key token for RSA, named VAR_OPK. **Note:** Key tokens created with this key type **cannot** be passed to the PKA Key Generate verb for creating RETAIN (retained) keys.
Key usage control (One, optional)	
KEY-MGMT	Indicates that an RSA or ECC private key can be used in both the Symmetric Key Import and the Digital Signature Generate verbs. **Note:** Key tokens created with this key usage **cannot** be passed to the PKA Key Generate verb for creating RETAIN (retained) keys.
KM-ONLY	Indicates that an RSA or ECC private key can be used only in symmetric key distribution. **Note:** Key tokens created with this key usage **cannot** be passed to the PKA Key Generate verb for creating RETAIN (retained) keys.
SIG-ONLY	Indicates that an RSA or ECC private key cannot be used in symmetric key distribution. This is the default. **Note:** Only a skeleton token created from PKA Key Token Build with this key usage type can be passed to PKA Key Generate to create a RETAIN (retained) key.
Translate control (One, optional)	
NO-XLATE	The RSA or ECC key cannot be used as a key-encrypting-key for "PKA Key Translate (CSNDPKT)" on page 655. **Note:** Use of this keyword does not matter when creating a skeleton key token for a later retained key generation operation. It is redundant to the necessary **SIG-ONLY** keyword.
XLATE-OK	The RSA or ECC key can be used as a key-encrypting-key for "PKA Key Translate (CSNDPKT)" on page 655. **Note:** Key tokens created with this keyword **cannot** be passed to the PKA Key Generate verb for creating RETAIN (retained) keys.

key_value_structure_length

Direction: Input
Type: Integer

This is a segment of contiguous storage containing a variable number of input clear key values. The length depends on the key type parameter in the *rule_array* and on the actual values input. The length is in bytes. For maximum values, see Table 171 on page 647.

Table 171. PKA Key Token Build - Key value structure length maximum values

Key type	Key value structure maximum value
ECC-PAIR	207
ECC-PUBL	139
RSA-CRT, **RSAMEVAR**	3500
RSA-PRIV	648
RSA-PUBL	520

key_value_structure

Direction: Input
Type: String

This is a segment of contiguous storage containing a variable number of input clear key values and the lengths of these values in bits or bytes, as specified. The structure elements are ordered, of variable length, and the input key values must be right-aligned within their respective structure elements and padded on the left with binary zeros. If the leading bits of the modulus are zeros, do not count them in the length. Table 172 defines the structure and contents as a function of key type.

Table 172. PKA Key Token Build - Key value structure elements

Offset	Length (bytes)	Description
Key value structure (**ECC-PAIR**)		
000	001	Curve type: **X'00'** Prime curve **X'01'** Brainpool curve
001	001	Reserved X'00'
002	002	Length of *p* in bits **X'00A0'** Brainpool P-160 **X'00C0'** Prime P-192, Brainpool P-192 **X'00E0'** Prime P-224, Brainpool P-224 **X'0100'** Prime P-256, Brainpool P-256 **X'0140'** Brainpool P-320 **X'0180'** Prime P-384, Brainpool P-384 **X'0200'** Brainpool P-512 **X'0209'** Prime P-521
004	002	*ddd* - this field is the length of the private key *d* in bytes. This value can be zero if the key token is used as a skeleton key token in the PKA Key Generate verb. The maximum value is 66 bytes.
006	002	*xxx* - this field is the length of the public key *Q* in bytes. This value can be zero if the key token is used as a skeleton key token in the PKA Key Generate verb. The maximum value is 133 bytes, which includes one byte to indicate if the value is compressed.
008	*ddd*	Private key, *d*
008 + *ddd*	*xxx*	Public key, *Q*
Key value structure (**ECC-PUBL**)		
000	001	Curve type: **X'00'** Prime curve **X'01'** Brainpool curve
001	001	Reserved X'00'

Table 172. PKA Key Token Build - Key value structure elements (continued)

Offset	Length (bytes)	Description
002	002	Length of *p* in bits **X'00A0'** Brainpool p-160 **X'00C0'** Prime P-192, Brainpool P-192 **X'00E0'** Prime P-224, Brainpool P-224 **X'0100'** Prime P-256, Brainpool P-256 **X'0140'** Brainpool P-320 **X'0180'** Prime P-384, Brainpool P-384 **X'0200'** Brainpool P-512 **X'0209'** Prime P-521
004	002	*xxx* - this field is the length of the public key *Q* in bytes. This value can be zero if the key token is used as a skeleton key token in the PKA Key Generate verb. The maximum value is 133 bytes, which includes one byte to indicate if the value is compressed.
006	*xxx*	Public key, *Q*
Key value structure (Optimized RSA, Chinese Remainder Theorem format, RSA-CRT)		
000	002	Modulus length in bits (512 - 2048). This is required.
002	002	Modulus field length in bytes, *nnn*. This value can be zero if the key token is used as a *skeleton_key_token* in the PKA Key Generate verb. This value must not exceed 256.
004	002	Public exponent field length in bytes, *eee*. This value can be zero if the key token is used as a *skeleton_key_token* in the PKA Key Generate verb.
006	002	Reserved, binary zero.
008	002	Length of the prime number, *p*, in bytes, *ppp*. This value can be zero if the key token is used as a *skeleton_key_token* in the PKA Key Generate verb. Maximum size of $p + q$ is 256 bytes.
010	002	Length of the prime number, *q*, in bytes, *qqq*. This value can be zero if the key token is used as a *skeleton_key_token* in the PKA Key Generate verb. Maximum size of $p + q$ is 256 bytes.
012	002	Length of d_p, in bytes, *rrr*. This value can be zero if the key token is used as a *skeleton_key_token* in the PKA Key Generate verb. Maximum size of $d_p + d_q$ is 256 bytes.
014	002	Length of d_q, in bytes, *sss*. This value can be zero if the key token is used as a *skeleton_key_token* in the PKA Key Generate verb. Maximum size of $d_p + d_q$ is 256 bytes.
016	002	Length of *U*, in bytes, *uuu*. This value can be zero if the key token is used as a *skeleton_key_token* in the PKA Key Generate verb. Maximum size of U is 256 bytes.
018	*nnn*	Modulus, *n*.
018 + *nnn*	*eee*	Public exponent, *e*. This is an integer such that $1 < e < n$. *e* must be odd. When you are building a *skeleton_key_token* to control the generation of an RSA key pair, the public key exponent can be one of the following values: 3, 65537 ($2^{16} + 1$), or 0 to indicate that a full random exponent should be generated. The exponent field can be a null-length field if the exponent value is 0.
018 + *nnn* + *eee*	*ppp*	Prime number, *p*.
018 + *nnn* + *eee* + *ppp*	*qqq*	Prime number, *q*.
018 + *nnn* + *eee* + *ppp* + *qqq*	*rrr*	$d_p = d \bmod(p\text{-}1)$.
018 + *nnn* + *eee* + *ppp* + *qqq* + *rrr*	*sss*	$d_q = d \bmod(q\text{-}1)$.

Table 172. PKA Key Token Build - Key value structure elements (continued)

Offset	Length (bytes)	Description
018 + *nnn* + *eee* + *ppp* + *qqq* + *rrr* + *sss*	*uuu*	$U = q^{-1} \bmod(p)$.
Key value structure (RSA private, RSA private variable, or RSA public)		
000	002	Modulus length in bits. This is required. When building a skeleton token, the modulus length in bits must be greater than or equal to 512 bits.
002	002	Modulus field length in bytes, *XXX*. This value can be zero if you are using the key token as a skeleton in the PKA Key Generate verb. This value must not exceed 256 when the **RSA-PUBL** keyword is used and must not exceed 128 when the **RSA-PRIV** keyword is used. This verb can build a key token for a public RSA key with a 2048-bit modulus length or it can build a key token for a 1024-bit modulus length private key.
004	002	Public exponent field length in bytes, *YYY*. This value must not exceed 256 when the **RSA-PUBL** keyword is used and must not exceed 128 when the **RSA-PRIV** keyword is used. This value can be zero if you are using the key token as a skeleton token in the PKA Key Generate verb. In this case, a random exponent is generated. To obtain a fixed, predetermined public key exponent, you can supply this field and the public exponent as input to the PKA Key Generate verb.
006	002	Private exponent field length in bytes, *ZZZ*. This field can be zero, indicating that private key information is not provided. This value must not exceed 128 bytes. This value can be zero if you are using the key token as a skeleton token in the PKA Key Generate verb.
008	*XXX*	Modulus, n. This is an integer such that $1 < n < 2^{2048}$. The n is the product of p and q for primes p and q.
008 + *XXX*	*YYY*	RSA public exponent, e. This is an integer such that $1 < e < n$. e must be odd. When you are building a *skeleton_key_token* to control the generation of an RSA key pair, the public key exponent can be one of the following values: 3, 65537 ($2^{16} + 1$), or 0 to indicate that a full random exponent should be generated. The exponent field can be a null-length field if the exponent value is 0.
008 + *XXX* + *YYY*	*ZZZ*	RSA secret exponent d. This is an integer such that $1 < d < n$. The value of d is $e^{-1} \bmod(p-1)(q-1)$. You need not specify this value if you specify **RSA-PUBL** in the *rule_array* parameter.

Note:

1. All length fields are in binary.
2. All binary fields (exponent, lengths, modulus, and so on) are stored with the high-order byte field first. This integer number is right-aligned within the key structure element field.
3. You must supply all values in the structure to create a token containing an RSA or ECC private key for input to the PKA Key Import verb.

private_key_name_length

Direction: Input
Type: Integer

The length can be 0 or 64.

private_key_name

Direction: Input

Type: String

This field contains the name of a private key. The name must conform to CCA key label syntax rules. That is, allowed characters are alphanumeric, national (@, #, $) or period (.). The first character must be alphabetic or national. The name is folded to upper case and converted to ASCII characters. ASCII is the permanent form of the name because the name should be independent of the platform. The name is then cryptographically coupled with clear private key data before encryption of the private key. Because of this coupling, the name can never change after the key token is imported. The parameter is valid only with key type **RSA-CRT**.

user_definable_associated_data_length

Direction: Input
Type: Integer

Length in bytes of the *user_definable_associated_data* parameter. This parameter is valid only for a key type of **ECC-PAIR**, and must be set to 0 for all other key types. The maximum value is 100.

user_definable_associated_data

Direction: Input
Type: String

The **user_definable_associated_data** parameter identifies a string variable containing the associated data that will be placed following the IBM associated data in the token. The associated data is data whose integrity, but not whose confidentiality, is protected by a key wrap mechanism. The **user_definable_associated_data** can be used to bind usage control information.

This parameter is valid only for a key type of **ECC-PAIR**.

reserved_2_length

Direction: Input
Type: Integer

Length in bytes of a reserved parameter. You must set this variable to 0.

reserved_2

Direction: Input
Type: String

The **reserved_2** parameter identifies a string that is reserved. The verb ignores it.

reserved_3_length

Direction: Input
Type: Integer

Length in bytes of a reserved parameter. You must set this variable to 0.

reserved_3

Direction: Input
Type: String

The **reserved_3** parameter identifies a string that is reserved. The verb ignores it.

reserved_4_length

Direction: Input
Type: Integer

Length in bytes of a reserved parameter. You must set this variable to 0.

reserved_4

Direction: Input
Type: String

The **reserved_4** parameter identifies a string that is reserved. The verb ignores it.

reserved_5_length

Direction: Input
Type: Integer

Length in bytes of a reserved parameter. You must set this variable to 0.

reserved_5

Direction: Input
Type: String

The **reserved_5** parameter identifies a string that is reserved. The verb ignores it.

key_token_length

Direction: Input/Output
Type: Integer

Length of the returned key token. The verb checks the field to ensure that it is at least equal to the size of the token to return. On return from this verb, this field is updated with the exact length of the **key_token** created. On input, a size of 3500 bytes is sufficient to contain the largest **key_token** created.

key_token

Direction: Output
Type: String

The returned key token containing an unenciphered private or public key. The private key is in an external form that can be exchanged with different CCA PKA systems. You can use the public key token directly in appropriate CCA signature verification or key management services.

Restrictions

The restrictions for CSNDPKB.

None.

Required commands

The required commands for CSNDPKB.

None.

Usage notes

The usage notes for CSNDPKB.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDPKBJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNDPKBJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger key_values_structure_length,
           byte[]            key_values_structure,
           hikmNativeInteger key_name_length,
           byte[]            key_name,
           hikmNativeInteger user_definable_associated_data_length,
           byte[]            user_definable_associated_data,
           hikmNativeInteger reserved_2_length,
           byte[]            reserved_2,
           hikmNativeInteger reserved_3_length,
           byte[]            reserved_3,
           hikmNativeInteger reserved_4_length,
           byte[]            reserved_4,
           hikmNativeInteger reserved_5_length,
           byte[]            reserved_5,
           hikmNativeInteger token_length,
           byte[]            token);
```

PKA Key Token Change (CSNDKTC)

The PKA Key Token Change verb changes PKA key tokens (RSA or ECC) or trusted block key tokens, from encipherment under old ASYM-MK or APKA-MK, to encipherment under the current ASYM-MK or APKA-MK master key.

IMPORTANT

Two problems have been discovered with the CCA microcode related to the reenciphering of master keys. Although similar, the two problems are slightly different and exist in different levels of the microcode. These problems could lead to a loss of operational private keys after a master key change. Symmetric keys are not affected. Although it is expected few customers will be impacted this document describes the problems and how to recover.

For details, see http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/FLASH10764.

The PKA Key Token Change (CSNDKTC) verb has been changed to not permit the use of the **RTNMK** keyword for processor firmware levels that have this problem.

- For RSA key tokens - Key tokens must be private internal PKA key tokens in order to be changed by this verb. PKA private keys encrypted under the Key Management Master Key (KMMK) cannot be reenciphered using this services unless the KMMK has the same value as the Signature Master Key (SMK).
- For trusted block key tokens - Trusted block key tokens must be internal.

- For ECC key tokens - Key tokens must be private internal ECC key tokens encrypted under the APKA-MK.

Format

The format of CSNDKTC.

```
CSNDKTC(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              rule_array_count,
              rule_array,
              key_identifier_length,
              key_identifier)
```

Parameters

The parameters for CSNDKTC.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1 or 2.

rule_array

Direction: Input
Type: String array

The process rule for the verb. The keyword must be in eight bytes of contiguous storage, left-aligned, and padded on the right with blanks. The *rule_array* keywords are described in Table 173.

Table 173. Keywords for PKA Key Token Change control information

Keyword	Description
Token type (One, optional)	
ECC	Specifies that the key being changed is an ECC key. This keyword was introduced with CCA 4.1.0.
RSA	Specifies that the key being changed is an RSA key or a trusted block. This is the default. This keyword was introduced with CCA 4.1.0.
Reencipherment method (One, required)	
RTCMK	If the *key_identifier* is an RSA key token, the verb will change an RSA private key from encipherment with the old ASYM-MK to encipherment with the current ASYM-MK. If the *key_identifier* is a trusted block token, the verb will change the trusted block's embedded MAC key from encipherment with the old ASYM-MK to encipherment with the current ASYM-MK. If the *key_identifier* is an ECC key token, the verb will change an ECC private key from encipherment with the old APKA-MK to encipherment with the current APKA-MK.

Table 173. Keywords for PKA Key Token Change control information (continued)

Keyword	Description
RTNMK	Re-enciphers a private (internal) RSA or ECC key to the new master key. A key enciphered under the new master key is not usable. It is expected that the user will use this keyword (RTNMK) to take a preparatory step in re-enciphering an external key store that they manage themselves to a new master-key, before the set operation has occurred. Note also that the new master-key register must be full; it must have had the last key part loaded and therefore not be empty or partially full (partially full means that one or more key parts have been loaded but not the last key part). The 'SET' operation makes the new master-key operational, moving it to the current master-key register, and the current master-key is displaced into the old master-key register. When this happens, all the keys that were re-enciphered to the new master-key are now usable, because the new master-key is not 'new' any more, it is 'current'. Because the RTNMK keyword is added primarily for support of externally managed key storage (see "Key Storage on z/OS (RTNMK-focused)" on page 405, it is not valid to pass a *key_identifer* when the RTNMK keyword is used. Only a full internal key token (encrypted under the current master-key) can be passed for re-encipherment with the RTNMK keyword. When a key LABEL is passed along with the RTNMK keyword, the error return code 8 with reason code 63 will be returned. For more information, see "Key storage with Linux on z Systems, in contrast to z/OS on z Systems" on page 403.
VALIDATE	Validate an internal PKA key token which is under the current master key (same processing as RTNMK without checking the new master key or actually re-enciphering the token).

key_identifier_length

Direction: Input
Type: Integer

The length of the **key_identifier** parameter. The maximum size is 3500 bytes.

key_identifier

Direction: Input/Output
Type: String

Contains an internal key token of an internal RSA or ECC key, or trusted block key. If the key token is an RSA key token, the private key within the token is securely re-enciphered under the current ASYM-MK. If the key token is a trusted block key token, the MAC key within the token is securely re-enciphered under the current ASYM-MK. If the key token is an ECC key token, the private key within the token is securely re-enciphered under the current APKA-MK.

Restrictions

The restrictions for CSNDKTC.

None.

Required commands

The required commands for CSNDKTC.

This verb requires the **PKA Key Token Change RTCMK** command (offset X'0102') to be enabled in the active role.

To disable the wrapping of a key with a weaker master key, the **Prohibit weak wrapping - Master keys** command (offset X'0333') must be enabled in the active role.

To receive a warning when wrapping a key with a weaker master key, enable the **Warn when weak wrap - Master keys** command (offset X'0332') in the active role. The **Prohibit weak wrapping - Master keys** command overrides this command.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNDKTC.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDKTCJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNDKTCJ(
  hikmNativeInteger return_code,
  hikmNativeInteger reason_code,
  hikmNativeInteger exit_data_length,
  byte[]            exit_data,
  hikmNativeInteger rule_array_count,
  byte[]            rule_array,
  hikmNativeInteger key_label_length,
  byte[]            key_label);
```

PKA Key Translate (CSNDPKT)

Use the PKA Key Translate verb to translate an RSA key in a PKA key-token using an output format specified by the input rule array. The RSA key to be translated is provided in a source PKA key-token that contains a private-key section, and the translated key is returned in the buffer identified by the target_key_token parameter. If the source key is in an external key-token, the source transport key must be in an operational fixed-length DES key-token.

This verb changes only Private Internal PKA Key Tokens.

The source CCA RSA key token must be wrapped with a transport key encrypting key (KEK). The XLATE bit must also be turned on in the key usage byte of the source token. The source token is unwrapped using the specified source transport KEK. The target key token will be wrapped with the specified target transport KEK. Existing information in the target token is overwritten.

There are three types of output formatting available described in the subsequent sections. The first type is an external-to-external translation of an RSA key to one of three smart card formats. The second is an external-to-external or internal-to-internal translation of an RSA key to a target PKA key-token that is protected by an AES transport key or an APKA master key. The third is an external-to-external translation of an RSA key to one of three EMV formats.

Target smart card formats

To use this verb to translate an RSA key into a smart-card format, do the following:

- Specify one of the following rule-array keywords for the smart-card format to apply to the target key:

  SCVISA
  Specifies translation of an RSA private-key to the Visa proprietary format. This format is defined in *Visa Smart Debit/Credit Technical Guide to Visa Applets for GlobalPlatform Cards* and *Visa Smart Debit Credit Personalization Guide for GlobalPlatform Cards*.

  SCCOMME
  Specifies translation of an RSA private-key to the common Modulus-Exponent (M-E) format. This format is defined in *Visa Smart Debit/Credit Technical Guide to Visa Applets for GlobalPlatform Cards*.

  SCCOMCRT
  Specifies translation of an RSA private-key to the common Chinese-Remainder Theorem (CRT) format. This format is defined in *Visa Smart Debit/Credit Technical Guide to Visa Applets for GlobalPlatform Cards*.

  Note:

  1. Translation from an M-E format to a CRT format is not supported.
  2. Translation from a CRT format to an M-E format is not supported.

- Specify the source key identifier of an external PKA key-token that has been protected by a fixed-length DES transport key to be translated. The RSA private key for smart card output formats SCVISA, SCCOMME, and SCCOMCRT must have translation control of XLATE-OK (offset 50 in the private-key section).
- Specify the source transport key identifier of an operational fixed-length DES transport key (EXPORTER or IMPORTER) to be used to unwrap the source key.
- Specify the target transport key identifier of an operational fixed-length DES transport key to be used to wrap the unwrapped source key. The control vector of the target transport key must have CV bit 22 = B'1' (XLATE).

  Note: Translation using an EXPORTER source transport key and an IMPORTER target transport key is not allowed.
- Specify the buffer of the target key token, to receive the returned external TDES-wrapped PKA key-token.

The verb builds the target external key-token using the chosen smart card format as follows:

1. The external RSA source key-token is unwrapped using the operational fixed-length DES source transport key.
2. The unwrapped key material is formatted into the specified target smart-card format.
3. The formatted key-material is TDES encrypted in ECB mode using the operational fixed-length DES target transport key.
4. The formatted and encrypted key material is written to the buffer identified by the **target_key_token** parameter, and the target key token length is updated.

Target AES-protected formats

To use this verb to translate an RSA key into an AES-protected format, specify:

- One of the following rule-array keywords for the AES-protected format to apply to the target key:

  EXTDWAKW
  specifies translation of an RSA key in an external TDES-wrapped (DES transport key) RSA key-token into an external AESKW-wrapped RSA key-token

  INTDWAKW
  specifies translation of an RSA key in an internal TDES-wrapped (PKA master key) RSA key-token into an internal AESKW-wrapped (APKA master key) PKA key-token.

  Note: An AESKW-wrapped key is protected at a higher level than a TDES-wrapped key.
- The source key identifier of a PKA key-token to be translated, either in an external key-token that has been protected by a fixed-length DES transport key to be translated, or an internal key-token that has been protected by a PKA master key.
- Source key is in an external PKA key-token:

  The source transport key identifier of an operational fixed-length DES transport key (EXPORTER or IMPORTER) to be used to unwrap the source key. The control vector of the source transport key does not require the XLATE bit on.

  The target transport key identifier of an operational variable-length AES transport key to be used to wrap the OPK data of the target key-token. In addition, the key usage fields must have the algorithm wrap control set so that the key can wrap or unwrap RSA keys (WR-RSA).

  The buffer of the target key-token, to receive the external RSA key-token with AES-wrapped OPK.
- Source key is in an internal PKA key-token:

  Set the source transport key identifier length to 0 or identify a null key-token as the source transport key.

  Set the target transport key identifier length to 0 or identify a null key-token as the target transport key.

  The buffer of the target key-token, to receive the internal PKA key-token with APKA-wrapped OPK.

The verb builds the target external key-token using the chosen AES-protected format as follows:

1. The source key-token is unwrapped using the source transport key or a PKA master-key, as appropriate.
2. The unwrapped key material is wrapped using the AES OPK of the target key-token. The OPK in turn is wrapped by the AES key from the target transport key if the target key is external, or the APKA master key if the target key is internal.
3. The completed external or internal key-token is written to the key token buffer identified by the **target_key_token** parameter, and the target key token length is updated.

Target EMV formats

To use this verb to translate an RSA key into an EMV format, specify one of the following **rule_array** keywords for the EMV format to apply to the target key:

EMVCRT
specifies the translation of an RSA CRT private-key to an EMV CRT format and wrapped using TDES-ECB

EMVDDA
specifies the translation of an RSA CRT private-key to an EMV DDA format and wrapped using TDES-CBC

EMVDDAE
specifies the translation of an RSA CRT private-key to an EMV DDA format and wrapped using TDES-ECB.

Note: The PKA source key-token must have a private key section of X'08', and the bit length of the modulus must be 512 - 2040.

Format

The format of CSNDPKT.

```
CSNDPKT(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             source_key_identifier_length,
             source_key_identifier,
             source_transport_key_identifier_length,
             source_transport_key_identifier,
             target_transport_key_identifier_length,
             target_transport_key_identifier,
             target_key_token_length,
             target_key_token)
```

Parameters

The parameters for CSNDPKT.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. This value must be 1.

rule_array

Direction: Input
Type: String array

The process rule for the verb. The keyword must be in eight bytes of contiguous storage, left-aligned, and padded on the right with blanks. The **rule_array** keywords are described in Table 174 on page 659.

Table 174. Keywords for PKA Key Translate control information

Keyword	Description
Output format (One, required)	
EXTDWAKW	Specifies that the source key is an external DES wrapped token to be converted to an AESKW wrapped token.
INTDWAKW	Specifies that the source key is an internal DES wrapped token to be converted to an AESKW wrapped token.
EMVDDA	This keyword indicates translating an external RSA CRT key into EMV DDA format and wrapping with TDES-CBC. The XLATE bit (bit 22) must be set in the **target_transport_key** control vector.
EMVDDAE	This keyword indicates translating an external RSA CRT key into EMV DDAE format and wrapping with TDES-ECB. The XLATE bit (bit 22) must be set in the **target_transport_key** control vector.
EMVCRT	This keyword indicates translating an external RSA CRT key into EMV CRT format and wrapping with TDES-ECB. The XLATE bit (bit 22) must be set in the **target_transport_key** control vector.
SCCOMCRT	This keyword indicates translating the key into the smart card Chinese Remainder Theorem format.
SCCOMME	This keyword indicates translating the key into the smart card Modulus-Exponent format.
SCVISA	This keyword indicates translating the key into the smart card Visa proprietary format.

source_key_length

Direction: Input
Type: Integer

The length of the **source_key** parameter. The maximum size is 3500 bytes.

source_key

Direction: Input
Type: String

This field contains either a key label identifying an RSA private key, or an external public-private key token. The private key must be wrapped with a key encrypting key.

source_transport_key_length

Direction: Input
Type: Integer

Length in bytes of the **source_transport_key** parameter. This value must be 64.

source_transport_key

Direction: Input/Output
Type: String

This field contains an internal token or label of a DES key-encrypting key. This key is used to unwrap the input RSA key token specified with parameter **source_key**. See "Usage notes" on page 661 for details on the type of transport key that can be used.

target_transport_key_length

Direction: Input
Type: Integer

Length in bytes of the **target_transport_key** parameter. This value must be 64.

target_transport_key

Direction: Input/Output
Type: String

This field contains an internal token or label of a DES key-encrypting key. This key is used to wrap the output RSA key returned with the **target_key_token** parameter. See "Usage notes" on page 661 for details on the type of transport key that can be used.

target_key_token_length

Direction: Input
Type: Integer

Length in bytes of the **target_key_token** parameter. On output, the value in this variable is updated to contain the actual length of the **target_key_token** produced by the verb. The maximum length is 3500 bytes.

target_key_token

Direction: Output
Type: String

This field contains the RSA key in the smartcard format specified in the **rule_array** parameter, and is protected by the key-encrypting key specified in the **target_transport_key** parameter. This is not a CCA token, and cannot be stored in the key storage.

Restrictions

The restrictions for CSNDPKT.

CCA RSA Modulus-Exponent tokens will not be translated to the **SCCOMCRT** format. CCA RSA Chinese Remainder Theorem tokens will not be translated to the **SCCOMME** format. SCVISA supports only Modulus-Exponent (ME) keys.

Required commands

The required commands for CSNDPKT.

This verb requires the following commands to be enabled in the active role based on the keyword:

Rule-array keyword	Offset	Command
EMVCRT	X'033A'	PKA Key Translate - from CCA RSA CRT to EMV CRT format
EMVDDA	X'0338'	PKA Key Translate - from CCA RSA CRT to EMV DDA format
EMVDDAE	X'0339'	PKA Key Translate - from CCA RSA CRT to EMV DDAE format
EXTDWAKW	X'00FF'	PKA Key Translate - Translate external key token
INTDWAKW	X'00FE'	PKA Key Translate - Translate internal key token
SCVISA	X'0318'	PKA Key Translate - from CCA RSA to SC Visa Format
SCCOMME	X'0319'	PKA Key Translate - from CCA RSA to SC ME Format

Rule-array keyword	Offset	Command
SCCOMCRT	X'031A'	**PKA Key Translate - from CCA RSA to SC CRT Format**

These commands must also be enabled to allow the key type combinations shown in this table:

Source transport key type	Target transport key type	Offset	Command
EXPORTER	EXPORTER	X'031B'	**PKA Key Translate - from source EXP KEK to target EXP KEK**
IMPORTER	EXPORTER	X'031C'	**PKA Key Translate - from source IMP KEK to target EXP KEK**
IMPORTER	IMPORTER	X'031D'	**PKA Key Translate - from source IMP KEK to target IMP KEK**
EXPORTER	IMPORTER	N/A	This key type combination is not allowed.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

The following access control points control the use of weak transport keys:

- To disable the wrapping of a key with a weaker transport key, the **Prohibit weak wrapping - Transport keys** command (offset X'0328') must be enabled in the active role.
- To receive an informational message when wrapping a key with a weaker key-encrypting key, enable the **Warn when weak wrap - Transport keys** command (offset X'032C') in the active role. The **Prohibit weak wrapping - Transport keys** command overrides this command.

The following access control points control the use of weak master keys:

- To disable the wrapping of a key with a weaker master key, the **Prohibit weak wrapping - Master keys** command (offset X'0333') must be enabled in the active role.
- To receive a warning when wrapping a key with a weaker master key, enable the **Warn when weak wrap - Master keys** command (offset X'0332') in the active role. The **Prohibit weak wrapping - Master keys** command overrides this command.

Usage notes

The usage notes for CSNDPKT.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDPKTJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNDPKTJ (
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger source_key_identifier_length,
           byte[]            source_key_identifier,
           hikmNativeInteger source_transport_key_identifier_length,
           byte[]            source_transport_key_identifier,
           hikmNativeInteger target_transport_key_identifier_length,
           byte[]            target_transport_key_identifier,
           hikmNativeInteger target_key_token_length,
           byte[]            target_key_token);
```

PKA Public Key Extract (CSNDPKX)

Use the PKA Public Key Extract verb to extract a PKA public key token from a supplied PKA internal or external private key token.

This verb performs no cryptographic verification of the PKA private token. You can verify the private token by using it in a verb such as Digital Signature Generate.

Format

The format of CSNDPKX.

```
CSNDPKX(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             source_key_indentifier_length,
             source_key_identifier,
             target_public_key_token_length,
             target_public_key_token)
```

Parameters

The parameters for CSNDPKX.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 0.

rule_array

Direction: Input
Type: String array

This parameter is ignored.

source_key_identifier_length

Direction: Input
Type: Integer

The length of the *source_key_identifier* parameter. The maximum size is 3500 bytes. When the *source_key_identifier* parameter is a key label, this field specifies the length of the label.

source_key_identifier

Direction: Input/Output
Type: String

The internal or external token of a PKA private key or the label of a PKA private key. This can be the input or output from the PKA Key Import or PKA Key Generate verbs. This verb supports:

- RSA private key token formats supported on the CEX*C. If the *source_key_identifier* specifies a label for a private key that has been retained within a CEX*C, this verb extracts only the public key section of the token.
- ECC private key token formats supported starting with CEX3C.

target_public_key_token_length

Direction: Input/Output
Type: Integer

The length of the *target_public_key_token* parameter. The maximum size is 2500 bytes. On output, this field will be updated with the actual byte length of the *target_public_key_token*.

target_public_key_token

Direction: Output
Type: String

This field contains the token of the extracted PKA public key.

Restrictions

The restrictions for CSNDPKX.

None.

Required commands

The required commands for CSNDPKX.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNDPKX.

This verb extracts the public key from the internal or external form of a private key. However, it does not check the cryptographic validity of the private token.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDPKXJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNDPKXJ(
          hikmNativeInteger return_code,
          hikmNativeInteger reason_code,
          hikmNativeInteger exit_data_length,
          byte[]            exit_data,
          hikmNativeInteger rule_array_count,
          byte[]            rule_array,
          hikmNativeInteger source_key_identifier_length,
          byte[]            source_key_identifier,
          hikmNativeInteger target_key_token_length,
          byte[]            target_key_token);
```

Remote Key Export (CSNDRKX)

This verb is used as a method of secured transport of DES keys using asymmetric techniques from a security module (for example, the CEX3C) to a remote device such as an Automated Teller Machine (ATM).

The DES keys to be transported are either key encrypting keys that are generated within the coprocessor or, alternately, operational keys or replacement KEKs enciphered under a KEK currently installed in a remote device.

Generating and exporting DES keys

This verb uses a trusted block to generate or export DES keys.

To create a trusted block, see "Trusted Block Create (CSNDTBC)" on page 676. Remote Key Export accepts as input parameters a trusted block, a public-key certificate and certificate parameters, a transport key, a rule ID to identify the appropriate rule section to be used within a trusted block, an importer key, a source key, optional extra data that can be used as part of the OAEP key-wrapping process, and key-check parameters used to calculate an optional key-check value.

This verb validates all input parameters for generate and export operations. After the verb performs the input parameter validation, the remaining steps depend on whether the generate option or the export option is specified in the selected rule of the trusted block.

This is a high-level description of the remaining processing steps for generate and export.

Processing for generate operation

The verb performs these steps for the generate operation:

1. Generates a random value for the generated key, K. The generated key length specified by the selected rule determines the key length.
2. XORs the output key variant with the randomly generated key K from the previous step, if the selected rule contains a common export key parameters subsection and the output key variant length is greater than zero. Adjusts the result to have valid DES key parity.

3. Continues with "Final processing common to generate and export operations."

Processing for export operation

The verb performs these steps for the export operation:

1. If the selected rule contains a transport key rule reference subsection, verifies that the rule ID in the transport key rule reference subsection matches the rule ID in the token identified by the **transport_key_identifier** parameter, provided that the token is an RKX key-token. For more information on RKX key tokens, see "External RKX DES key tokens" on page 773.
2. Verifies that the length of the transport key variant in the transport key variant subsection of the selected rule is greater than or equal to the length of the key identified by the **transport_key_identifier** parameter.
3. Verifies that the key token identified by the **importer_key_identifier** parameter is of key type IMPORTER, if the **source_key_identifier** parameter identifies an external CCA DES key-token.
4. Recovers the clear value of the source key, K, identified by the **source_key_identifier** parameter.
5. Verifies that the length of key K is between the export key minimum length and export key maximum length specified in the common export key parameters subsection of the selected rule.
6. XORs the output key variant with the randomly generated key K from the previous step, if the selected rule contains a common export key parameters subsection and the output key variant length is greater than zero. Adjusts the result to have valid DES key parity.
7. Uses the public key in the trusted block to verify the digital signature embedded in the certificate variable if the **certificate_length** variable is greater than zero. Any necessary certificate objects are located with information from the **certificate_parms** variable. Returns an error if the signature verification fails.
8. XORs the transport key variant with the clear value of the transport key (recovered in the previous step) if the selected rule contains a transport key variant subsection and the output key variant length is greater than zero. Adjusts the result to have valid DES key parity.
9. Continues with "Final processing common to generate and export operations."

Final processing common to generate and export operations

1. Based on the symmetric encrypted output key format flag of the selected rule, returns the encrypted result in the token identified by the *sym_encrypted_key_identifier* parameter:
   - of "Processing for generate operation" on page 664, step 2, or of "Processing for export operation," step 6, into an RKX key-token, if the flag indicates to return an RKX key-token.
   - using the resulting key from "Processing for export operation," step 6 into a CCA DES key-token and returns it in the token identified by the *sym_encrypted_key_identifier* parameter, if the flag indicates to return a CCA DES key-token.
2. Encrypts the key result from "Processing for generate operation" on page 664, step 2 or from "Processing for export operation," step 6, with the format

specified, if the asymmetric encrypted output key format flag of the selected rule indicates to output an asymmetric encrypted key and return it to the **asym_encrypted_key** parameter.

3. Returns the computed key-check value as determined by the key-check algorithm identifier if the key-check algorithm identifier in the specified rule indicates to compute a key-check value. The value is returned in the **key_check_value** variable.

Format

The format of CSNDRKX.

```
CSNDRKX(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             trusted_block_identifier_length,
             trusted_block_identifier,
             certificate_length,
             certificate,
             certificate_parms_length,
             certificate_parms,
             transport_key_identifier_length,
             transport_key_identifier,
             rule_id_length,
             rule_id,
             importer_key_identifier_length,
             importer_key_identifier,
             source_key_identifier_length,
             source_key_identifier,
             asym_encrypted_key_length,
             asym_encrypted_key,
             sym_encrypted_key_identifier_length,
             sym_encrypted_key_identifier,
             extra_data_length,
             extra_data,
             key_check_parameters_length,
             key_check_parameters,
             key_check_value_length,
             key_check_value)
```

Parameters

The parameters for CSNDRKX.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of keywords in the **rule_array** variable. This number must be 0, 1 or 2.

rule_array

Direction: Output
Type: String array

The **rule_array** is an array of keywords. The keywords must be 8 bytes of contiguous storage with the keyword left-justified in its 8-byte location and padded on the right with blanks. The **rule_array** keywords are described in Table 175.

Table 175. Keywords for Remote Key Export control information

Keyword	Description
Key wrapping method (Optional)	
USECONFG	This is the default. Specifies to wrap the key using the configuration setting for the default wrapping method. The default wrapping method configuration setting may be changed using the TKE. This keyword is ignored for AES keys.
WRAP-ENH	Specifies that the new enhanced wrapping method is to be used to wrap the key.
WRAP-ECB	Specifies that the original wrapping method is to be used.
Translation control (Optional, valid only for enhanced wrapping)	
ENH-ONLY	Specify this keyword to indicate that the key once wrapped with the enhanced method cannot be wrapped with the original method. This restricts translation to the original method.

trusted_block_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *trusted_block_identifier* variable. The maximum length is 3500 bytes.

trusted_block_identifier

Direction: Input
Type: String

A pointer to a string variable containing a trusted block key-token of an internal trusted block, or the key label of a trusted block key-token record of an internal trusted block. It is used to validate the public-key certificate and to define the rules for key generation and key export.

certificate_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *certificate* variable. The maximum length is 5000 bytes.

It is an error if the *certificate_length* variable is 0 and the trusted block's asymmetric encrypted output key format in the rule section selected by the *rule_id* variable indicates PKCS-1.2 output format or RSA-OAEP output format.

If the *certificate_length* variable is 0 or the trusted block's asymmetric encrypted output key format in the rule section selected by the *rule_id* variable indicates no asymmetric key output, the certificate is ignored.

certificate

Direction: Input
Type: String

A pointer to a string variable containing a public-key certificate. The certificate

must contain the public-key modulus and exponent in binary form, as well as a digital certificate. The certificate must verify using the root public key that is in the trusted block pointed to by the *trusted_block_identifier* parameter.

Note: After the hash is computed over the certificate data specified by offsets 28 and 32, the hash is BER encoded by pre-pending these bytes:

X'30213009 06052B0E 03021A05 000 414'

See "PKCS #1 hash formats" on page 950.

certificate_parms_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *certificate_parms* variable. The length must be 36 bytes if the *certificate_length* variable is 0, else the length must be 0.

certificate_parms

Direction: Input
Type: String

A pointer to a string variable containing a structure for identifying the location and length of values within the public-key certificate pointed to by the *certificate* parameter. If the value of the *certificate_length* variable is 0, then the information in this variable is ignored but the variable must be declared. The format of the *certificate_parms* variable is defined in Table 176.

Table 176. Keywords for Remote Key Export certificate_parms parameter

Offset (bytes)	Length (bytes)	Description
0	4	Offset of modulus
4	4	Length of modulus
8	4	Offset of public exponent
12	4	Length of public exponent
16	4	Offset of digital signature
20	4	Length of digital signature
24	1	Identifier for hash algorithm. The following values are defined: **Identifier** — **Hash algorithm** **X'01'** SHA-1 **X'02'** MD5 (Currently not supported) **X'03'** RIPEMD-160 (Currently not supported)
25	1	Identifier for digital signature hash formatting method used. The following values are defined: **Identifier** — **Hash formatting method** **X'01'** PKCS-1.0 **X'02'** PKCS-1.1 **X'03'** X9.31 (Currently not supported) **X'04'** ISO-9796 (Currently not supported) **X'05'** ZERO-PAD (Currently not supported)
26	2	Reserved, must be binary zeros

Table 176. Keywords for Remote Key Export certificate_parms parameter (continued)

Offset (bytes)	Length (bytes)	Description
28	4	Offset of first byte of certificate data hashed to compute the digital signature
32	4	Length of certificate data hashed to compute the digital signature

Note: The modulus, exponent, and signature values can have bit lengths that are not multiples of 8; each of these values is right-aligned and padded on the left with binary zeroes to make it an even number of bytes in length.

transport_key_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *transport_key_identifier* variable. The length must be 0 or 64 bytes.

transport_key_identifier

Direction: Input
Type: String

A pointer to a string variable containing a KEK key-token, or a key label of a KEK key-token record. The KEK is either an internal CCA DES key-token (key type IMPORTER or EXPORTER), or an external version X'10' (RKX) DES key-token. It is used to encrypt a key exported by the verb.

When the symmetric encrypted output key format flag of the selected rule indicates return an RKX key-token, this parameter is ignored but must be declared.

- If this parameter points to a CCADES key-token, the token must be of key type IMPORTER or EXPORTER.
- If the *source_key_identifier* parameter identifies an internal CCA DES key-token, the token must be of key type EXPORTER.

For more information on RKX key tokens, see "External RKX DES key tokens" on page 773.

rule_id_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *rule_id* variable. The length must be eight bytes.

rule_id

Direction: Input
Type: String

A pointer to a string variable that identifies the rule in the trusted block to be used to control key generation or export. The trusted block can contain multiple rules, each of which is identified by a unique rule ID value.

importer_key_identifier_length

Direction: Input

Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *importer_key_identifier* variable. The length must be 0 or 64 bytes.

importer_key_identifier

Direction: Input
Type: String

A pointer to a string variable containing an **IMPORTER** KEK key-token or a label of an **IMPORTER** KEK key-token record. This KEK is used to decipher the key pointed to by the *source_key_identifier* parameter.

This variable is ignored if the verb is used to generate a new key, or the *source_key_identifier* variable contains either an RKX key token or an internal CCA DES key-token. For more information on RKX key tokens, see "External RKX DES key tokens" on page 773.

source_key_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *source_key_identifier* variable. The length must be 0 or 64 bytes.

source_key_identifier

Direction: Input
Type: String

A pointer to a string variable containing a DES key-token or a label of a DES key-token record. The key token contains the key to be exported, and must meet one of these criteria:

- It is a single-length or double-length external CCA DES key-token.
- It is a single-length or double-length internal CCA DES key-token.
- It is a single-length, double-length, or triple-length RKX key-token.

Note:

1. If the key token is a CCA DES key-token, its XPORT-OK control vector bit (bit 17) must be B'1', or else the export will not be allowed.
2. If a DES key-token has three 8-byte key parts, the parts are considered unique if any two of the three key parts differ.

asym_encrypted_key_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *asym_encrypted_key* variable. On output, the variable is updated with the actual length of the *asym_encrypted_key* variable. The input length must be at least the length of the modulus in bytes of the public-key in the certificate variable.

asym_encrypted_key

Direction: Output
Type: String

A pointer to a string variable containing a generated or exported clear key

returned by the verb. The clear key is encrypted by the public (asymmetric) key provided by the certificate variable.

sym_encrypted_key_identifier_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *sym_encrypted_key_identifier* variable. On output, the variable is updated with the actual length of the *sym_encrypted_key_identifier* variable. The input length must be a minimum of 64 bytes.

sym_encrypted_key_identifier

Direction: Output
Type: String

A pointer to a string variable. On input, the *sym_encrypted_key_identifier* variable must contain either a key label of a CCA DES key-token record or an RKX key-token record, or be filled with binary zeros.

On output, the verb produces a CCA DES key-token or an RKX key-token, depending on the value of the symmetric encrypted output key format value of the rule section within the *trusted_block_identifier* variable. The key token produced contains either a generated or exported key encrypted using the key-encrypting key provided by the *transport_key_identifier* variable.

- If the output is an external CCA DES key-token:
  1. If a common export key parameters subsection (X'0003') is present in the selected rule, the control vector (CV) is copied from the subsection into the output CCA DES key-token. Otherwise, the CV is copied from source key-token.
  2. If a transport key variant subsection (X'0001') is present in the selected rule, the key is multiply enciphered under the transport key XORed with the transport key variant from the subsection. Otherwise, the key is multiply enciphered under the transport key XORed with binary zero
  3. XORs the CV in the token with the encrypted result from the previous step.
  4. Stores the previous result in the token and updates the TVV.
- If the output is an (external) RKX key-token:
  1. Encrypts the key using a variant of the trusted block MAC key.
  2. Builds the token with the encrypted key and the *rule_id* variable.
  3. Calculates the MAC of the token contents and stores the result in the token.

If the *sym_encrypted_key_identifier* variable is a key label on input, on output the key token produced by the verb is stored in DES key-storage and the variable remains the same. Otherwise, on output the variable is updated with the key token produced by the verb, provided the field is of sufficient length.

extra_data_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *extra_data* variable. The length must be less than or equal to the byte length of the certificate public key modulus minus the generated/exported key length minus 42 (X'2A'), which is the OAEP overhead. For example, if the public key

in the certificate has a modulus length of 1024 bits (128 bytes), and the exported key is single length, then the extra data length must be less than or equal to 128 minus 8 minus 42, which equals 78.

extra_data

Direction: Input
Type: String

A pointer to a string variable containing extra data to be used as part of the OAEP key-wrapping process. The *extra_data* variable is used when the output format for the RSA-encrypted key that is returned in the *asym_encrypted_key* variable is RSA-OAEP; otherwise, it is ignored.

Note: The RSA-OAEP format is specified as part of the rule in the trusted block.

key_check_parameters_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *key_check_parameters* variable. The length must be 0.

key_check_parameters

Direction: Input
Type: String

Reserved for future use.

key_check_value_length

Direction: Input/Output
Type: Integer

A pointer to a string variable containing the number of bytes of data in the *key_check_value* variable. On output, and if the field is of sufficient length, the variable is updated with the actual length of the *key_check_value* variable.

key_check_value

Direction: Output
Type: String

A pointer to a string variable containing the result of the key-check algorithm chosen in the rule section of the selected trusted block. See "Encrypt zeros DES-key verification algorithm" on page 930 and "Modification Detection Code calculation" on page 930. When the selected key-check algorithm is to encrypt an 8-byte block of binary zeros with the key, and the generated or exported key is:

- Single length
  1. A value of 0, 1, or 2 is considered insufficient space to hold the output encrypted result, and the verb returns an error.
  2. A value of 3 returns the leftmost three bytes of the encrypted result if the *key_check_value_length* variable is 3 or greater. Otherwise, an error is returned.
  3. A value of 4 - 8 returns the leftmost four bytes of the encrypted result if the *key_check_value_length* variable is 4 or greater. Otherwise, an error is returned.

- Double length or triple length

  The verb returns the entire 8-byte result of the encryption in the *key_check_value* variable if the *key_check_value_length* variable is 8 or more. Otherwise, an error is returned.

When the selected key-check algorithm is to compute the MDC-2 hash of the key, and the generated or exported key is single length, the 8-byte key is made into a double-length key by replicating the key halves. This is because the MDC-2 calculation method does no padding, and requires that the data be a minimum of 16 bytes and a multiple of eight bytes. If the generated or exported key is double length or triple length, the key is processed as is. The verb returns the 16-byte hash result of the key in the *key_check_value* variable if the *key_check_value_length* variable is large enough, else an error is returned.

Restrictions

The restrictions for CSNDRKX.

- AES keys are not supported by this verb.
- Keys with a modulus length greater than 2048 bits are not supported in releases before Release 3.30.
- The maximum public exponent is 17 bits for any key that has a modulus greater than 2048 bits.
- A key identifier length must be 64 for a key label.
- Key-wrapping method keywords are not supported in releases before Release 4.4.
- A key-wrapping method rule-array keyword cannot be specified if the symmetric encrypted output key format flag is not set to return a CCA fixed-length DES key-token.

Required commands

The required commands for CSNDRKX.

This verb requires the **Remote Key Export - Gen or export a non-CCA node key** command (offset X'0312') to be enabled in the active role.

The verb also requires the **Key Generate - SINGLE-R** command (offset X'00DB') to be enabled to replicate a single-length source key (either from a CCA DES key-token or an RKX key-token). If authorized, key replication occurs if all of the following are true:

1. The key token returned using the *sym_encrypted_key_identifier* parameter is a CCA DES key-token, as defined in the rule section identified by the *rule_id* parameter.
2. The rule section identified by the *rule_id* parameter has a common export key parameters subsection defined, and the control vector in the subsection is 16 bytes in length with key-form bits of B'010' for the left half and B'001' for the right half.
3. The token identified by the *source_key_identifier* parameter is single length, and is either a CCA DES key-token or an RKX key-token.

Note: A role with X'00DB' enabled can also use the Key Generate verb with the SINGLE-R key-length keyword.

To enable the use of key-encrypting-keys with the NOCV option for export, this verb requires the **NOCV KEK usage for export-related functions** command (offset X'0300') to be enabled in the active role.

To enable the use of key-encrypting-keys with the NOCV option for import, this verb requires the **NOCV KEK usage for import-related functions** command (offset X'030A') to be enabled in the active role.

This verb also requires the following commands to be enabled in the active role:

Rule-array keyword	Offset	Command
USECONFG **WRAP-ECB** **WRAP-ENH** **ENH-ONLY**	X'013F'	**Remote Key Export - include RKX in default wrap config**
WRAP-ECB or **WRAP-ENH** and default key-wrapping method setting does not match keyword	X'02BA'	**Remote Key Export - Allow wrapping override keywords**

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Beginning with Release 4.4, commands related to wrapping DES keys are added. These commands coincide with the addition of rule array keywords to specify key-wrapping method and translation control. The commands added with Release 4.4 are:

- **Remote Key Export - include RKX in default wrap config** (offset X'013F')

  Enable this command in the active role to allow any rule array keywords to be specified or, in the absence of a key-wrapping method keyword, to have the verb wrap the DES key using the default key-wrapping configuration (USECONFG) setting. Enabling X'013F' makes the verb behave in the same manner as other verbs that accept key-wrapping method and translation control keywords beginning with Release 4.1.

  Note: The purpose of offset X'013F' is to provide a way to maintain backward compatibility with the key-wrapping method used in releases before Release 4.4. To maintain backward compatibility with releases that do not accept key-wrapping method keywords, do not enable X'013F' in the active role.
- **Remote Key Export - Allow wrapping override keywords** (offset X'02BA')

  This command requires offset X'013F' to be enabled in the active role. Enable offset X'02BA' in the active role to allow a key-wrapping keyword to be specified that overrides the default key-wrapping configuration.

When an existing key is exported into a CCA fixed-length DES key-token (that is, offset 18 in the selected trusted block rule section is X'01') and the **Remote Key Export - include RKX in default wrap config** command (offset X'013F') is not enabled in the active role (or the release is before Release 4.4), the DES key identified by the **sym_encrypted_key_identifier** parameter is wrapped based on the following criteria depending on whether the source key is an RKX key-token or a CCA fixed-length DES key-token:

- Source key is an RKX key-token:

Configuration setting of default wrapping method	CV bit 56 of rule subsection X'0003' of selected trusted block rule section	Wrapping method (and CV bit 56) of exported DES key
WRAP-ECB (legacy)	No CV (CV length = 0)	WRAP-ECB (B'0')
	B'0' (not ENH-ONLY)	WRAP-ECB (B'0')
	B'1' (ENH-ONLY)	Error
WRAP-ENH (enhanced)	No CV	WRAP-ENH (B'0')
	B'0'	WRAP-ENH (B'0')
	B'1'	WRAP-ENH (B'1')

- Source key is a CCA fixed-length DES key-token:

Source key wrapping method	CV bit 56 of source key	CV bit 56 of rule subsection X'0003' of selected trusted block rule section	Wrapping method (and CV bit 56) of exported DES key
WRAP-ECB (legacy)	B'0' (not ENH-ONLY)	No CV (CV length = 0)	WRAP-ECB (B'0')
		B'0' (not ENH-ONLY)	WRAP-ECB (B'0')
		B'1' (ENH-ONLY)	Error
WRAP-ENH (enhanced)	B'0'	No CV	WRAP-ENH (B'0')
		B'0'	WRAP-ENH (B'0')
		B'1'	WRAP-ENH (B'1')
	B'1' (ENH-ONLY)	No CV	WRAP-ENH (B'0')
		B'0'	Error
		B'1'	WRAP-ENH (B'1')

Usage notes

The usage notes for CSNDRKX.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDRKXJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNDRKXJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger trusted_block_length,
           byte[]            trusted_block_identifier,
           hikmNativeInteger certificate_length,
           byte[]            certificate,
           hikmNativeInteger certificate_parms_length,
           byte[]            certificate_parms,
           hikmNativeInteger transport_key_identifier_length,
```

```
byte[]            transport_key_identifier,
hikmNativeInteger rule_id_length,
byte[]            rule_id,
hikmNativeInteger export_key_kek_length,
byte[]            export_key_kek_identifier,
hikmNativeInteger export_key_length,
byte[]            export_key_identifier,
hikmNativeInteger asym_encrypted_key_length,
byte[]            asym_encrypted_key,
hikmNativeInteger sym_encrypted_key_length,
byte[]            sym_encrypted_key,
hikmNativeInteger extra_data_length,
byte[]            extra_data,
hikmNativeInteger key_check_parameters_length,
byte[]            key_check_parameters,
hikmNativeInteger key_check_length,
byte[]            key_check_value);
```

Trusted Block Create (CSNDTBC)

The verb creates an external trusted block under dual control. A trusted block is an extension of CCA PKA key tokens using new section identifiers.

Trusted blocks are an integral part of a remote key-loading process. They contain various items, some of which are optional, and some of which can be present in different forms. Tokens are composed of concatenated sections. For a detailed description of a trusted block, including its format and field values, see "Trusted blocks" on page 877.

Creating an external trusted block: Create an active external trusted block in two steps:

1. Create an inactive external trusted block using the **INACTIVE** *rule_array* keyword. This step requires the **Trusted Block Create - Create Block in inactive form** command (offset X'030F') to be enabled in the active role.
2. Complete the creation process by activating (promoting) an inactive external trusted block using the **ACTIVE** *rule_array* keyword. This step requires the **Trusted Block Create - Activate an inactive block** command (offset X'0310') to be enabled in the active role. Changing an external trusted block from inactive to active effectively approves the trusted block for further use.

Note: Authorize each command in a different role to enforce a dual-control policy.

The creation of an external trusted block typically takes place in a highly secure environment. Use "PKA Key Import (CSNDPKI)" on page 640 to import an active external trusted block into the desired node. The imported internal trusted block can then be used as input to "Remote Key Export (CSNDRKX)" on page 664 in order to generate or export DES keys.

Creating an inactive external trusted block: To create an inactive external trusted block, use a *rule_array_count* of 1 and a *rule_array* keyword of **INACTIVE**. Identify the input trusted block using the *input_block_identifier* parameter, and set the *input_block_identifier_length* variable to the length of the key label or the key token of the input block. The input block can be any one of these forms:

- An uninitialized trusted block. The trusted block is complete except that it does not have MAC protection.
- An inactive trusted block. The trusted block is external, and it is in inactive form. MAC protection is present due to recycling of an existing inactive trusted block.

- An active trusted block. The trusted block is internal or external, and it is in active form. MAC protection is present due to recycling of an existing active trusted block.

Note: The MAC key is replaced with a new MAC key, and any RKX key-token created with the input trusted block cannot be used with the output trusted block.

This verb randomly generates a confounder and triple-length MAC key, and uses a variant of the MAC key to calculate an ISO 16609 CBC mode TDES MAC of the trusted block contents. To protect the MAC key, the verb encrypts the confounder and MAC key using a variant of an **IMP-PKA** key. The calculated MAC and the encrypted confounder and MAC key are embedded in the output trusted block. Use the *transport_key_identifier* parameter to identify the key token that contains the **IMP-PKA** key.

On input, set the *trusted_block_identifier_length* variable to the length of the key label or at least the size of the output trusted block. The output trusted block is returned in the key-token identified by the *trusted_block_identifier* parameter, and the verb updates the *trusted_block_identifier_length* variable to the size of the key token if a key label is not specified.

Creating an active external trusted block: To create an active external trusted block, use a *rule_array_count* of 1 and a *rule_array* keyword of **ACTIVE**. Identify the input trusted block using the *input_block_identifier* parameter, and set the *input_block_identifier_length* variable to the length of the key label or the key token of the input block. The input block must be an inactive external trusted block that was created using the **INACTIVE** *rule_array* keyword.

Use the *transport_key_identifier* parameter to identify the key token that contains the **IMP-PKA** key.

On input, set the *trusted_block_identifier_length* variable to the length of the key label or at least the size of the output trusted block. The verb returns an error if the input trusted block is not valid. Otherwise, it changes the flag in the trusted block information section from the inactive state to the active state, recalculates the MAC, and embeds the updated MAC value in the output trusted block.

The output trusted block is returned in the key-token identified by the *trusted_block_identifier* parameter, and the verb updates the *trusted_block_identifier_length* variable to the size of the key token if a key label is not specified.

Format

The format of CSNDTBC.

```
CSNDTBC(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             input_block_identifier_length,
             input_block_identifier,
             transport_key_identifier,
             trusted_block_identifier_length,
             trusted_block_identifier)
```

Parameters

The parameters for CSNDTBC.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1.

rule_array

Direction: Input
Type: String array

A pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* keywords are described in Table 177.

Table 177. Keywords for Trusted Block Create control information

Keyword	Description
Operation (One required)	
INACTIVE	Create an external trusted block, based on the *input_block_identifier* variable, and set the active flag to B'0'. This makes the trusted block unusable in any other CCA services.
ACTIVE	Create an external trusted block, based on the token identified by the *input_block_identifier* parameter, and change the active flag from B'0' to B'1'. This makes the trusted block usable in other CCA services

input_block_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes in the *input_block_identifier* variable. The maximum length is 3500 bytes.

input_block_identifier

Direction: Input
Type: String

A pointer to a string variable containing a trusted block key-token or the key label of a trusted block key-token that has been built according to the format specified in "Trusted blocks" on page 877. The trusted block key-token will be updated by the verb and returned in the *trusted_block_identifier* variable.

When the operation is **INACTIVE**, the trusted block can have MAC protection (for example, due to recycling of an existing trusted block), but typically it does not.

transport_key_identifier

Direction: Input
Type: String

A pointer to a string variable containing an operational CCA DES key-token or

the key label of an operational CCA DES key-token record. The key token must be of type **IMP-PKA**.

An **IMP-PKA** key type is an **IMPORTER** key-encrypting key with only its **IMPORT** key-usage bit (bit 21) on; its other key-usage bits (IMEX, OPIM, IMIM, and XLATE) must be off.

Note: An **IMP-PKA** control vector can be built using "Control Vector Generate (CSNBCVG)" on page 134 with a key type of **IMPORTER** and a *rule_array* keyword of **IMPORT**.

trusted_block_identifier_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *trusted_block_identifier* variable. The maximum length is 3500 bytes. The output trusted block token can be up to seven bytes longer than the input trusted block token due to padding.

trusted_block_identifier

Direction: Output
Type: String

A pointer to a string variable containing a trusted block token or a label of a trusted block token returned by the verb.

Restrictions

The restrictions for CSNDRKX.

1. AES keys are not supported by this verb.
2. Keys with a modulus length greater than 2048 bits are not supported in releases before Release 3.30.

Required commands

The required commands for CSNDTBC.

The verb requires the following commands to be enabled in the active role based on the keyword specified for the operation rule:

rule_array **keyword**	**Offset**	**Command**
INACTIVE	X'030F'	**Trusted Block Create - Create Block in inactive form**
ACTIVE	X'0310'	**Trusted Block Create - Activate an inactive block**

To prevent a weaker transport key (key-encrypting key) from being used to encipher a triple-length MAC key into an external trusted block, enable the **TBC - Disallow triple-length MAC key**command (offset X'032E') to be enabled in the active role.

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNDRKX.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDTBCJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNDTBCJ(
           hikmNativeInteger return_code,
           hikmNativeInteger reason_code,
           hikmNativeInteger exit_data_length,
           byte[]            exit_data,
           hikmNativeInteger rule_array_count,
           byte[]            rule_array,
           hikmNativeInteger input_block_length,
           byte[]            input_block_identifier,
           byte[]            transport_key_identifier,
           hikmNativeInteger trusted_blokc_length,
           byte[]            trusted_blokc_identifier);
```

Chapter 14. TR-31 symmetric key management verbs

These verbs support TR-31 symmetric key management.

Key Export to TR31 (CSNBT31X)

Use the Key Export to TR31 verb to convert a proprietary CCA external or internal symmetric key-token and its attributes into a non-proprietary key block that is formatted under the rules of TR-31.

After being exported into a TR-31 key block, the key and its attributes are ready to be interchanged with any outside third party who uses TR-31. The verb takes as input either an external or internal fixed-length DES key-token that contains a DES or Triple-DES (TDES) key, along with an internal DES EXPORTER or OKEYXLAT key-encrypting key used to wrap the external TR-31 key block.

The Key Export to TR31 verb is analogous to the Key Export verb, except that Key Export to TR31 accepts an external or internal fixed-length DES key-token as input, instead of only an internal fixed-length DES key-token, and it translates the key to an external non-CCA format instead of an external fixed-length DES key-token. The purpose of both verbs is to export a DES key to another party.

An external-to-external translation would not normally be called an *export* or *import operation*. Instead, it would be called a *key translation*, and would be handled by a verb such as Key Translate2. For practical reasons, the export of an external CCA DES key-token to external TR-31 format is supported by the Key Export to TR31 verb, and the import of an external TR-31 key block to an external CCA DES key-token is supported by the TR31 Key Import verb.

Note that the Key Export to TR31 verb does not support the translation of an external key from encipherment under one key-encrypting key to encipherment under a different key-encrypting key. When converting an external DES key to an external TR-31 format, the key-encrypting key used to wrap the external source key must be the same as the one used to wrap the TR-31 key block. If a translation of an external DES key from encipherment under one key-encrypting to a different key-encrypting key is desired, use the Key Translate or Key Translate2 verbs.

Both CCA and TR-31 define key attributes that control key usage. In both cases, the usage information is securely bound to the key so that the attributes cannot be changed in unintended or uncontrolled ways. CCA maintains its DES key attributes in a control vector (CV), while a TR-31 key block uses fields: *key usage*, *algorithm*, *mode of use*, and *exportability*.

Each attribute in a CCA control vector falls under one of these categories:

1. There is a one-to-one correspondence between the CV attribute and the TR-31 attribute. For these attributes, conversion is straightforward.

2. There is not a one-to-one correspondence between the CV attribute and the TR-31 attribute, but the attribute can be automatically translated when performing this export operation.
3. There is not a one-to-one correspondence between the CV attribute and the TR-31 attribute, in which case a rule-array keyword is defined to specify which attribute is used in the TR-31 key block.
4. Category (1), (2), or (3) applies, but there are some attributes that are lost completely on translation (for example, key-generating bits in key-encrypting keys).
5. None of the above categories applies, because the key type, its attributes, or both simply cannot be reasonably translated into a TR-31 key block.

The control vector is always checked for compatibility with the TR-31 attributes. It is an error if the specified TR-31 attributes are in any way incompatible with the control vector of the input key. In addition, access control points are defined that can be used to restrict the permitted attribute conversions.

The TR-31 key block has a header that can contain optional blocks. Optional blocks become securely bound to the key by virtue of the MAC on the TR-31 key block. The *opt_blocks* parameter is provided to allow a complete and properly formatted optional block structure to be included as part of the TR-31 key block that is returned by the verb. The TR31 Optional Data Build (CSNBT31O) verb can be used to construct an optional block structure, one optional block at a time.

An optional block has a 2-byte ASCII block ID value that determines the use of the block. The use of a particular optional block is either defined by TR-31, or it has a proprietary use. An optional block that has a block ID with a numeric value is a proprietary block. IBM has its own proprietary optional block to contain a CCA control vector. See “TR-31 optional block data” on page 876 for a description of the IBM-defined data.

To include a copy of the control vector from the DES source key in an optional block of the TR-31 key block, specify the **ATTR-CV** or **INCL-CV** control vector transport control keyword in the rule array. If either optional keyword is specified, the verb copies the single-length or double-length control vector field from the source key into the optional data field of the TR-31 header. The TR31 Key Import verb can later extract this data and use it as the control vector for the CCA key that it creates when importing the TR-31 key block. This method provides a way to use TR-31 for transport of CCA keys and to make the CCA key have identical control vectors on the sending and receiving nodes.

The **ATTR-CV** and **INCL-CV** keywords both cause the control vector to be included in a TR-31 optional block, but each has a different purpose:

ATTR-CV
: Causes a copy of the control vector to be included, but both the TR-31 usage and mode of use fields in the non-optional part of the TR-31 key block header are set to IBM proprietary values. These values, described in “TR-31 optional block data” on page 876, indicate that the usage and mode information are specified in the control vector of the optional block and not in the TR-31 header. The restrictions imposed by the setting of the relevant access control points are bypassed, and any CCA key can be exported as long as the export control fields in the control vector allow it.

INCL-CV
: Causes a copy of the control vector to be included as additional detail. The

resulting attributes set in the non-optional part of the TR-31 key block header are identical to not using either keyword, except that the value for the number of optional blocks is increased by one. The export operation is still subject to the restrictions imposed by the settings of the relevant access control points.

Format

The format of CSNBT31X.

```
CSNBT31X(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_version_number,
             key_field_length,
             source_key_identifier_length,
             source_key_identifier,
             unwrap_kek_identifier_length,
             unwrap_kek_identifier,
             wrap_kek_identifier_length,
             wrap_kek_identifier,
             opt_blocks_length,
             opt_blocks,
             tr31_key_block_length,
             tr31_key_block)
```

Parameters

The parameters for CSNBT31X.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 2, 3, 4 or 5.

rule_array

Direction: Input
Type: String array

A pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* keywords for this verb are shown in Table 178.

Table 178. Keywords for Key Export to TR31 control information

Keyword	Meaning
Key block protection method (one required). Specifies which version of the TR-31 key block to use for exporting the CCA DES key. The version defines the method by which the key block is cryptographically protected and the content and layout of the block.	

Table 178. Keywords for Key Export to TR31 control information (continued)

Keyword	Meaning
VARXOR-A	Specifies to use the Key Variant Binding Method 2005 Edition. Corresponds to TR-31 Key Block Version ID of "A" (X'41').
VARDRV-B	Specifies to use the Key Derivation Binding Method 2010 Edition. Corresponds to TR-31 Key Block Version ID of "B" (X'42').
VARXOR-C	Specifies to use the Key Variant Binding Method 2010 Edition. Corresponds to TR-31 Key Block Version ID of "C" (X'43').
Control vector transport control (one, optional). If no keyword from this group is provided, or keyword **INCL-CV** is specified, the control vector in the CCA key token identified by the source_key_identifier parameter is verified to agree with the TR-31 key usage and mode of key use keywords specified from the groups below.	
INCL-CV	Specifies to copy the control vector from the CCA key-token into an optional proprietary block that is included in the TR-31 key block header. See Table 243 on page 877. The TR-31 key usage and mode of use fields indicate the key attributes. Those attributes, as derived from the keywords specified, must be compatible with the ones in the included CV. In addition, the export of the key must meet the translation and ACP authorizations indicated in the export translation table for the specified keywords. A key usage keyword and a mode of use keyword are required when this keyword is specified.
ATTR-CV	Same as keyword **INCL-CV**, except that the key usage field of the TR-31 key block (byte number 5 - 6) is set to the proprietary value "10" (X'3130'), and the mode of use field (byte number 8) is set to the proprietary value "1" (X'31'). These proprietary values indicate that the key usage and mode of use attributes are specified by the CV in the optional block. For this option, only the general ACPs related to export are checked, not the ones relating to specific CCA to TR-31 translations. No key usage or mode of use keywords are allowed when this keyword is specified.

TR-31 key usage value for output key (one required). Not valid if **ATTR-CV** keyword is specified. Only those TR-31 usages shown are supported.

Keyword	TR-31 key usage	CCA key types	Meaning
BDK	"B0"	KEYGENKY	Specifies to export to a TR-31 base derivation key (BDK). You must select one mode of use keyword from Table 179 on page 688 with this usage keyword. The table shows all of the supported translations for key usage keyword **BDK**. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.
CVK	"C0"	MAC or DATA	Specifies to export to a TR-31 CVK card verification key. You must select one mode of use keyword from Table 180 on page 690 with this usage keyword. The table shows all of the supported translations for key usage keyword **CVK**. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.
ENC	"D0"	ENCIPHER, DECIPHER, CIPHER, or DATA	Specifies to export to a TR-31 data encryption key. You must select one mode of use keyword from Table 181 on page 691 with this usage keyword. The table shows all of the supported translations for key usage keyword **ENC**. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.

Table 178. Keywords for Key Export to TR31 control information (continued)

Keyword	Meaning		
KEK	"K0"	EXPORTER or OKEYXLAT	Specifies to export to a TR-31 key-encryption or wrapping key. You must select one mode of use keyword from Table 182 on page 692 with this usage keyword. The table shows all of the supported translations for key usage keyword **KEK**. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.
KEK-WRAP	"K1"	IMPORTER or IKEXLAT	Specifies to export to a TR-31 key block protection key. You must select one mode of use keyword from Table 182 on page 692 with this usage keyword. The table shows all of the supported translations for key usage keyword **KEK-WRAP**. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.
ISOMAC0	"M0"	MAC, MACVER, DATA, DATAM, or DATAMV	Specifies to export to a TR-31 ISO 16609 MAC algorithm 1 (using TDEA) key. You must select one mode of use keyword from Table 183 on page 693 with this usage keyword. The table shows all of the supported translations for key usage keyword **ISOMAC0**. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.
ISOMAC1	"M1"	MAC, MACVER, DATA, DATAM, or DATAMV	Specifies to export to a TR-31 ISO 9797-1 MAC algorithm 1 key. You must select one mode of use keyword from Table 183 on page 693 with this usage keyword. The table shows all of the supported translations for key usage keyword **ISOMAC1**. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.
ISOMAC3	"M3"	MAC, MACVER, DATA, DATAM, or DATAMV	Specifies to export to a TR-31 ISO 9797-1 MAC algorithm 3 key. You must select one mode of use keyword from Table 183 on page 693 with this usage keyword. The table shows all of the supported translations for key usage keyword **ISOMAC3**. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.
PINENC	"P0"	OPINENC or IPINENC	Specifies to export to a TR-31 PIN encryption key. You must select one mode of use keyword from Table 185 on page 696 with this usage keyword. The table shows all of the supported translations for key usage keyword **PINENC**. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.

Table 178. Keywords for Key Export to TR31 control information (continued)

Keyword	Meaning		
PINVO	"V0"	PINGEN or PINVER	Specifies to export to a TR-31 PIN verification key or other algorithm. You must select one mode of use keyword from Table 185 on page 696 with this usage keyword. The table shows all of the supported translations for key usage keyword **PINVO**. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.
PINV3624	"V1"	PINGEN or PINVER	Specifies to export to a TR-31 PIN verification, IBM 3624 key. You must select one mode of use keyword from Table 185 on page 696 with this usage keyword. The table shows all of the supported translations for key usage keyword **PINV3624**. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.
VISAPVV	"V2"	PINGEN or PINVER	Specifies to export to a TR-31 PIN verification, VISA PVV key. You must select one mode of use keyword from Table 185 on page 696 with this usage keyword. The table shows all of the supported translations for key usage keyword **VISAPVV**. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.
EMVACMK	"E0"	DKYGENKY	Specifies to export to a TR-31 EMV/chip issuer master key: application cryptograms key. You must select one mode of use keyword from Table 186 on page 699 with this usage keyword. The table shows all of the supported translations for key usage keyword **EMVACMK**. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.
EMVSCMK	"E1"	DKYGENKY	Specifies to export to a TR-31 EMV/chip issuer master key: secure messaging for confidentiality key. You must select one mode of use keyword from Table 186 on page 699 with this usage keyword. The table shows all of the supported translations for key usage keyword **EMVSCMK**. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.
EMVSIMK	"E2"	DKYGENKY	Specifies to export to a TR-31 EMV/chip issuer master key: secure messaging for integrity key. You must select one mode of use keyword from Table 186 on page 699 with this usage keyword. The table shows all of the supported translations for key usage keyword **EMVSIMK**. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.

Table 178. Keywords for Key Export to TR31 control information (continued)

Keyword	Meaning		
EMVDAMK	"E3"	DATA, MAC, CIPHER, or ENCIPHER	Specifies to export to a TR-31 EMV/chip issuer master key: data authentication code key. You must select one mode of use keyword from Table 186 on page 699 with this usage keyword. The table shows all of the supported translations for key usage keyword **EMVDAMK**. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.
EMVDNMK	"E4"	DKYGENKY	Specifies to export to a TR-31 EMV/chip issuer master key: dynamic numbers key. You must select one mode of use keyword from Table 186 on page 699 with this usage keyword. The table shows all of the supported translations for key usage keyword **EMVDNMK**. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.
EMVCPMK	"E5"	DKYGENKY	Specifies to export to a TR-31 EMV/chip issuer master key: card personalization key. You must select one mode of use keyword from Table 186 on page 699 with this usage keyword. The table shows all of the supported translations for key usage keyword **EMVCPMK**. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.
TR-31 mode of key use (one required). Not valid if **ATTR-CV** keyword is specified. Only those TR-31 modes shown are supported.			
Keyword	**TR-31 mode**	**Usage keyword**	**Meaning**
ENCDEC	"B"	**ENC**, **KEK**, **KEK-WRAP**	Specifies both encrypt and decrypt, wrap and unwrap.
DEC-ONLY	"D"	**ENC**, **KEK**, **KEK-WRAP**, **PINENC**	Specifies to decrypt and unwrap only.
ENC-ONLY	"E"	**ENC**, **PINENC**	Specifies to encrypt and wrap only.
GENVER	"C"	**CVK**, **ISOMAC0**, **ISOMAC1**, **ISOMAC3**, **PINVO**, **PINV3624**, **VISAPVV**	Specifies to both generate and verify.
GEN-ONLY	"G"	**CVK**, **ISOMAC0**, **ISOMAC1**, **ISOMAC3**, **PINVO**, **PINV3624**, **VISAPVV**	Specifies to generate only.
VER-ONLY	"V"	**CVK**, **ISOMAC0**, **ISOMAC1**, **ISOMAC3**, **PINVO**, **PINV3624**, **VISAPVV**	Specifies to verify only.

Table 178. Keywords for Key Export to TR31 control information (continued)

Keyword	Meaning		
DERIVE	"X"	**BDK**, **EMVACMK**, **EMVSCMK**, **EMVSIMK**, **EMVDAMK**, **EMVDNMK**, **EMVCPMK**	Specifies that key is used to derive other keys.
ANY	"N"	**BDK**, **PINVO**, **PINV3624**, **VISAPVV**, **EMVACMK**, **EMVSCMK**, **EMVSIMK**, **EMVDAMK**, **EMVDNMK**, **EMVCPMK**	Specifies no special restrictions (other than restrictions implied by the key usage).
TR-31 exportability (one, optional). Use to set exportability field in TR-31 key block. Defines whether the key may be transferred outside the cryptographic domain in which the key is found.			
Keyword	**TR-31 byte**	**Meaning**	
EXP-ANY	"E"	Specifies that the key in the TR-31 key block is exportable under a key-encrypting key in a form that meets the requirements of X9.24 Parts 1 or 2. This is the default. **Note:** A TR-31 key block with a key block version ID of "B" or "C" and an exportability field value of "E" cannot be wrapped by a key-encrypting key that is wrapped in ECB mode (legacy wrap mode). This limitation is because ECB mode does not comply with ANSI X9.24 Part 1.	
EXP-TRST	"S"	Specifies that the key in the TR-31 key block is sensitive, exportable under a key-encrypting key in a form not necessarily meeting the requirements of X9.24 parts 1 or 2.	
EXP-NONE	"N"	Specifies that the key in the TR-31 key block is non-exportable.	

Note:

1. These keys are the base keys from which derived unique key per transaction (DUKPT) initial keys are derived for individual devices such as PIN pads.
2. This table defines the only supported translations for this TR-31 usage. Usage must be the following value:
   "B0" BDK base derivation key.
3. KEYGENKY keys are double length only.

Table 179. Export translation table for a TR-31 BDK base derivation key (BDK)

Key usage keyword	Key block protection method keyword	CCA key type and required control vector attributes	Mode of use keyword	Offset	Command
BDK ("B0")	**VARXOR-A**	KEYGENKY, double length, UKPT (CV bit 18 = B'1')	**ANY** ("N")	X'0180'	**TR31 Export - Permit KEYGENKY:UKPT to B0**
BDK ("B0")	**VARDRV-B**, **VARXOR-C**	KEYGENKY, double length, UKPT (CV bit 18 = B'1')	**DERIVE** ("X")	X'0180'	**TR31 Export - Permit KEYGENKY:UKPT to B0**

Security considerations:

1. There is asymmetry in the translation from a CCA DATA key to a TR-31 key. The asymmetry results from CCA DATA keys having attributes of both data encryption keys and MAC keys, while TR-31 separates data encryption keys from MAC keys. A CCA DATA key can be exported to a TR-31 "C0" key, if one

or both applicable MAC generate and MAC verify control vector bits are on. However, a TR-31 "C0" key cannot be imported to the lower-security CCA DATA key, it **can be imported only** to a CCA key type of MAC or MACVER. This restriction eliminates the ability to export a CCA MAC or MACVER key to a TR-31 key and re-importing it back as a CCA DATA key with the capability to Encipher, Decipher, or both.

2. Since the translation from TR-31 usage "C0" is controlled by rule array keywords when using the CSNBT31I verb, it is possible to convert an exported CCA CVVKEY-A or CVVKEY-B key into an AMEX-CSC key or the other way around. This conversion can be restricted by not enabling offsets X'015A' (TR31 Import - Permit C0 to MAC/MACVER:CVVKEY-A) and X'015B' (TR31 Import - Permit C0 to MAC/MACVER:AMEXCSC) at the same time. However, if both CVVKEY-x and AMEX-CSC translation types are required, then offsets X'015A' and X'015B' must be enabled. In this case, control is up to the development, deployment, and execution of the applications themselves.

Note:

1. Card verification keys are used for computing or verifying (against supplied value) a card verification code with the CVV, CVC, CVC2, and CVV2 algorithms. In CCA, these keys correspond to keys used with two algorithms:
   - Visa CVV and MasterCard CVC codes are generated and verified using the CVV Generate and CVV Verify verbs. These verbs require a key type of DATA or MAC/MACVER with a subtype extension (CV bits 0 - 3) of ANY-MAC, single-length CVVKEY-A and single-length CVVKEY-B, a double-length CVVKEY-A (see CVV Key Combine verb). The MAC generate and the MAC verify (CV bits 20 - 21) key usage values must be set appropriately.
   - American Express CSC codes are generated and verified using the Transaction Validation verb. This verb requires a key type of MAC or MACVER with a subtype extension of ANY-MAC or AMEX-CSC.
2. This table defines the only supported translations for this TR-31 usage. Usage must be the following value:
   "C0" CVK card verification key.
3. CCA and TR-31 represent CVV keys differently. These differences make representations between CCA and TR-31 incompatible. CCA represents the key-A and key-B keys as two 8-byte (single length) keys, while TR-31 represents these keys as one 16-byte (double length) key. Current Visa standards now require one 16-byte key. The CVV Generate and CVV Verify verbs have support added to accept one 16-byte CVV key, using left and right key parts as key-A and key-B. See “CVV Key Combine (CSNBCKC)” on page 476. This new verb provides a way to combine two single-length MAC-capable keys into one double-length CVV key.
4. Import and export of 8-byte CVVKEY-A and CVVKEY-B MAC/MACVER keys is allowed only using the IBM proprietary TR-31 usage and mode values ("10" and "1", respectively) to indicate encapsulation of the IBM control vector in an optional block, since the 8-byte CVVKEY-A is meaningless and useless as a TR-31 "C0" usage key of any mode.

Table 180. Export translation table for a TR-31 CVK card verification key (CVK)

Key usage keyword	Key block protection method keyword	CCA key type and required control vector attributes		Mode of use keyword	Offset	Command
CVK ("C0")	**VARXOR-A**, **VARDRV-B**, **VARXOR-C**	MAC, single or double length, AMEX-CSC (CV bits 0 - 3 = B'0100')	MAC generate on, MAC verify off (CV bits 20 - 21 = B'10')	**GEN-ONLY** ("G")	X'0181'	**TR31 Export - Permit MAC/MACVER:AMEX-CSC to C0:G/C/V**
			MAC generate off, MAC verify on (CV bits 20 - 21 = B'01')	**VER-ONLY** ("V")		
			MAC generate on, MAC verify on (CV bits 20 - 21 = B'11')	**GENVER** ("C")		
		MAC, double length, CVVKEY-A (CV bits 0 - 3 = B'0010')	MAC generate on, MAC verify off (CV bits 20 - 21 = B'10')	**GEN-ONLY** ("G")	X'0182'	**TR31 Export - Permit MAC/MACVER:CVV-KEYA to C0:G/C/V**
			MAC generate off, MAC verify on (CV bits 20 - 21 = B'01')	**VER-ONLY** ("V")		
			MAC generate on, MAC verify on (CV bits 20 - 21 = B'11')	**GENVER** ("C")		
		MAC, double length, ANY-MAC (CV bits 0 - 3 = B'0000')	MAC generate on, MAC verify off (CV bits 20 - 21 = B'10')	**GEN-ONLY** ("G")	X'0183'	**TR31 Export - Permit MAC/MACVER:ANY-MAC to C0:G/C/V**
			MAC generate off, MAC verify on (CV bits 20 - 21 = B'01')	**VER-ONLY** ("V")		
			MAC generate on, MAC verify on (CV bits 20 - 21 = B'11')	**GENVER** ("C")		
		DATA, double length	MAC generate on, MAC verify off (CV bits 20 - 21 = B'10')	**GEN-ONLY** ("G")	X'0184'	**TR31 Export - Permit DATA to C0:G/C**
			MAC generate on, MAC verify on (CV bits 20 - 21 = B'11')	**GENVER** ("C")		

Security consideration: There is asymmetry in the translation from a CCA DATA key to a TR-31 key. The asymmetry results from CCA DATA keys having attributes of both data encryption keys and MAC keys, while TR-31 separates data encryption keys from MAC keys. A CCA DATA key can be exported to a TR-31

"D0" key, if one or both applicable Encipher or Decipher control vector bits are on. However, a TR-31 "D0" key cannot be imported to the lower-security CCA DATA key, it **can be imported only** to a CCA key type of ENCIPHER, DECIPHER, or CIPHER. This restriction eliminates the ability to export a CCA DATA key to a TR-31 key and re-importing it back as a CCA DATA key with the capability to MAC generate and MAC verify.

Note:
1. Data encryption keys are used for the encryption and decryption of data.
2. This table defines the only supported translations for this TR-31 usage. Usage must be the following value:
   "D0" Data encryption

Table 181. Export translation table for a TR-31 data encryption key (ENC)

Key usage keyword	Key block protection method keyword	CCA key type and required control vector attributes	Mode of use keyword	Offset	Command
ENC ("D0")	**VARXOR-A**, **VARDRV-B**, **VARXOR-C**	ENCIPHER, single or double length	**ENC-ONLY** ("E")	X'0185'	**TR31 Export - Permit ENCIPHER/DECIPHER/ CIPHER to D0:E/D/B**
		DECIPHER, single or double length	**DEC-ONLY** ("D")		
		CIPHER, single or double length	**ENCDEC** ("B")		
		DATA, single or double length, Encipher on, Decipher on (CV bits 18 - 19 = B'11')	**ENCDEC** ("B")	X'0186'	**TR31 Export - Permit DATA to D0:B**

Security consideration: The CCA OKEYXLAT, EXPORTER, IKEYXLAT, or IMPORTER KEK translation to a TR-31 "K0" key with mode "B" (both wrap and unwrap) is not allowed for security reasons. Even with access-control point control, this capability would give an immediate path to turn a CCA EXPORTER key into a CCA IMPORTER key, and the other way around.

Note:
1. Key encryption or wrapping keys are used only to encrypt or decrypt other keys, or as a key used to derive keys that are used for that purpose.
2. This table defines the only supported translations for this TR-31 usage. Usage must be one of the following values:
   "K0" Key encryption or wrapping
   "K1" TR-31 key block protection key
3. CCA mode support is the same for version IDs "B" and "C", because the distinction between TR-31 "K0" and "K1" does not exist in CCA keys. CCA does not distinguish between targeted protocols, and so there is no good way to represent the difference. Also note that most wrapping mechanisms now involve derivation or key variation steps.

Table 182. Export translation table for a TR-31 key encryption or wrapping, or key block protection key (KEK or KEK-WRAP)

Key usage keyword	Key block protection method keyword	CCA key type and required control vector attributes	Mode of use keyword	Offset	Command
KEK ("K0")	**VARXOR-A**, **VARDRV-B**, **VARXOR-C**	EXPORTER, double length, EXPORT on (CV bit 21 = B'1')	**ENC-ONLY** ("E")	X'0187'	**TR31 Export - Permit EXPORTER/OKEYXLAT to K0:E**
		OKEYXLAT, double length			
		IMPORTER, double length, IMPORT on (CV bit 21 = B'1')	**DEC-ONLY** ("D")	X'0188'	**TR31 Export - Permit IMPORTER/IKEYXLAT to K0:D**
		IKEYXLAT, double length			
KEK-WRAP ("K1")	**VARDRV-B**, **VARXOR-C**	EXPORTER, double length, EXPORT on (CV bit 21 = B'1')	**ENC-ONLY** ("E")	X'0189'	**TR31 Export - Permit EXPORTER/OKEYXLAT to K1:E**
		OKEYXLAT, double length			
		IMPORTER, double length, IMPORT on (CV bit 21 = B'1')	**DEC-ONLY** ("D")	X'018A'	**TR31 Export - Permit IMPORTER/IKEYXLAT to K1:D**
		IKEYXLAT, double length			

Security consideration: There is asymmetry in the translation from a CCA DATA key to a TR-31 key. The asymmetry results from CCA DATA keys having attributes of both data encryption keys and MAC keys, while TR-31 separates data encryption keys from MAC keys. A CCA DATA key can be exported to a TR-31 "M0", "M1", or "M3" key, if one or both applicable MAC generate and MAC verify control vector bits are on. However, a TR-31 "M0", "M1", or "M3" key cannot be imported to the lower-security CCA DATA key, it **can be imported only** to a CCA key type of MAC or MACVER. This restriction eliminates the ability to export a CCA MAC or MACVER key to a TR-31 key and re-importing it back as a CCA DATA key with the capability to Encipher, Decipher, or both.

Note:

1. MAC keys are used to compute or verify a code for message authentication.
2. This table defines the only supported translations for this TR-31 usage. Usage must be one of the following values:
   "M0" ISO 16609 MAC algorithm 1, TDEA

   The ISO 16609 MAC algorithm 1 is based on ISO 9797. It is identical to "M1", except that it does not support 8-byte DES keys.
   "M1" ISO 9797 MAC algorithm 1

   The ISO 9797 MAC algorithm 1 is identical to "M0", except that it also supports 8-byte DES keys.
   "M3" ISO 9797 MAC algorithm 3

   The X9.19 style of Triple-DES MAC.
3. A CCA control vector has no bits defined to limit key usage by algorithm, such as CBC MAC (TR-31 usage "M0" and "M1") or X9.19 (TR-31 usage "M3"). When importing a TR-31 key block, the resulting CCA key token deviates from the restrictions of usages "M0", "M1", and "M3". Importing a TR-31 key block which allows MAC generation ("G" or "C") results in a control vector with the ANY-MAC attribute rather than for the restricted algorithm that is set in the TR-31 key block. The ANY-MAC attribute provides the same restrictions as what CCA currently uses for generating and verifying MACs.

Table 183. Export translation table for a TR-31 ISO MAC algorithm key (ISOMACn)

Key usage keyword	Key block protection method keyword	CCA key type and required control vector attributes	Mode of use keyword	Offset	Command
ISOMAC0 ("M0")	**VARXOR-A**, **VARDRV-B**, **VARXOR-C**	MAC, double length, MAC generate on (CV bit 20 = B'1')	**GEN-ONLY** ("G")	X'018B'	**TR31 Export - Permit MAC/DATA/DATAM to M0:G/C**
		DATA, double length, MAC generate on (CV bit 20 = B'1')			
		MAC, double length, MAC generate on, MAC verify on (CV bits 20 - 21 = B'11')	**GENVER** ("C")		
		DATAM, double length, MAC generate on, MAC verify on (CV bits 20 - 21 = B'11')			
		DATA, double length, MAC generate on, MAC verify on (CV bits 20 - 21 = B'11')			
		MACVER, double length, MAC generate off, MAC verify on (CV bits 20 - 21 = B'01')	**VER-ONLY** ("V")	X'018C'	**TR31 Export - Permit MACVER/DATAMV to M0:V**
		DATAMV, double length, MAC generate off, MAC verify on (CV bits 20 - 21 = B'01')			
ISOMAC1 ("M1")	**VARXOR-A**, **VARDRV-B**, **VARXOR-C**	MAC, single or double length, MAC generate on (CV bit 20 = B'1')	**GEN-ONLY** ("G")	X'018D'	**TR31 Export - Permit MAC/DATA/DATAM to M1:G/C**
		DATA, single or double length, MAC generate on (CV bit 20 = B'1')			
		MAC, single or double length, MAC generate on, MAC verify on (CV bits 20 - 21 = B'11')	**GENVER** ("C")		
		DATAM, single or double length, MAC generate on, MAC verify on (CV bits 20 - 21 = B'11')			
		DATA, single or double length, MAC generate on, MAC verify on (CV bits 20 - 21 = B'11')			
		MACVER, single or double length, MAC generate off, MAC verify on (CV bits 20 - 21 = B'01')	**VER-ONLY** ("V")	X'018E'	**TR31 Export - Permit MACVER/DATAMV to M1:V**
		DATAMV, single or double length, MAC generate off, MAC verify on (CV bits 20 - 21 = B'01')			

Table 183. Export translation table for a TR-31 ISO MAC algorithm key (ISOMACn) (continued)

Key usage keyword	Key block protection method keyword	CCA key type and required control vector attributes	Mode of use keyword	Offset	Command
ISOMAC3 ("M3")	**VARXOR-A**, **VARDRV-B**, **VARXOR-C**	MAC, single or double length, MAC generate on (CV bit 20 = B'1')	**GEN-ONLY** ("G")	X'018F'	**TR31 Export - Permit MAC/DATA/DATAM to M3:G/C**
		DATA, single or double length, MAC generate on (CV bit 20 = B'1')			
		MAC, single or double length, MAC generate on, MAC verify on (CV bits 20 - 21 = B'11')	**GENVER** ("C")		
		DATAM, single or double length, MAC generate on, MAC verify on (CV bits 20 - 21 = B'11')			
		DATA, single or double length, MAC generate on, MAC verify on (CV bits 20 - 21 = B'11')			
		MACVER, single or double length, MAC generate off, MAC verify on (CV bits 20 - 21 = B'01')	**VER-ONLY** ("V")	X'0190'	**TR31 Export - Permit MACVER/DATAMV to M3:V**
		DATAMV, single or double length, MAC generate off, MAC verify on (CV bits 20 - 21 = B'01')			

Security considerations:

1. It is highly recommended that the **INCL-CV** keyword be used when exporting PINGEN, PINVER, IPINENC, or OPINENC keys. Using this keyword ensures that importing the TR-31 key block back into CCA will have the desired attributes.
2. TR-31 key blocks that are protected under legacy version ID "A" (keyword VARXOR-A, using the Key Variant Binding Method 2005 Edition) use the same mode of use "N" (keyword ANY) for PINGEN and PINVER keys. For version ID "A" keys only, for a given PIN key usage, enabling both the PINGEN and PINVER access-control points at the same time while enabling offset X'01B0' (for mode "N") is NOT recommended. In other words, for a particular PIN verification usage, you should not simultaneously enable the four commands shown below for that usage:

Table 184. Commands

Key type, mode, or version	Offset	Command
"V0": For usage V0, a user with the following four commands enabled in the active role can change a PINVER key into a PINGEN key and the other way round. **Avoid simultaneously enabling these four commands.**		
Key type PINVER	X'0193'	**TR31 Export - Permit PINVER:NO-SPEC to V0**
Key type PINGEN	X'0194'	**TR31 Export - Permit PINGEN:NO-SPEC to V0**
Mode **ANY**	X'01B0'	**TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N**
Version **VARXOR-A**	X'014D'	**TR31 Export - Permit Version A TR-31 Key Blocks**

Table 184. Commands (continued)

Key type, mode, or version	Offset	Command
"V1": For usage V1, a user with the following four commands enabled in the active role can change a PINVER key into a PINGEN key and the other way around. **Avoid simultaneously enabling these four commands.**		
Key type PINVER	X'0195'	**TR31 Export - Permit PINVER:NO-SPEC/IBM-PIN/IBM-PINO to V1**
Key type PINGEN	X'0196'	**TR31 Export - Permit PINGEN:NO-SPEC/IBM-PIN/IBM-PINO to V1**
Mode **ANY**	X'01B0'	**TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N**
Version **VARXOR-A**	X'014D'	**TR31 Export - Permit Version A TR-31 Key Blocks**
"V2": For usage V2, a user with the following four commands enabled in the active role can change a PINVER key into a PINGEN key and the other way around. **Avoid simultaneously enabling these four commands.**		
Key type PINVER	X'0197'	**TR31 Export - Permit PINVER:NO-SPEC/VISA-PVV to V2**
Key type PINGEN	X'0198'	**TR31 Export - Permit PINGEN:NO-SPEC/VISA-PVV to V2**
Mode **ANY**	X'01B0'	**TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N**
Version **VARXOR-A**	X'014D'	**TR31 Export - Permit version A TR-31 key blocks**

Failure to comply with this recommendation allows changing PINVER keys into PINGEN and the other way around.

Note:

1. PIN encryption keys are used to protect PIN blocks. PIN verification keys are used to generate or verify a PIN using a particular PIN-calculation method for that key type.
2. This table defines the only supported translations for this TR-31 usage. Usage must be one of the following values:
   "P0" PIN encryption
   "V0" PIN verification, KPV, other algorithm

   Usage "V0" is intended to be a PIN-calculation method "other" than those methods defined for "V1" or "V2". Because CCA does not have a PIN-calculation method of "other" defined, it maps usage "V0" to the subtype extension of NO-SPEC (CV bits 0 - 3 = B'0000'). Be aware that NO-SPEC allows any method, including "V1" and "V2", and that this mapping is suboptimal.
   "V1" PIN verification, IBM 3624
   "V2" PIN verification, Visa PVV
3. Mode must be one of the following values:
   "E" Encrypt/wrap only

   This mode restricts PIN encryption keys to encrypting a PIN block. May be used to create or reencipher an encrypted PIN block (for key-to-key translation).
   "D" Decrypt/unwrap only

   This mode restricts PIN encryption keys to decrypting a PIN block. Generally used in a PIN translation to decrypt the incoming PIN block.
   "N" No special restrictions (other than restrictions implied by the key usage)

   This mode is used by several vendors for a PIN generate or PIN verification key when the key block version ID is "A".

"G" Generate only

This mode is used for a PINGEN key that may not perform a PIN verification. This mode is the only mode available when the control vector in the CCA key-token (applicable when **INCL-CV** keyword is not provided) does **NOT** have the EPINVER control vector bit on.

"V" Verify only

This mode is used for PIN verification only. This mode is the only mode available when the control vector in the CCA key-token (applicable when **INCL-CV** is not provided) **ONLY** has the EPINVER control vector usage bit on (CV bits 18 - 22 = B'00001').

"C" Both generate and verify (combined)

This mode is the only output mode available for TR-31 when any of the CCA key-token PIN generating bits are on in the control vector (CPINGENA, EPINGENA, EPINGEN, or CPINGENA) in addition to the EPINVER bit.

Table 185. Export translation table for a TR-31 PIN encryption or PIN verification key (PINENC, PINVO, PINV3624, VISAPVV)

Key usage keyword	Key block protection method keyword	CCA key type and required control vector attributes	Mode of use keyword	Offset	Command
PINENC ("P0")	**VARXOR-A**, **VARDRV-B**, **VARXOR-C**	OPINENC, double length	**ENC-ONLY** ("E")	X'0191'	**TR31 Export - Permit OPINENC to P0:E**
		IPINENC, double length	**DEC-ONLY** ("D")	X'0192'	**TR31 Export - Permit IPINENC to P0:D**
PINVO ("V0")	**VARXOR-A**	PINVER, double length, NO-SPEC (CV bits 0 - 4 = B'0000')	**ANY** ("N") (requires both commands)	X'0193'	**TR31 Export - Permit PINVER:NO-SPEC to V0**
				X'01B0'	**TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N**
	VARXOR-A, **VARDRV-B**, **VARXOR-C**	PINVER, double length, NO-SPEC (CV bits 0 - 4 = B'0000'), CPINGEN off, EPINGENA off, EPINGEN off, CPINGENA off (CV bits 18 - 21 = B'0000')	**VER-ONLY** ("V")	X'0193'	**TR31 Export - Permit PINVER:NO-SPEC to V0**
	VARXOR-A	PINGEN, double length, NO-SPEC (CV bits 0 - 4 = B'0000')	**ANY** ("N") (requires both commands)	X'0194'	**TR31 Export - Permit PINGEN:NO-SPEC to V0**
				X'01B0'	**TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N**
	VARXOR-A, **VARDRV-B**, **VARXOR-C**	PINGEN, double length, NO-SPEC (CV bits 0 - 4 = B'0000'), EPINVER off (CV bit 22 = B'0')	**GEN-ONLY** ("G")	X'0194'	**TR31 Export - Permit PINGEN:NO-SPEC to V0**
		PINGEN, double length, NO-SPEC (CV bits 0 - 4 = B'0000'), EPINVER on (CV bit 22 = B'1')	**GENVER** ("C")		

Table 185. Export translation table for a TR-31 PIN encryption or PIN verification key (PINENC, PINVO, PINV3624, VISAPVV) (continued)

Key usage keyword	Key block protection method keyword	CCA key type and required control vector attributes	Mode of use keyword	Offset	Command
PINV3624 ("V1")	**VARXOR-A**	PINVER, double length, NO-SPEC or IBM-PIN/IBM-PINO (CV bits 0 - 4 = B'0000' or B'0001')	**ANY** ("N") (requires both commands)	X'0195'	**TR31 Export - Permit PINVER:NO-SPEC/IBM-PIN/ IBM-PINO to V1**
				X'01B0'	**TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N**
	VARXOR-A, **VARDRV-B**, **VARXOR-C**	PINVER, double length, NO-SPEC or IBM-PIN/IBM-PINO (CV bits 0 - 4 = B'0000' or B'0001'), CPINGEN off, EPINGENA off, EPINGEN off, CPINGENA off (CV bits 18 - 21 = B'0000')	**VER-ONLY** ("V")	X'0195'	**TR31 Export - Permit PINVER:NO-SPEC/IBM-PIN/ IBM-PINO to V1**
	VARXOR-A	PINGEN, double length, NO-SPEC or IBM-PIN/IBM-PINO (CV bits 0 - 4 = B'0000' or B'0001')	**ANY** ("N") (requires both commands)	X'0196'	**TR31 Export - Permit PINGEN:NO-SPEC/IBM-PIN/ IBM-PINO to V1**
				X'01B0'	**TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N**
	VARXOR-A, **VARDRV-B**, **VARXOR-C**	PINGEN, double length, NO-SPEC or IBM-PIN/IBM-PINO (CV bits 0 - 4 = B'0000' or B'0001'), EPINVER off (CV bit 22 = B'0')	**GEN-ONLY** ("G")	X'0196'	**TR31 Export - Permit PINGEN:NO-SPEC/IBM-PIN/ IBM-PINO to V1**
		PINGEN, double length, NO-SPEC or IBM-PIN/IBM-PINO (CV bits 0 - 4 = B'0000' or B'0001'), EPINVER on (CV bit 22 = B'1')	**GENVER** ("C")		

Table 185. Export translation table for a TR-31 PIN encryption or PIN verification key (PINENC, PINVO, PINV3624, VISAPVV) (continued)

Key usage keyword	Key block protection method keyword	CCA key type and required control vector attributes	Mode of use keyword	Offset	Command
VISAPVV ("V2")	**VARXOR-A**	PINVER, double length, NO-SPEC or VISA-PVV (CV bits 0 - 4 = B'0000' or B'0010')	**ANY** ("N") (requires both commands)	X'0197'	**TR31 Export - Permit PINVER:NO-SPEC/VISA-PVV to V2**
				X'01B0'	**TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N**
	VARXOR-A, **VARDRV-B**, **VARXOR-C**	PINVER, double length, NO-SPEC or VISA-PVV (CV bits 0 - 4 = B'0000' or B'0010'), CPINGEN off, EPINGENA off, EPINGEN off, CPINGENA off (CV bits 18 - 21 = B'0000')	**VER-ONLY** ("V")	X'0197'	**TR31 Export - Permit PINVER:NO-SPEC/VISA-PVV to V2**
	VARXOR-A	PINGEN, double length, NO-SPEC or VISA-PVV (CV bits 0 - 4 = B'0000' or B'0010')	**ANY** ("N") (requires both commands)	X'0198'	**TR31 Export - Permit PINGEN:NO-SPEC/VISA-PVV to V2**
				X'01B0'	**TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N**
	VARXOR-A, **VARDRV-B**, **VARXOR-C**	PINGEN, double length, NO-SPEC or VISA-PVV (CV bits 0 - 4 = B'0000' or B'0010'), EPINVER off (CV bit 22 = B'0')	**GEN-ONLY** ("G")	X'0198'	**TR31 Export - Permit PINGEN:NO-SPEC/VISA-PVV to V2**
		PINGEN, double length, NO-SPEC or VISA-PVV (CV bits 0 - 4 = B'0000' or B'0010'), EPINVER on (CV bit 22 = B'1')	**GENVER** ("C")		

Note:

1. EMV/chip issuer master-key keys are used by the chip cards to perform cryptographic operations or, in some cases, to derive keys used to perform operations. In CCA, these keys are (a) diversified key-generating keys (key type DKYGENKY), allowing derivation of operational keys, or (b) operational keys. Note that in this context, "master key" has a different meaning than for CCA. These master keys, also called KMCs, are described by EMV as DES master keys for personalization session keys. They are used to derive the corresponding chip card master keys, and not typically used directly for cryptographic operations other than key derivation. In CCA, these keys are usually key generating keys with derivation level DKYL1 (CV bits 12 - 14 = B'001'), used to derive other key generating keys (the chip card master keys). For some cases, or for older EMV key derivation methods, the issuer master keys could be level DKYL0 (CV bits 12 - 14 = B'000').
2. This table defines the only supported translations for this TR-31 usage. Usage must be one of the following values:
   "E0" Application cryptograms
   "E1" Secure messaging for confidentiality
   "E2" Secure messaging for integrity
   "E3" Data authentication code
   "E4" Dynamic numbers

"E5" Card personalization

3. EMV support in CCA is different than TR-31 support, and CCA key types do not match TR-31 types.
4. DKYGENKY keys are double length only.

Table 186. Export translation table for a TR-31 EMV/chip issuer master-key key (DKYGENKY, DATA)

Key usage keyword	Key block protection method keyword	CCA key type and required control vector attributes	Mode of use keyword	Offset	Command
EMVACMK ("E0")	**VARXOR-A**	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMAC (CV bits 19 - 22 = B'0010')	**ANY** ("N")	X'0199'	**TR31 Export - Permit DKYGENKY:DKYL0 +DMAC to E0**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMV (CV bits 19 - 22 = B'0011')	**ANY** ("N")	X'019A'	**TR31 Export - Permit DKYGENKY:DKYL0 +DMV to E0**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DALL (CV bits 19 - 22 = B'1111')	**ANY** ("N")	X'019B'	**TR31 Export - Permit DKYGENKY:DKYL0 +DALL to E0**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DMAC (CV bits 19 - 22 = B'0010')	**ANY** ("N")	X'019C'	**TR31 Export - Permit DKYGENKY:DKYL1 +DMAC to E0**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DMV (CV bits 19 - 22 = B'0011')	**ANY** ("N")	X'019D'	**TR31 Export - Permit DKYGENKY:DKYL1 +DMV to E0**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DALL (CV bits 19 - 22 = B'1111')	**ANY** ("N")	X'019E'	**TR31 Export - Permit DKYGENKY:DKYL1 +DALL to E0**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
EMVSCMK ("E1")	**VARXOR-A**	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DDATA (CV bits 19 - 22 = B'0001')	**ANY** ("N")	X'019F'	**TR31 Export - Permit DKYGENKY:DKYL0 +DDATA to E1**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	DKYGENKY, double length, DKYL0 (CV bits 12 - 144 = B'000'), DMPIN (CV bits 19 - 22 = B'1001')	**ANY** ("N")	X'01A0'	**TR31 Export - Permit DKYGENKY:DKYL0 +DMPIN to E1**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DALL (CV bits 19 - 22 = B'1111')	**ANY** ("N")	X'01A1'	**TR31 Export - Permit DKYGENKY:DKYL0 +DALL to E1**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DDATA (CV bits 19 - 2 = B'0001')	**ANY** ("N")	X'01A2'	**TR31 Export - Permit DKYGENKY:DKYL1 +DDATA to E1**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DMPIN (CV bits 19 - 22 = B'1001')	**ANY** ("N")	X'01A3'	**TR31 Export - Permit DKYGENKY:DKYL1 +DMPIN to E1**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DALL (CV bits 19 - 22 = B'1111')	**ANY** ("N")	X'01A4'	**TR31 Export - Permit DKYGENKY:DKYL1 +DALL to E1**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		

Table 186. Export translation table for a TR-31 EMV/chip issuer master-key key (DKYGENKY, DATA) (continued)

Key usage keyword	Key block protection method keyword	CCA key type and required control vector attributes	Mode of use keyword	Offset	Command
EMVSIMK ("E2")	**VARXOR-A**	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMAC (CV bits 19 - 22 = B'0010')	**ANY** ("N")	X'01A5'	**TR31 Export - Permit DKYGENKY:DKYL0 +DMAC to E2**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DALL (CV bits 19 - 22 = B'1111')	**ANY** ("N")	X'01A6'	**TR31 Export - Permit DKYGENKY:DKYL0 +DALL to E2**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DMAC (CV bits 19 - 22 = B'0010')	**ANY** ("N")	X'01A7'	**TR31 Export - Permit DKYGENKY:DKYL1 +DMAC to E2**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DALL (CV bits 19 - 22 = B'1111')	**ANY** ("N")	X'01A8'	**TR31 Export - Permit DKYGENKY:DKYL1 +DALL to E2**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
EMVDAMK ("E3")	**VARXOR-A**	DATA, double length	**ANY** ("N")	X'01A9'	**TR31 Export - Permit DATA/MAC/CIPHER/ ENCIPHER to E3**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	MAC (not MACVER), double length	**ANY** ("N")		
	VARXOR-A		**GEN-ONLY** ("G")		
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	CIPHER, double length	**ANY** ("N")		
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	ENCIPHER, double length	**ANY** ("N")		
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
EMVDNMK ("E4")	**VARXOR-A**	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DDATA (CV bits 19 - 22 = B'0001')	**ANY** ("N")	X'01AA'	**TR31 Export - Permit DKYGENKY:DKYL0 +DDATA to E4**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DALL (CV bits 19 - 22 = B'1111')	**ANY** ("N")	X'01AB'	**TR31 Export - Permit DKYGENKY:DKYL0 +DALL to E4**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		

Table 186. Export translation table for a TR-31 EMV/chip issuer master-key key (DKYGENKY, DATA) (continued)

Key usage keyword	Key block protection method keyword	CCA key type and required control vector attributes	Mode of use keyword	Offset	Command
EMVCPMK ("E5")	**VARXOR-A**	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DEXP (CV bits 19 - 22 = B'0101')	**ANY** ("N")	X'01AC'	**TR31 Export - Permit DKYGENKY:DKYL0 +DEXP to E5**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMAC (CV bits 19 - 22 = B'0010')	**ANY** ("N")	X'01AD'	**TR31 Export - Permit DKYGENKY:DKYL0 +DMAC to E5**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DDATA (CV bits 19 - 22 = B'0001')	**ANY** ("N")	X'01AE'	**TR31 Export - Permit DKYGENKY:DKYL0 +DDATA to E5**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		
	VARXOR-A	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DALL (CV bits 19 - 22 = B'1111')	**ANY** ("N")	X'01AF'	**TR31 Export - Permit DKYGENKY:DKYL0 +DALL to E5**
	VARDRV-B, **VARXOR-C**		**DERIVE** ("X")		

key_version_number

Direction: Input
Type: String

A pointer to a string variable containing two numeric ASCII bytes that are copied into the key version number field of the output TR-31 key block. Use a value of "00" (X'3030') if no key version number is needed.

This value is ignored If the key identified by the *source_key_identifier* parameter contains a partial key, that is, the KEY-PART bit (CV bit 44) is on in the control vector. When a partial key is passed, the verb sets the key version number field in the TR-31 key block to "c0" (X'6330'). According to TR-31, this value indicates that the TR-31 key block contains a component of a key (key part).

key_field_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the length of the key field that is encrypted in the TR-31 block. The length must be a multiple the DES cipher block size, which is eight. It must also be greater than or equal to the length of the clear text key passed using the *source_key_identifier* parameter plus the length of the key length field (two bytes) that precedes this key in the TR-31 block. For example, if the source key is a double-length TDES key (its length is 16 bytes), then the key field length must be greater than or equal to (16 + 2) bytes, and must also be a multiple of 8. This means that the minimum *key_field_length* in this case would be 24.

TR-31 allows a variable number of padding bytes to follow the clear text key, and the application designer can choose to pad with more than the minimum number of bytes needed to form a block that is a multiple of 8. This padding is generally done to hide the length of the clear text key from those who cannot decipher that key. Most often, all keys (single, double, or triple length) are padded to the same length so that it is not possible to determine which length is carried in the TR-31 block by examining the encrypted block.

Note: This parameter is not expected to allow for ASCII encoding of the encrypted data stored in the key field according to the TR-31 specification. For example, when a value of 24 is passed here, following the minimum example above, the length of the final ASCII-encoded encrypted data in the key field in the output TR-31 key block is 48 bytes.

source_key_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the length in bytes of the *source_key_identifier* variable. The value must be 64.

source_key_identifier

Direction: Input
Type: String

A pointer to a string variable containing either the key label for the source key, or the key token containing the source key. The source key is the key that is to be exported. The key must be a CCA fixed-length DES internal or external key-token. If the source key is an external token, an identifier for the KEK that wraps the source key must be identified by the *unwrap_kek_identifier* parameter. TR-31 currently supports only DES and TDES keys. AES is not supported.

unwrap_kek_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the length in bytes of the *unwrap_kek_identifier* variable. The value must be greater than or equal to 0. A null key-token can have a length of 1. Set this value to 64 for a key label or a KEK.

unwrap_kek_identifier

Direction: Input
Type: String

A pointer to a string variable containing either the key label for the source key KEK, or the key token containing the source key KEK when the source key is an external CCA key token, and a NULL key token otherwise. The source key KEK can also be the wrapping key for the key that is to be exported if the *wrap_kek_identifier* is not specified. The source key KEK must be a CCA internal DES KEK token of type EXPORTER or OKEYXLAT.

Note: ECB-mode wrapped DES keys (CCA legacy wrap mode) cannot be used to wrap or unwrap TR-31 "B" or "C" key blocks that have or will have "E" exportability, because ECB mode does not comply with ANSI X9.24 Part 1.

wrap_kek_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the length in bytes of the *wrap_kek_identifier* variable. Set this value to 64.

wrap_kek_identifier

Direction: Input

Type: String

A pointer to a string variable containing an operational fixed-length DES key-token with a key type of EXPORTER or OKEYXLAT to use for wrapping the output TR-31 key block, a null key token, or a key label of such a key in DES key-storage. If the identified key token is not null, the key token must have the same key that is in the key-token identified by the *unwrap_kek_identifier* parameter. If it is null, then the key identified by the *unwrap_kek_identifier* parameter is also used for wrapping the output TR-31 key block.

Note: ECB-mode wrapped DES keys (CCA legacy wrap mode) cannot be used to wrap or unwrap TR-31 "B" or "C" key blocks that have or will have "E" exportability, because ECB-mode does not comply with ANSI X9.24 Part 1. This parameter exists to allow for KEK separation. It is possible that KEKs are restricted as to what they can wrap, such that a KEK for wrapping CCA external keys might not be usable for wrapping TR-31 external keys, or the other way around.

opt_blocks_length

Direction: Input
Type: Integer

A pointer to an integer variable that specifies the length in bytes of the *opt_blocks* variable. If no optional data is to be included in the TR-31 key block, set this value to zero.

opt_blocks

Direction: Input
Type: String

A pointer to a string variable containing optional blocks data that is to be included in the output TR-31 key block. The optional blocks data can be constructed using the TR31 Optional Data Build verb.

Note: The Padding Block, ID "PB" cannot be added by the user, and therefore is not accepted in the *opt_blocks* parameter. CCA adds a Padding Block of the appropriate size as needed when building the TR-31 key block in Key Export to TR31. The Padding Block for optional blocks serves no security purpose, unlike the padding in the encrypted key portion of the payload.

tr31_key_block_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the length in bytes of the *tr31_key_block* variable. On input, specify the size of the application program buffer available for the output key-token. On return from the verb, this variable is updated to contain the actual length of that returned token. TR-31 key blocks are variable in length.

tr31_key_block

Direction: Output
Type: String

A pointer to a string variable containing the output key block produced by the verb. The output key block contains the external form of the key created by the

verb, wrapped according to the method specified.

Note: The padding optional block in the output TR-31 key block can be present with zero data bytes. This situation can occur if the optional block portion of the header needs exactly four bytes of padding, the size of an optional block header without the data portion. The data portion is defined as optional by TR-31, which allows this.

Restrictions

The restrictions for CSNBT31X.

- The only proprietary values for the TR-31 header fields supported by this verb are those values defined and used by IBM CCA when carrying a control vector in an optional block in the header.
- AES is not currently supported for TR-31 key blocks.
- The export is prohibited if the CCA key does not have attributes XPORT-OK (CV bit 17 = B'1') and T31XPTOK (CV bit 57 = B'1').

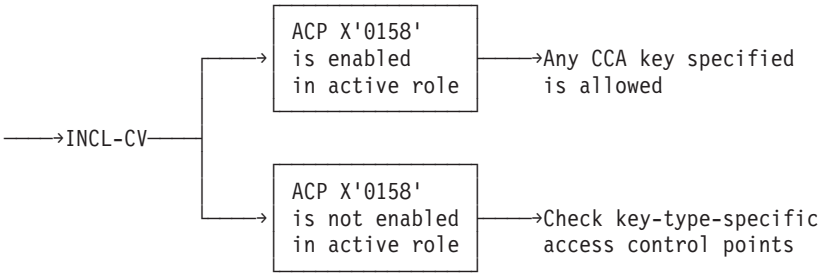
Required commands

The required commands for CSNBT31X.

The Key Export to TR31 verb requires the following commands to be enabled in the active role:

Rule-array keyword	Offset	Command
TR-31 key block protection method		
VARXOR-A	X'014D'	**TR31 Export - Permit version A TR-31 key blocks**
VARDRV-B	X'014E'	**TR31 Export - Permit version B TR-31 key blocks**
VARXOR-C	X'014F'	**TR31 Export - Permit version C TR-31 key blocks**
INCL-CV	X'0158'	**TR31 Export - Permit any CCA key if INCL-CV is specified** When providing the **INCL-CV** keyword: • If this command is enabled in the active role, the key-type specific commands are not checked. • If this command is not enabled in the active role, the key-type specific commands are required.
ATTR-CV	N/A	**Note:** No commands relating to specific CCA to TR-31 transitions are checked when this keyword is specified. Only the general access control commands related to export are checked.

Be aware of the interaction of access-control point X'0158' (**TR31 Export - Permit any CCA key if INCL-CV is specified**) with the **INCL-CV** keyword. Without the **INCL-CV** keyword, most export translations are guarded by key-type-specific access control points, to guard the source CCA system against attacks involving re-import of the exported key under ambiguous circumstances. When the control vector is exported along with the key as an optional block securely bound to the encrypted key, the source system is somewhat protected because the key on import is allowed to have only the form of the included control vector. No expansion of capability is allowed. If access-control point X'0158' is enabled in the active role, the key-type-specific access control points are not checked when **INCL-CV** is provided. If ACP X'0158' is not enabled, the type-specific access control points are still required. The following figure illustrates this concept:



Be aware of access-control point X'01B0' (**TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N**) for export of PINGEN or PINVER keys to wrapping method A , usage V0, V1, or V2, and mode N. TR-31 key blocks with legacy key usage A (key block protected using the Key Variant Binding Method 2005 Edition) use the same mode N for PINGEN as well as PINVER keys. For usage A keys only, enabling a PINGEN and PINVER access-control point while enabling offset X'01B0' (for keyword **ANY**, mode N) is **NOT** recommended. Failure to comply with this recommendation allows changing PINVER keys into PINGEN and the other way around.

In addition to the above commands, the verb requires these additional commands to be enabled in the active role depending on the TR-31 key usage rule-array keyword provided and additional information as shown in the table referenced in the rightmost column:

TR-31 key usage keyword	Offset	Command	Specific key type and control vector attributes
"B0": TR-31 BDK base derivation keys			
BDK	X'0180'	**TR31 Export - Permit KEYGENKY:UKPT to B0**	See Table 179 on page 688.
"C0": TR-31 CVK card verification keys			
CVK	X'0181'	**TR31 Export - Permit MAC/MACVER:AMEX-CSC to C0:G/C/V**	See Table 180 on page 690.
	X'0182'	**TR31 Export - Permit MAC/MACVER:CVV-KEYA to C0:G/C/V**	
	X'0183'	**TR31 Export - Permit MAC/MACVER:ANY-MAC to C0:G/C/V**	
	X'0184'	**TR31 Export - Permit DATA to C0:G/C**	
"D0": TR-31 data encryption keys			
ENC	X'0185'	**TR31 Export - Permit ENCIPHER/DECIPHER/CIPHER to D0:E/D/B**	See Table 181 on page 691.
	X'0186'	**TR31 Export - Permit DATA to D0:B**	
"E0", "E1", "E2", "E3", "E4", and "E5": TR-31 EMC/chip issuer master-key keys			
EMVACMK	X'0199'	**TR31 Export - Permit DKYGENKY:DKYL0+DMAC to E0**	See Table 182 on page 692.
	X'019A'	**TR31 Export - Permit DKYGENKY:DKYL0+DMV to E0**	
	X'019B'	**TR31 Export - Permit DKYGENKY:DKYL0+DALL to E0**	
	X'019C'	**TR31 Export - Permit DKYGENKY:DKYL1+DMAC to E0**	
	X'019D'	**TR31 Export - Permit DKYGENKY:DKYL1+DMV to E0**	
	X'019E'	**TR31 Export - Permit DKYGENKY:DKYL1+DALL to E0**	

TR-31 key usage keyword	Offset	Command	Specific key type and control vector attributes
EMVSCMK	X'019F'	**TR31 Export - Permit DKYGENKY:DKYL0+DDATA to E1**	See Table 182 on page 692.
	X'01A0'	**TR31 Export - Permit DKYGENKY:DKYL0+DMPIN to E1**	
	X'01A1'	**TR31 Export - Permit DKYGENKY:DKYL0+DALL to E1**	
	X'01A2'	**TR31 Export - Permit DKYGENKY:DKYL1+DDATA to E1**	
	X'01A3'	**TR31 Export - Permit DKYGENKY:DKYL1+DMPIN to E1**	
	X'01A4'	**TR31 Export - Permit DKYGENKY:DKYL1+DALL to E1**	
EMVSIMK	X'01A5'	**TR31 Export - Permit DKYGENKY:DKYL0+DMAC to E2**	See Table 182 on page 692.
	X'01A6'	**TR31 Export - Permit DKYGENKY:DKYL0+DALL to E2**	
	X'01A7'	**TR31 Export - Permit DKYGENKY:DKYL1+DMAC to E2**	
	X'01A8'	**TR31 Export - Permit DKYGENKY:DKYL1+DALL to E2**	
EMVDAMK	X'01A9'	**TR31 Export - Permit DATA/MAC/CIPHER/ENCIPHER to E3**	See Table 182 on page 692.
EMVDNMK	X'01AA'	**TR31 Export - Permit DKYGENKY:DKYL0+DDATA to E4**	See Table 182 on page 692.
	X'01AB'	**TR31 Export - Permit DKYGENKY:DKYL0+DALL to E4**	
EMVCPMK	X'01AC'	**TR31 Export - Permit DKYGENKY:DKYL0+DEXP to E5**	See Table 182 on page 692.
	X'01AD'	**TR31 Export - Permit DKYGENKY:DKYL0+DMAC to E5**	
	X'01AE'	**TR31 Export - Permit DKYGENKY:DKYL0+DDATA to E5**	
	X'01AF'	**TR31 Export - Permit DKYGENKY:DKYL0+DALL to E5**	
"K0" and "K1": TR-31 key encryption or wrapping, or key block protection keys			
KEK	X'0187'	**TR31 Export - Permit EXPORTER/OKEYXLAT to K0:E**	See Table 183 on page 693.
	X'0188'	**TR31 Export - Permit IMPORTER/IKEYXLAT to K0:D**	
KEK-WRAP	X'0189'	**TR31 Export - Permit EXPORTER/OKEYXLAT to K1:E**	
	X'018A'	**TR31 Export - Permit IMPORTER/IKEYXLAT to K1:D**	
"M0", "M1", and "M3": TR-31 ISO MAC algorithm keys			
ISOMAC0	X'018B'	**TR31 Export - Permit MAC/DATA/DATAM to M0:G/C**	See Table 185 on page 696.
	X'018C'	**TR31 Export - Permit MACVER/DATAMV to M0:V**	
ISOMAC1	X'018D'	**TR31 Export - Permit MAC/DATA/DATAM to M1:G/C**	
	X'018E'	**TR31 Export - Permit MACVER/DATAMV to M1:V**	
ISOMAC3	X'018F'	**TR31 Export - Permit MAC/DATA/DATAM to M3:G/C**	
	X'0190'	**TR31 Export - Permit MACVER/DATAMV to M3:V**	
"P0", "V0", "V1": TR-31 PIN encryption or PIN verification keys			

TR-31 key usage keyword	Offset	Command	Specific key type and control vector attributes
PINENC	X'0191'	**TR31 Export - Permit OPINENC to P0:E**	See Table 186 on page 699.
	X'0192'	**TR31 Export - Permit IPINENC to P0:D**	
PINVO	X'0193'	**TR31 Export - Permit PINVER:NO-SPEC to V0**	
	X'0194'	**TR31 Export - Permit PINGEN:NO-SPEC to V0**	
	X'01B0'	**TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N**	
PINV3624	X'0195'	**TR31 Export - Permit PINVER:NO-SPEC/IBM-PIN/IBM-PINO to V1**	
	X'0196'	**TR31 Export - Permit PINGEN:NO-SPEC/IBM-PIN/IBM-PINO to V1**	
	X'01B0'	**TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N**	
VISAPVV	X'0197'	**TR31 Export - Permit PINVER:NO-SPEC/VISA-PVV to V2**	
	X'0198'	**TR31 Export - Permit PINGEN:NO-SPEC/VISA-PVV to V2**	
	X'01B0'	**TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N**	

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBT31XJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void  CSNBT31XJ(
              hikmNativeInteger return_code,
              hikmNativeInteger reason_code,
              hikmNativeInteger exit_data_length,
              byte[]            exit_data,
              hikmNativeInteger rule_array_count,
              byte[]            rule_array,
              byte[]            key_version_number,
              hikmNativeInteger key_field_length,
              hikmNativeInteger source_key_identifier_length,
              byte[]            source_key_identifier,
              hikmNativeInteger unwrap_kek_identifier_length,
              byte[]            unwrap_kek_identifier,
              hikmNativeInteger wrap_kek_identifier_length,
              byte[]            wrap_kek_identifier,
              hikmNativeInteger opt_blocks_length,
              byte[]            opt_blocks,
              hikmNativeInteger tr31_key_block_length,
              byte[]            tr31_key_block);
```

TR31 Key Import (CSNBT31I)

Use the TR31 Key Import verb to convert a non-proprietary external key-block that is formatted under the rules of TR-31 into a proprietary CCA external or internal fixed-length DES key-token with its attributes in a control vector.

After being imported into a CCA key-token, the key and its attributes are ready to be used in a CCA system. The verb takes as input an external TR-31 key block and the internal DES IMPORTER or IKEYXLAT key-encrypting key of the key that was used to wrap the TR-31 key block.

TR31 Key Import (CSNBT31I)

The TR31 Key Import verb is analogous to the existing Key Import verb, except that TR31 Key Import accepts an external non-CCA DES key-token instead of an external CCA fixed-length DES key-token, and it translates the key to either an external or internal fixed-length DES key-token instead of only an internal fixed-length DES key-token. An import by TR31 Key Import to an external key-token requires a suitable internal fixed-length DES key-encrypting key. The purpose of both verbs is to import a DES key from another party.

An external-to-external translation would not normally be called an *export* or *import operation*. Instead, it would be called a *key translation* and would be handled by a verb such as Key Translate or Key Translate2. For practical reasons, the export of an external CCA DES key-token to an external TR-31 format is supported by the Key Export to TR31 verb, and the import of an external TR-31 key block to an external CCA DES key-token format is supported by the TR31 Key Import verb.

Note that the TR31 Key Import verb does not support the translation of an external key from encipherment under one key-encrypting key to encipherment under a different key-encrypting key. When converting an external TR-31 key block to an external fixed-length DES key-token, the key-encrypting key used to wrap the external TR-31 key block must be the same as the one used to wrap the external fixed-length DES key-token. Use the Key Translate or Key Translate2 verbs for switching external key wrapping keys: the normal function of those verbs.

Both CCA and TR-31 define key attributes that control key usage. In both cases, the usage information is securely bound to the key so that the attributes cannot be changed in unintended or uncontrolled ways. CCA maintains its DES key attributes in a control vector (CV), while a TR-31 key block uses fields: *key usage*, *algorithm*, *mode of use*, and *exportability*.

Each attribute in a CCA control vector falls under one of these categories:

1. There is a one-to-one correspondence between the CV attribute and the TR-31 attribute. For these attributes, conversion is straightforward.
2. There is not a one-to-one correspondence between the CV attribute and the TR-31 attribute, but the attribute can be automatically translated when performing this export operation.
3. There is not a one-to-one correspondence between the CV attribute and the TR-31 attribute, in which case a rule-array keyword has been defined to specify which attribute is to be used in the TR-31 key block.
4. Category (1), (2), or (3) applies, but there are some attributes that are lost completely on translation (for example, key-generating bits in key-encrypting keys).
5. None of the above categories applies, because the key type, its attributes, or both simply cannot be reasonably translated into a TR-31 key block.

The control vector is always checked for compatibility with the TR-31 attributes. It is an error if the specified control vector attributes are in any way incompatible with the attributes of the input key. In addition, access control points are defined that can be used to restrict the permitted attribute conversions.

The import operation produces the CCA external or internal fixed-length DES key-token as its output. It does not return any field values or optional block data from the TR-31 key block header. To obtain the header field values, use the TR31 Key Token Parse verb. To obtain optional block data from the header, use the TR31 Optional Data Read verb.

An optional control vector transport control rule-array keyword can be passed to the Key Export to TR31 verb. Such a keyword specifies that the verb is to copy the control vector from the CCA DES key into the TR-31 key block. A copy of the control vector is passed in an IBM-proprietary optional block. See "TR-31 optional block data" on page 876.

If the TR-31 key block contains an IBM-proprietary block, the TR31 Key Import verb verifies that the control vector is compatible with the attributes in the TR-31 key block. If any incompatibility is found, the verb rejects the import. If the control vector is valid for the key, the verb uses it for the control vector of the CCA DES key-token. Note that the import operation is always subject to the restrictions imposed by the relevant access control points, even if a control vector is received.

A control vector, if present, can be in the TR-31 key block in one of two ways, depending on the control vector transport control keyword specified in the rule array of the Key Export to TR31 verb when the key was exported. One keyword option is **ATTR-CV**, and the other is **INCL-CV**:

ATTR-CV
: Causes a copy of the control vector to be included in the TR-31 key block. The TR-31 key usage and mode of use fields are set to IBM proprietary. See "TR-31 optional block data" on page 876. These proprietary values indicate that the usage and mode information is contained in the included control vector. In this case, if the TR31 Key Import verb successfully verifies that the included control vector does not conflict with the rule-array keywords specified, it uses it as the control vector for the imported CCA DES key-token.

INCL-CV
: Causes a copy of the control vector to be included in the TR-31 key block. The TR-31 key usage and mode of use fields contain attributes from the set defined in the TR-31 standard. In this case, the TR31 Key Import verb verifies that the usage and mode information in those fields are compatible with the included control vector. The verb also verifies that no rule array keywords conflict with the control vector.

  Note that the included CV could have more capability from a CCA perspective than the TR-31 usage and mode fields indicate. This difference is not an error, because the key block binding methods give the importer assurance that the key block optional blocks are as secure as any other attribute.

Special notes

Additional information about CSNBT31I.

1. Several import situations might require keywords. Keywords are ignored for **INCL-CV** scenarios unless they directly conflict with the included CV. For example, the verb returns an error if the control vector indicates that a DKYGENKY key has a subtype of DKYL0 and the user specifies the DKYL1 keyword.
2. Be aware of the interaction of ACP X'0158' (TR31 Export - Permit Any CCA Key if INCL-CV Is Specified) with the **INCL-CV** keyword for the Key Export to TR31 verb. Without the **INCL-CV** keyword specified, most export translations are guarded by key-type specific ACPs. These ACPs are used to guard the source CCA system against attacks involving reimport of the exported key under ambiguous circumstances. When the control vector is exported in an optional block along with the key, it is securely bound to the encrypted key.

This somewhat protects the source system because the key on import is allowed to have only the form of the included control vector. Expansion of capability is blocked. If ACP X'0158' is not enabled in the active role, the type-specific ACPs are still required. However, if ACP X'0158' is enabled, the key-type specific ACPs are not checked when **INCL-CV** is specified.

3. Be aware of ACP X'017C' (TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER) for import of PINGEN or PINVER keys to wrapping mode A, usage V0, V1, V2, and mode N.

The extra translation-specific ACPs are intended to enable control of situations where the CCA imported key type is ambiguous based on the TR-31 key attributes. This ambiguity is never the case when **INCL-CV** has been specified with the Key Export to TR31 verb, which ensures that the imported TR-31 key block has a valid CV to precisely control the resultant CCA key. Therefore, there are no translation-specific ACPs governing **INCL-CV** import translations.

Examples

Examples for CSNBT31I.

1. A full MAC key that is exported as TR-31 "C0" key block with an included control vector will be re-imported as a full MAC key.
2. A DKYGENKY key with key usage DALL key exported as a TR-31 "E0", "E1", or "E2" key block with an included control vector will be re-imported as a DKYGENKY key with key usage DALL, even though the "E0", "E1", and "E2 types are more restricted.

Format

The format of CSNBT31I.

```
CSNBT31I(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             tr31_key_block_length,
             tr31_key_block,
             unwrap_kek_identifier_length,
             unwrap_kek_identifier,
             wrap_kek_identifier_length,
             wrap_kek_identifier,
             output_key_identifier_length,
             output_key_identifier,
             num_opt_blocks,
             cv_source,
             protection_method)
```

Parameters

The parameter definitions for CSNBT31I.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 1, 2, 3, or 4.

rule_array

Direction: Input
Type: String array

A pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* keywords are shown in the following table:

Keyword	Meaning
Token identifier (one required)	
INTERNAL	Specifies to return the output key in an internal CCA key-token.
EXTERNAL	Specifies to return the output key in an external CCA key-token, wrapped by the transport key identified by the wrap_kek_identifier parameter.
CCA output key usage ***subgroups*** (One from one subgroup required based on TR-31 input key usage. Keywords for the subgroup are valid **only** for given TR-31 key usage.) **Note:** None of the following keywords are allowed if the TR-31 key block provided as input has an optional block that contains a CCA control vector. See Table 243 on page 877. If the TR-31 key block header contains an optional block with a control vector in it, the control vector is used in place of keywords to produce the output CCA key-token. If the key usage and mode of use fields of the key block are not IBM-defined (see Table 243 on page 877), the control vector must not conflict with any TR-31 header fields.	
CV subtype extension for "C0" key usage (one required). Only valid for TR-31 key block with key usage "C0" and no control vector in optional block.	
CVK-CVV	Convert a TR-31 card verification key (CVK) to a double-length CCA DES MAC key that has a subtype extension of CVVKEY-A. This restricts the key to generating or verifying a Visa CVV or MasterCard CVC.
CVK-CSC	Convert a TR-31 CVK to a CCA DES MAC key that has a subtype extension of AMEX-CSC. This restricts the key to generating or verifying an American Express card security code, also known as a card identification number (CID).
CV key type for "K0" key usage (one required). Only valid for TR-31 key block with key usage "K0" and no control vector in optional block.	
EXPORTER	For TR-31 key usage "K0" and mode of use "E" or "B", convert a TR-31 key encryption or wrapping key to a CCA EXPORTER key.
OKEYXLAT	For TR-31 key usage "K0" and mode of use "E" or "B", convert a TR-31 key encryption or wrapping key to a CCA OKEYXLAT key.
IMPORTER	For TR-31 key usage "K0" and mode of use "D" or "B", convert a TR-31 key encryption or wrapping key to a CCA IMPORTER key.
IKEYXLAT	For TR-31 key usage "K0" and mode of use "D" or "B", convert a TR-31 key encryption or wrapping key to a CCA IKEYXLAT key.
CV key type for "V0", "V1", or "V2" key usage (one required). Only valid for TR-31 key block with key usage "V0", "V1", or "V2" and no control vector in optional block. When this keyword is specified, an optional CV key type modifier can be specified for key usage "V0" or "V1".	
PINGEN	Convert a TR-31 PIN verification key to a CCA PINGEN key.
PINVER	Convert a TR-31 PIN verification key to a CCA PINVER key.
CV key usage for "E0" or "E2" key usage (one required) Only valid for TR-31 key block with key usage "E0" or "E2" and no control vector in optional block.	

Keyword	Meaning
DMAC	Convert TR-31 EMV/chip issuer master key: application cryptograms or secure messaging for integrity to CCA DKYGENKY with key usage DMAC.
DMV	Convert TR-31 EMV/chip issuer master key: application cryptograms or secure messaging for integrity to CCA DKYGENKY with key usage DMV.
CV key usage for "E1" key usage (one required) Only valid for TR-31 key block with key usage "E1" and no control vector in optional block.	
DMPIN	Convert TR-31 EMV/chip issuer master key: secure messaging for confidentiality to CCA DKYGENKY with key usage DMPIN
DDATA	Convert TR-31 EMV/chip issuer master key: secure messaging for confidentiality to CCA DKYGENKY with key usage DDATA.
CV key usage for "E5" key usage (one required) Only valid for TR-31 key block with key usage "E5" and no control vector in optional block.	
DMAC	Convert TR-31 EMV/chip issuer master key: card personalization to CCA DKYGENKY with key usage DMAC.
DMV	Convert TR-31 EMV/chip issuer master key: card personalization to CCA DKYGENKY with key usage DMV.
DEXP	Convert TR-31 EMV/chip issuer master key: card personalization to CCA DKYGENKY with key usage DEXP.
CV subtype for "E0", "E1", or "E2" key usage (one required). Only valid for TR-31 key block with key usage "E0", "E1", or "E2" and no control vector in optional block.	
DKYL0	Convert TR-31 EMV/chip issuer master key: application cryptograms, secure message for confidentiality, or secure message for integrity to CCA DKYGENKY with subtype DKYL0.
DKYL1	Convert TR-31 EMV/chip issuer master key: application cryptograms, secure message for confidentiality, or secure message for integrity to CCA DKYGENKY with subtype DKYL1.
CV key type modifier for "V0" or "V1" key usage (one required). Only valid for TR-31 key block with key usage "V0" or "V1" and no control vector in optional block.	
NOOFFSET	Convert a TR-31 PIN verification key to a CCA PINGEN or PINVER key with the key type modifier NOOFFSET, so that the key cannot participate in a PIN offset process or PVV process.
Key-wrapping method (one, optional)	
USECONFG	This is the default. Specifies to wrap the key using the configuration setting for the default wrapping method. The default wrapping method configuration setting may be changed using the TKE. This keyword is ignored for AES keys. **Note:** Do not use this keyword if the default wrapping method is WRAP-ECB and a control vector is present in an optional block of the TR-31 key block with CV bit 56 = B'1' (ENH-ONLY). Use the WRAP-ENH keyword instead.
WRAP-ECB	Specifies to wrap the key using the legacy wrapping method. **Note:** Do not use this keyword if a control vector is present in an optional block of the TR-31 key block with CV bit 56 = B'1' (ENH-ONLY).
WRAP-ENH	Specifies to wrap the key using the enhanced wrapping method.
Translation control (optional). This keyword is valid only with key-wrapping method **WRAP-ENH** or with **USECONFG** when the default wrapping method is **WRAP-ENH**. This option cannot be used on a key with a control vector valued to binary zeros.	
ENH-ONLY	Specifies to restrict the key from being wrapped with the legacy method once it has been wrapped with the enhanced method. Sets CV bit 56 = B'1' (ENH-ONLY). **Note:** If a control vector is present in an optional block of the TR-31 key block with CV bit 56 = B'0', this keyword overrides that value in the CCA key-token. This keyword has no effect if the control vector in an optional block is all zeros.

Table 187 on page 713 shows all valid translations for import of a TR-31 BDK base derivation key (usage "B0") to a CCA KEYGENKY key, along with any

access control commands that must be enabled in the active role for that key type and control vector attributes. These keys are for translating derived unique key per transaction (DUKPT) base derivation keys.

Table 187. Import translation table for a TR-31 BDK base derivation key (usage "B0")

Key usage	Key block protection method keyword (version ID)	Mode of use	Rule-array keywords	CCA key type and control vector attributes	Offset	Command
"B0"	"A"	"N"	N/A	KEYGENKY, double length, UKPT (CV bit 18 = B'1')	N/A	N/A
"B0"	"B" or "C"	"X"	N/A	KEYGENKY, double length, UKPT (CV bit 18 = B'1')	N/A	N/A

Note:
1. These keys are the base keys from which derived unique key per transaction (DUKPT) initial keys are derived for individual devices such as PIN pads.
2. This table defines the only supported translations for this TR-31 usage. Usage must be the following value:
   "B0" BDK base derivation key.
3. There are no specific access-control commands for this translation because it is not ambiguous or in need of interpretation.

Table 188 shows all valid translations for import of a TR-31 CVK card verification key (usage "C0") to a CCA MAC or DATA key, along with any access control commands that must be enabled in the active role for that key type and control vector attributes. These keys are for computing or verifying (against supplied value) a card verification code with the CVV, CVC, CVC2, and CVV2 algorithm.

Table 188. Import translation table for a TR-31 CVK card verification key (usage "C0")

Key usage	Key block protection method keyword (version ID)	Mode of use	Rule-array keywords	CCA key type and control vector attributes	Offset	Command
"C0"	"A", "B", or "C"	"G" or "C"	**CVK-CSC**	MAC, single or double length, AMEX-CSC (CV bits 0 - 3 = B'0100')	X'015B'	**TR31 Import - Permit C0 to MAC/MACVER:AMEXCSC**
			CVK-CVV	MAC, double length, CVVKEY-A (CV bits 0 - 3 = B'0010')	X'015A'	**TR31 Import - Permit C0 to MAC/MACVER:CVVKEY-A**
		"V"	**CVK-CSC**	MACVER, single or double length, AMEX-CSC (CV bits 0 - 3 = B'0100')	X'015B'	**TR31 Import - Permit C0 to MAC/MACVER:AMEXCSC**
			CVK-CVV	MACVER, double length, CVVKEY-A (CV bits 0 - 3 = B'0010')	X'015A'	**TR31 Import - Permit C0 to MAC/MACVER:CVVKEY-A**

Table 188. Import translation table for a TR-31 CVK card verification key (usage "C0") (continued)

Security considerations:

1. There is asymmetry in the translation from a CCA DATA key to a TR-31 key. The asymmetry results from CCA DATA keys having attributes of both data encryption keys and MAC keys, while TR-31 separates data encryption keys from MAC keys. A CCA DATA key can be exported to a TR-31 "C0" key, if one or both applicable MAC generate and MAC verify control vector bits are on. However, a TR-31 "C0" key cannot be imported to the lower-security CCA DATA key, it can be imported only to a CCA key type of MAC or MACVER. This restriction eliminates the ability to export a CCA MAC or MACVER key to a TR-31 key and re-importing it back as a CCA DATA key with the capability to Encipher, Decipher, or both.
2. The translation from TR-31 usage "C0" is controlled by rule array keywords when using the TR31_Key_Import verb. This makes it possible to convert an exported CCA CVVKEY-A key into an AMEX-CSC key or the other way around. To prevent such a conversion, do not enable offsets X'015A' (TR31 Import - Permit C0 to MAC/MACVER:CVVKEY-A) and X'015B' (TR31 Import - Permit C0 to MAC/MACVER:AMEXCSC) at the same time. However, if both CVVKEY-A and AMEX-CSC translation types are required, then offsets X'015A' and X'015B' must be enabled. In this case, control is up to the development, deployment, and execution of the applications themselves.

Note:

1. Card verification keys are used for computing or verifying (against supplied value) a card verification code with the CVV, CVC, CVC2, and CVV2 algorithms. In CCA, these keys correspond to keys used with two algorithms:
   - Visa CVV and MasterCard CVC codes are generated and verified using the CVV Generate and CVV Verify verbs. These verbs require a key type of DATA or MAC/MACVER with a subtype extension (CV bits 0 - 3) of ANY-MAC, single-length CVVKEY-A and single-length CVVKEY-B, and a double-length CVVKEY-A (see CVV Key Combine verb). The MAC generate and the MAC verify (CV bits 20 - 21) key usage values must be set appropriately.
   - American Express CSC codes are generated and verified using the Transaction Validation verb. This verb requires a key type of MAC or MACVER with a subtype extension of ANY-MAC or AMEX-CSC.
2. The translation from TR-31 usage "C0" to a CCA MAC/MACVER key with a subtype extension of ANY-MAC (CV bits 0 - 3 = B'0000') is not allowed.
3. This table defines the only supported translations for this TR-31 usage. Usage must be the following value:
   "C0" CVK card verification key
4. CCA does not have an equivalent to the TR-31 "generate only" mode of use, so a translation from TR-31 mode "G" will result in a CCA MAC key with both MAC generate and MAC verify attributes (CV bits 20 - 21 = B'11'). Note that any key that can perform a generate operation can readily verify a MAC as well.
5. The CCA representation and the TR-31 representation of CVV keys are incompatible. CCA represents the CVVKEY-A and CVVKEY-B keys as two 8-byte (single length) keys, while TR-31 represents these keys as one 16-byte key. The CVV Generate and CVV Verify verbs have support added to accept one 16-byte CVV key, using left and right key parts as A and B. Current Visa standards require this.
6. Import and export of 8-byte CVVKEY-A and CVVKEY-B MAC/MACVER keys is allowed only using the proprietary TR-31 usage+mode values ("10" and "1", respectively) to indicate encapsulation of the IBM control vector in an optional block, because the 8-byte CVVKEY-A is meaningless and useless as a TR-31 "C0" usage key of any mode.

Table 189 shows all valid translations for import of a TR-31 data encryption key (usage "D0") to a CCA ENCIPHER, DECIPHER, CIPHER, or DATA key, along with any access control commands that must be enabled in the active role for that key type and control vector attributes. These keys are used for the encryption and/or decryption of data.

Table 189. Import translation table for a TR-31 data encryption key (usage "D0")

Key usage	Key block protection method keyword (version ID)	Mode of use	Rule-array keywords	CCA key type and control vector attributes	Offset	Command
"D0"	"A", "B", or "C"	"E"	N/A	ENCIPHER, single or double length	N/A	N/A
		"D"	N/A	DECIPHER, single or double length		
		"B"	N/A	CIPHER, single or double length		

Table 189. Import translation table for a TR-31 data encryption key (usage "D0") (continued)

Key usage	Key block protection method keyword (version ID)	Mode of use	Rule-array keywords	CCA key type and control vector attributes	Offset	Command
Security consideration: There is asymmetry in the translation from a CCA DATA key to a TR-31 key. The asymmetry results from CCA DATA keys having attributes of both data encryption keys and MAC keys, while TR-31 separates data encryption keys from MAC keys. A CCA DATA key can be exported to a TR-31 "D0" key, if one or both applicable Encipher or Decipher control vector bits are on. However, a TR-31 "D0" key **cannot be imported** to the lower-security CCA DATA key, it can be imported only to a CCA key type of ENCIPHER, DECIPHER, or CIPHER. This restriction eliminates the ability to export a CCA DATA key to a TR-31 key, and re-importing it back as a CCA DATA key with the capability to MAC generate and MAC verify. **Note:** 1. Data encryption keys are used for the encryption and decryption of data. 2. This table defines the only supported translations for this TR-31 usage. Usage must be the following value: **"D0"** Data encryption 3. There are no specific access-control commands for this translation since it is not ambiguous or in need of interpretation.						

Table 190 shows all valid translations for import of a TR-31 key encryption or wrapping, or key block protection key (usages "K0", "K1") to a CCA EXPORT, OKEYXLAT, IMPORTER, or IKEYXLAT key, along with any access control commands that must be enabled in the active role for that key type and control vector attributes. These keys are used only to encrypt or decrypt other keys, or as a key used to derive keys that are used for that purpose.

Table 190. Import translation table for a TR-31 key encryption or wrapping, or key block protection key (usages "K0", "K1")

Key usage	Key block protection method keyword (version ID)	Mode of use	Rule-array keywords	CCA key type and control vector attributes	Offset	Command
"K0"	"A", "B", or "C"	"E"	**OKEYXLAT**	OKEYXLAT, double length	X'015C'	**TR31 Import - Permit K0:E to EXPORTER/OKEYXLAT**
			EXPORTER	EXPORTER, double length, EXPORT on (CV bit 21 = B'1')		
		"D"	**IKEYXLAT**	IKEYXLAT, double length	X'015D'	**TR31 Import - Permit K0:D to IMPORTER/IKEYXLAT**
			IMPORTER	IMPORTER, double length, IMPORT on (CV bit 21 = B'1')		
		"B"	**OKEYXLAT**	OKEYXLAT, double length	X'015E'	**TR31 Import - Permit K0:B to EXPORTER/OKEYXLAT**
			EXPORTER	EXPORTER, double length, EXPORT on (CV bit 21 = B'1')		
			IKEYXLAT	IKEYXLAT, double length	X'015F'	**TR31 Import - Permit K0:B to IMPORTER/IKEYXLAT**
			IMPORTER	IMPORTER, double length, IMPORT on (CV bit 21 = B'1')		

Table 190. Import translation table for a TR-31 key encryption or wrapping, or key block protection key (usages "K0", "K1") (continued)

"K1"	"B" or "C"	"E"	**OKEYXLAT**	OKEYXLAT, double length	X'0160'	**TR31 Import - Permit K1:E to EXPORTER/OKEYXLAT**
			EXPORTER	EXPORTER, double length, EXPORT on (CV bit 21 = B'1')		
		"D"	**IKEYXLAT**	IKEYXLAT, double length	X'0161'	**TR31 Import - Permit K1:D to IMPORTER/IKEYXLAT**
			IMPORTER	IMPORTER, double length, IMPORT on (CV bit 21 = B'1')		
		"B"	**OKEYXLAT**	OKEYXLAT, double length	X'0162'	**TR31 Import - Permit K1:B to EXPORTER/OKEYXLAT**
			EXPORTER	EXPORTER, double length, EXPORT on (CV bit 21 = B'1')		
			IKEYXLAT	IKEYXLAT, double length	X'0163'	**TR31 Import - Permit K1:B to IMPORTER/IKEYXLAT**
			IMPORTER	IMPORTER, double length, IMPORT on (CV bit 21 = B'1')		

Security considerations:

1. The CCA OKEYXLAT, EXPORTER, IKEYXLAT, or IMPORTER KEK translation to a TR-31 "K0" key with mode "B" (both wrap and unwrap) is not allowed for security reasons. Even with access-control point control, this capability would give an immediate path to turn a CCA EXPORTER key into a CCA IMPORTER, and the other way around.
2. When a TR-31 key block does not have an included control vector as an optional block, the default control vector is used to construct the output key-token. Default CCA EXPORTER or IMPORTER keys have CV bits 18 - 20 on, which are used for key generation.

Note:

1. Key encryption or wrapping keys are used only to encrypt or decrypt other keys, or as a key used to derive keys that are used for that purpose.
2. This table defines the only supported translations for this TR-31 usage. Usage must be the following value:
   "K0" Key encryption or wrapping
   "K1" TR-31 key block protection key
3. Any attempt to import a TR-31 "K0" or "K1" key that has algorithm "D" (DEA) will result in an error because CCA does not support single-length KEKs.
4. CCA mode support is the same for version IDs "A", "B", and "C", because the distinction between TR-31 "K0" and "K1" does not exist in CCA keys. CCA does not distinguish between targeted protocols currently, and so there is no good way to represent the difference. Also note that most wrapping mechanisms now involve derivation or key variation steps.

Table 191 shows all valid translations for import of a TR-31 ISO MAC algorithm key (usages "M0", "M1", "M3") to a CCA MAC, MACVER, DATA, DATAM, or DATAMV key, along with any access control commands that must be enabled in the active role for that key type and control vector attributes. These keys are use to compute or verify a code for message authentication.

Table 191. Import translation table for a TR-31 ISO MAC algorithm key (usages "M0", "M1", "M3")

Key usage	Key block protection method keyword (version ID)	Mode of use	Rule-array keywords	CCA key type and control vector attributes	Offset	Command

Table 191. Import translation table for a TR-31 ISO MAC algorithm key (usages "M0", "M1", "M3") (continued)

"M0"	"A", "B", or "C"	"G" or "C"	N/A	MAC, double length, ANY-MAC (CV bits 0 - 3 = B'0000')	X'0164'	**TR31 Import - Permit M0/M1/M3 to MAC/MACVER:ANY-MAC**
		"V"		MACVER, double length, ANY-MAC (CV bits 0 - 3 = B'0000')		
"M1"		"G" or "C"		MAC, single or double length, ANY-MAC (CV bits 0 - 3 = B'0000')		
		"V"		MACVER, single or double length, ANY-MAC (CV bits 0 - 3 = B'0000')		
"M3"		"G" or "C"		MAC, single or double length, ANY-MAC (CV bits 0 - 3 = B'0000')		
		"V"		MACVER, single or double length, ANY-MAC (CV bits 0 - 3 = B'0000')		

Security consideration: There is asymmetry in the translation from a CCA DATA key to a TR-31 key. The asymmetry results from CCA DATA keys having attributes of both data encryption keys and MAC keys, while TR-31 separates data encryption keys from MAC keys. A CCA DATA key can be exported to a TR-31 "M0", "M1", or "M3" key, if one or both applicable MAC generate and MAC verify control vector bits are on. However, a TR-31 "M0", "M1", or "M3" key cannot be imported to the lower-security CCA DATA key, it can **be imported only** to a CCA key type of MAC or MACVER. This restriction eliminates the ability to export a CCA MAC or MACVER key to a TR-31 key, and re-importing it back as a CCA DATA key with the capability to Encipher, Decipher, or both.

Note:

1. MAC keys are used to compute or verify a code for message authentication.
2. This table defines the only supported translations for this TR-31 usage. Usage must be one of the following values:

   "M0" SO 16609 MAC algorithm 1, TDEA

   The ISO 16609 MAC algorithm 1 is based on ISO 9797. It is identical to "M1" except that it does not support 8-byte DES keys.

   "M1" SO 9797 MAC algorithm 1

   The ISO 9797 MAC algorithm 1 is identical to "M0" except that it also supports 8-byte DES keys.

   "M3" ISO 9797 MAC algorithm 3

   The X9.19 style of Triple-DES MAC.
3. A CCA control vector has no bits defined to limit key usage by algorithm, such as CBC MAC (TR-31 usage "M0" and "M1") or X9.19 (TR-31 usage "M3"). When importing a TR-31 key block, the resulting CCA key token deviates from the restrictions of usages "M0", "M1", and "M3". Importing a TR-31 key block which allows MAC generation ("G" or "C") results in a control vector with the ANY-MAC attribute rather than for the restricted algorithm that is set in the TR-31 key block. The ANY-MAC attribute provides the same restrictions as what CCA currently uses for generating and verifying MACs.

Table 192 on page 718 shows all valid translations for import of a TR-31 PIN encryption or PIN verification key (usages "P0", "V0", "V1", "V2") to a CCA OPINENC, IPINENC, PINGEN, or PINVER key, along with any access control commands that must be enabled in the active role for that key type and control vector attributes. These keys are used to protect PIN blocks and to generate or verify a PIN using a particular PIN-calculation method for that key type.

Table 192. Import translation table for a TR-31 PIN encryption or PIN verification key (usages "P0", "V0", "V1", "V2")

Key usage	Key block protection method keyword (version ID)	Mode of use	Rule-array keywords	CCA key type and control vector attributes		Offset	Command
"P0"	"A", "B", or "C"	"E"	N/A	OPINENC, double length		X'0165'	**TR31 Import - Permit P0:E to OPINENC**
		"D"		IPINENC, double length		X'0166'	**TR31 Import - Permit P0:D to IPINENC**
"V0"	"A"	"N" (requires both commands)	**PINGEN**	PINGEN, double length, NO-SPEC (CV bits 0 - 3 = B'0000')	NOOFFSET off (CV bit 37 = B'0')	X'0167'	**TR31 Import - Permit V0 to PINGEN:NO-SPEC**
						X'017C'	**TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER**
	"A", "B", or "C"	"G" or "C"			NOOFFSET off (CV bit 37 = B'0')	X'0167'	**TR31 Import - Permit V0 to PINGEN:NO-SPEC**
	"A"	"N" (requires both commands)	**PINGEN**, **NOOFFSET**		NOOFFSET on (CV bit 37 = B'1')	X'0167'	**TR31 Import - Permit V0 to PINGEN:NO-SPEC**
						X'017C'	**TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER**
	"A", "B", or "C"	"G" or "C"			NOOFFSET on (CV bit 37 = B'1')	X'0167'	**TR31 Import - Permit V0 to PINGEN:NO-SPEC**
	"A"	"N" (requires both commands)	**PINVER**	PINVER, double length, NO-SPEC (CV bits 0 - 3 = B'0000')	NOOFFSET off (CV bit 37 = B'0')	X'0168'	**TR31 Import - Permit V0 to PINVER:NO-SPEC**
						X'017C'	**TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER**
	"A", "B", or "C"	"V"			NOOFFSET off (CV bit 37 = B'0')	X'0168'	**TR31 Import - Permit V0 to PINVER:NO-SPEC**
	"A"	"N" (requires both commands)	**PINVER**, **NOOFFSET**		NOOFFSET on (CV bit 37 = B'1')	X'0168'	**TR31 Import - Permit V0 to PINVER:NO-SPEC**
						X'017C'	**TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER**
	"A", "B", or "C"	"V"			NOOFFSET on (CV bit 37 = B'1')	X'0168'	**TR31 Import - Permit V0 to PINVER:NO-SPEC**

Table 192. Import translation table for a TR-31 PIN encryption or PIN verification key (usages "P0", "V0", "V1", "V2") (continued)

"V1"	"A"	"N" (requires both commands)	**PINGEN**	PINGEN, double length, IBM PIN/IBM-PINO (CV bits 0 - 3 = B'0001')	NOOFFSET off (CV bit 37 = B'0')	X'0169'	**TR31 Import - Permit V1 to PINGEN:IBM-PIN/ IBMPINO**
						X'017C'	**TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER**
	"A", "B", or "C"	"G" or "C"			NOOFFSET off (CV bit 37 = B'0')	X'0169'	**TR31 Import - Permit V1 to PINGEN:IBM-PIN/ IBMPINO**
	"A"	"N" (requires both commands)	**PINGEN, NOOFFSET**		NOOFFSET on (CV bit 37 = B'1')	X'0169'	**TR31 Import - Permit V1 to PINGEN:IBM-PIN/ IBMPINO**
						X'017C'	**TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER**
	"A", "B", or "C"	"G" or "C"			NOOFFSET on (CV bit 37 = B'1')	X'0169'	**TR31 Import - Permit V1 to PINGEN:IBM-PIN/ IBMPINO**
	"A"	"N" (requires both commands)	**PINVER**	PINVER, double length, IBM PIN/IBM-PINO (CV bits 0 - 3 = B'0001')	NOOFFSET off (CV bit 37 = B'0')	X'016A'	**TR31 Import - Permit V1 to PINVER:IBM-PIN/ IBMPINO**
						X'017C'	**TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER**
	"A", "B", or "C"	"V"			NOOFFSET off (CV bit 37 = B'0')	X'016A'	**TR31 Import - Permit V1 to PINVER:IBM-PIN/ IBMPINOEC**
	"A"	"N" (requires both commands)	**PINVER, NOOFFSET**		NOOFFSET on (CV bit 37 = B'1')	X'016A'	**TR31 Import - Permit V1 to PINVER:IBM-PIN/ IBMPINOC**
						X'017C'	**TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER**
	"A", "B", or "C"	"V"			NOOFFSET on (CV bit 37 = B'1')	X'016A'	**TR31 Import - Permit V1 to PINVER:IBM-PIN/ IBMPINO**

Table 192. Import translation table for a TR-31 PIN encryption or PIN verification key (usages "P0", "V0", "V1", "V2") (continued)

"V2"	"A"	"N" (requires both commands)	**PINGEN**	PINGEN, double length, VISA-PVV (CV bits 0 - 3 = B'0010')	X'016B'	**TR31 Import - Permit V2 to PINGEN:VISA-PVV**
					X'017C'	**TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER**
	"A", "B", or "C"	"G" or "C"		PINGEN, double length, VISA-PVV (CV bits 0 - 3 = B'0010')	X'016B'	**TR31 Import - Permit V2 to PINGEN:VISA-PVV**
	"A"	"N" (requires both commands)	**PINVER**	PINVER, double length, VISA-PVV (CV bits 0 - 3 = B'0010')	X'016C'	**TR31 Import - Permit V2 to PINVER:VISA-PVV**
					X'017C'	**TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER**
	"A", "B", or "C"	"V"		PINVER, double length, VISA-PVV (CV bits 0 - 3 = B'0010')	X'016C'	**TR31 Import - Permit V2 to PINVER:VISA-PVV**

Security note: TR-31 key blocks that are protected under legacy version ID "A" (using the Key Variant Binding Method 2005 Edition) use the same mode of use "N" for PINGEN and PINVER keys. For version ID "A" keys only, for a given PIN key usage, enabling both the PINGEN and PINVER access-control points at the same time while enabling offset X'017C' (for mode "N", no special restrictions) is **NOT** recommended. In other words, for a particular PIN verification key usage, you should not simultaneously enable the four commands shown below for that usage:

Table 193. Commands

Key type, mode, or version	Offset	Command
"V0": For usage V0, a user with the following four commands enabled in the active role can change a PINVER key into a PINGEN key and the other way around. **Avoid simultaneously enabling these four commands.**		
Key type PINGEN	X'0167'	**TR31 Import - Permit V0 to PINGEN:NO-SPEC**
Key type PINVER	X'0168'	**TR31 Import - Permit V0 to PINVER:NO-SPEC**
Mode "N"	X'017C'	**TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER**
Version "A"	X'0150'	**TR31 Import - Permit Version A TR-31 Key Blocks**
"V1": For usage V1, a user with the following four commands enabled in the active role can change a PINVER key into a PINGEN key and the other way around. **Avoid simultaneously enabling these four commands.**		
Key type PINGEN	X'0169'	**TR31 Import - Permit V1 to PINGEN:IBM-PIN/IBMPINO**
Key type PINVER	X'016A'	**TR31 Import - Permit V1 to PINVER:IBM-PIN/IBMPINO**
Mode "N"	X'017C'	**TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER**
Version "A"	X'0150'	**TR31 Import - Permit Version A TR-31 Key Blocks**
"V2": For usage V2, a user with the following four commands enabled in the active role can change a PINVER key into a PINGEN key and the other way around. **Avoid simultaneously enabling these four commands.**		
Key type PINGEN	X'016B'	**TR31 Import - Permit V2 to PINGEN:VISA-PVV**

Table 193. Commands (continued)

Key type, mode, or version	Offset	Command
Key type PINVER	X'016C'	**TR31 Import - Permit V2 to PINVER:VISA-PVV**
Mode "N"	X'017C'	**TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER**
Version "A"	X'0150'	**TR31 Import - Permit Version A TR-31 Key Blocks**

Failure to comply with this recommendation allows changing PINVER keys into PINGEN and the other way around.

Note:

1. PIN encryption keys are used to protect PIN blocks. PIN verification keys are used to generate or verify a PIN using a particular PIN-calculation method for that key type.
2. This table defines the only supported translations for this TR-31 usage. Usage must be one of the following values:
   "P0" PIN encryption
   "V0" PIN verification, KPV, other algorithm

   Usage "V0" does not have its own PIN-calculation method defined. The mapping to NO-SPEC is sub-optimal. Exporting to "N" mode restricts keys from being imported with the IBM-PIN/IBM-PINO or VISA-PVV attribute, while CCA NO-SPEC allows any method.
   "V1" PIN verification, IBM 3624
   "V2" PIN verification, Visa PVV

   The **NOOFFSET** keyword is not allowed for the Visa PVV algorithm because it does not support this attribute.
3. Mode must be one of the following values:
   "E" Encrypt/wrap only

   This mode restricts PIN encryption keys to encrypting a PIN block. May be used to create or reencipher an encrypted PIN block (for key-to-key translation).
   "D" Decrypt/unwrap only

   This mode restricts PIN encryption keys to decrypting a PIN block. Generally used in a PIN translation to decrypt the incoming PIN block.
   "N" No special restrictions (other than restrictions implied by the key usage)

   This mode is used by several vendors for a PIN generate or PIN verification key when the key block version ID is "A".
   "G" Generate only

   This mode is used for a PINGEN key that may not perform a PIN verification. The control vector will not have its EPINVER attribute on (CV bit 22 = B'0').
   "V" Verify only

   This mode is used for PIN verification only. If the TR-31 key block does not have a control vector included, the only usage bits set on in the control vector is the EPINVER bit (CV bits 18 - 22 = B'00001').
   "C" Both generate and verify (combined)

This mode indicates that the control vector will have the default PINGEN bits on (CV bits 18 -22 = B'11111').

4. Any attempt to import a TR-31 "P0" key that has mode "B" (both encrypt and decrypt) will result in an error because CCA does not support this combination of attributes.
5. If the TR-31 key block contains a control vector, and the control vector has NOOFFSET on, the **NOOFFSET** keyword is not necessary because the verb will automatically set NOOFFSET on in this case.

Table 194 shows all valid translations for import of a TR-31 EMV/chip issuer master-key key (usages "E0", "E1", "E2", "E3", "E4", "E5") to a CCA DKYGENKY, DATA, MAC, CIPHER, or ENCIPHER key, along with any access control commands that must be enabled in the active role for that key type and control vector attributes. These keys are used by the chip cards to perform cryptographic operations or, in some cases, to derive keys used to perform operations.

Table 194. Import translation table for a TR-31 EMV/chip issuer master-key key (usages "E0", "E1", "E2", "E3", "E4", "E5")

Key usage	Key block protection method keyword (version ID)	Mode of use	Rule-array keywords	CCA key type and control vector attributes	Offset	Command
"E0"	"A"	"N"	**DKYL0**, **DMAC**	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMAC (CV bits 19 - 22 = B'0010')	X'016D'	**TR31 Import - Permit E0 to DKYGENKY:DKYL0+DMAC**
	"B" or "C"	"X"				
	"A"	"N"	**DKYL0**, **DMV**	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMV (CV bits 19 - 22 = B'0011')	X'016E'	**TR31 Import - Permit E0 to DKYGENKY:DKYL0+DMV**
	"B" or "C"	"X"				
	"A"	"N"	**DKYL1**, **DMAC**	DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DMAC (CV bits 19 - 22 = B'0010')	X'016F'	**TR31 Import - Permit E0 to DKYGENKY:DKYL1+DMAC**
	"B" or "C"	"X"				
	"A"	"N"	**DKYL1**, **DMV**	DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DMV (CV bits 19 - 22 = B'0011')	X'0170'	**TR31 Import - Permit E0 to DKYGENKY:DKYL1+DMV**
	"B" or "C"	"X"				

Table 194. Import translation table for a TR-31 EMV/chip issuer master-key key (usages "E0", "E1", "E2", "E3", "E4", "E5") (continued)

"E1"	"A"	"N"	**DKYL0**, **DMPIN**	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMPIN (CV bits 19 - 22 = B'1001')	X'0171'	**TR31 Import - Permit E1 to DKYGENKY:DKYL0+DMPIN**
		"E"				
		"D"				
		"B"				
	"B" or "C"	"X"				
	"A"	"N"	**DKYL0**, **DDATA**	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DDATA (CV bits 19 - 22 = B'0001')	X'0172'	**TR31 Import - Permit E1 to DKYGENKY:DKYL0+DDATA**
		"E"				
		"D"				
		"B"				
	"B" or "C"	"X"				
	"A"	"N"	**DKYL1**, **DMPIN**	DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DMPIN (CV bits 19 - 22 = B'1001')	X'0173'	**TR31 Import - Permit E1 to DKYGENKY:DKYL1+DMPIN**
		"E"				
		"D"				
		"B"				
	"B" or "C"	"X"				
	"A"	"N"	**DKYL1**, **DDATA**	DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DDATA (CV bits 19 - 22 = B'0001')	X'0174'	**TR31 Import - Permit E1 to DKYGENKY:DKYL1+DDATA**
		"E"				
		"D"				
		"B"				
	"B" or "C"	"X"				
"E2"	"A"	"N"	**DKYL0**, **DMAC**	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMAC (CV bits 19 - 22 = B'0010')	X'0175'	**TR31 Import - Permit E2 to DKYGENKY:DKYL0+DMAC**
	"B" or "C"	"X"				
	"A"	"N"	**DKYL1**, **DMAC**	DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DMAC (CV bits 19 - 22 = B'0010')	X'0176'	**TR31 Import - Permit E2 to DKYGENKY:DKYL1+DMAC**
	"B" or "C"	"X"				
"E3"	"A"	"N"	N/A	ENCIPHER	X'0177'	**TR31 Import - Permit E3 to ENCIPHER**
		"E"				
		"D"				
		"B"				
		"G"				
	"B" or "C"	"X"				
"E4"	"A"	"N"	N/A	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DDATA (CV bits 19 - 22 = B'0001')	X'0178'	**TR31 Import - Permit E4 to DKYGENKY:DKYL0+DDATA**
		"B"				
	"B" or "C"	"X"				

Table 194. Import translation table for a TR-31 EMV/chip issuer master-key key (usages "E0", "E1", "E2", "E3", "E4", "E5") (continued)

"E5"	"A"	"G"	**DKYL0**, **DMAC**	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMAC (CV bits 19 - 22 = B'0010')	X'0179'	**TR31 Import - Permit E5 to DKYGENKY:DKYL0+DMAC**
		"C"				
		"V"				
		"E"				
		"D"				
		"B"				
		"N"				
	"B" or "C"	"X"				
	"A"	"G"	**DKYL0**, **DDATA**	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DDATA (CV bits 19 - 22 = B'0001')	X'017A'	**TR31 Import - Permit E5 to DKYGENKY:DKYL0+DDATA**
		"C"				
		"V"				
		"E"				
		"D"				
		"B"				
		"N"				
	"B" or "C"	"X"				
	"A"	"G"	**DKYL0**, **DEXP**	DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DEXP (CV bits 19 - 22 = B'0101')	X'017B'	**TR31 Import - Permit E5 to DKYGENKY:DKYL0+DEXP**
		"C"				
		"V"				
		"E"				
		"D"				
		"B"				
		"N"				
	"B" or "C"	"X"				

Note:

1. EMV/chip issuer master-key keys are used by the chip cards to perform cryptographic operations or, in some cases, to derive keys used to perform operations. In CCA, these keys are (a) diversified key-generating keys (key type DKYGENKY), allowing derivation of operational keys, or (b) operational keys. Note that in this context, "master key" has a different meaning than for CCA. These master keys, also called KMCs, are described by EMV as DES master keys for personalization session keys. They are used to derive the corresponding chip card master keys, and not typically used directly for cryptographic operations other than key derivation. In CCA, these keys are usually key generating keys with derivation level DKYL1 (CV bits 12 - 14 = B'001'), used to derive other key generating keys (the chip card master keys). For some cases, or for older EMV key derivation methods, the issuer master keys could be level DKYL0 (CV bits 12 - 14 = B'000').
2. This table defines the only supported translations for this TR-31 usage. Usage must be one of the following values:
   "E0" Application cryptograms
   "E1" Secure messaging for confidentiality
   "E2" Secure messaging for integrity
   "E3" Data authentication code
   "E4" Dynamic numbers
   "E5" Card personalization
3. EMV support in CCA is different than TR-31 support, and CCA key types do not match TR-31 types.
4. DKYGENKY keys are double length only.
5. In CCA, a MAC key that can perform a MAC generate operation also can perform a MAC verify. For TR-31 mode "G" (generate only), the translation to a CCA key results in a key that can perform MAC generate and MAC verify.

tr31_key_block_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *tr31_key_block* variable. The length field in the TR-31 block is a 4-digit decimal number, so the maximum acceptable length is 9992 bytes. For more information, see "TR31 Key Token Parse (CSNBT31P)" on page 730.

tr31_key_block

Direction: Input
Type: String

A pointer to a string variable containing the TR-31 key block that is to be imported. The key block is protected with the key identified by the *unwrap_kek_identifier* parameter.

unwrap_kek_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *unwrap_kek_identifier* variable. Set this value to 64.

unwrap_kek_identifier

Direction: Input
Type: String

A pointer to a string variable containing the operational fixed-length DES key-token used to unwrap the key identified by the *tr31_key_block* parameter, or a key label of such a key in DES key-storage. The key must have a key type of IMPORTER or IKEYXLAT, and be authorized for import.

Note: DES keys wrapped in ECB mode (CCA legacy wrap mode) cannot be used to wrap or unwrap TR-31 "B" or "C" key blocks that have or will have "E" exportability, because ECB mode does not comply with ANSI X9.24 Part 1.

wrap_kek_identifier_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the length in bytes of the *wrap_kek_identifier* variable. The value must be greater than or equal to 0. A null key-token can have a length of 1. Set this value to 64 for a key label or a KEK.

wrap_kek_identifier

Direction: Input
Type: String

A pointer to a string variable containing the operational fixed-length DES key-token used to wrap the key identified by the *output_key_identifier* parameter, a null key-token, or a key label of such a key in DES key-storage. If the key token identified by the parameter is not null, it must have a key type of IMPORTER or IKEYXLAT, be authorized for import, and have the same

clear key as the key identified by the *unwrap_kek_identifier* parameter. If the parameter identifies a null key-token, then the *unwrap_kek_identifier* parameter is also used for wrapping the CCA output key token.

output_key_identifier_length

Direction: Input/Output
Type: Integer

A pointer to an integer specifying the length in bytes of the *output_key_identifier* variable. This is an input/output parameter.

output_key_identifier

Direction: Input/Output
Type: String

A pointer to a string variable containing the key token or the key label for the token that is to receive the imported key. The output key-token is a CCA internal or external key token containing the key received in the TR-31 token. If a key token is provided, it must be a null token (64 bytes of X'00'). If a key label is provided, the imported token is stored in the key storage file and identified by that label.

num_opt_blocks

Direction: Output
Type: Integer

A pointer to an integer variable where the verb stores the number of optional blocks that are present in the TR-31 key token.

cv_source

Direction: Output
Type: Integer

A pointer to an integer variable where the verb stores a value indicating how the control vector in the output key token was created. It can be one of the values in Table 195.

Table 195. TR31 Key Import CV sources

CSNBT31I CV source	Meaning
0	No CV was present in an optional block, and the output CV was created by the verb based on input parameters and on the attributes in the TR-31 key block header.
1	A CV was obtained from an optional block in the TR-31 key block, and the key usage and mode of use were also specified in the TR-31 header. The verb verified compatibility of the header values with the CV and then used that CV in the output key token.
2	A CV was obtained from an optional block in the TR-31 key block, and the key usage and mode of use in the TR-31 header held the proprietary values indicating that key use and mode should be obtained from the included CV. The CV from the TR-31 token was used as the CV for the output key token.

Any values other than these three are reserved and are currently invalid.

protection_method

Direction: Output

Type: Integer

A pointer to an integer variable where the verb stores a value indicating what method was used to protect the input TR-31 key block. The TR-31 standard allows two methods, and the application program might want to know which was used for security purposes. The variable can have one of the values in Table 196.

Table 196. TR31 Key Import protection methods

CSNBT31I protection method	Meaning
0	The TR-31 key block was protected using the variant method as identified by a Key Block Version ID value of "A" (X'41').
1	The TR-31 key block was protected using the derived key method as identified by a Key Block Version ID value of "B" (X'42').
2	The TR-31 key block was protected using the variant method as identified by a Key Block Version ID value of "C" (X'43'). Functionally this method is the same as "A", but to maintain consistency a different value is returned here for "C".

Any values other than these three are reserved and are currently invalid.

Restrictions

The restrictions for CSNBT31I.

None.

Required commands

The required commands for CSNBT31I.

The TR31 Key Import verb requires the following commands to be enabled in the active role:

Rule-array keyword	Offset	Command
WRAP-ENH or **WRAP-ECB**	X'0153'	**TR31 Import - Permit Override of default wrapping method**

TR-31 key block version ID	Offset	Command
"A" (X'41')	X'0150'	**TR-31 Import - Permit version A TR-31 key blocks**
"B" (X'42')	X'0151'	**TR-31 Import - Permit version B TR-31 key blocks**
"C" (X'43')	X'0152'	**TR-31 Import - Permit version C TR-31 key blocks**

In order to access key storage, this verb also requires the **Key Test and Key Test2** command (offset X'001D') to be enabled in the active role.

Be aware of access-control point X'017C' (TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER) for import of PINGEN or PINVER keys to wrapping method

"A", usage "V0", "V1", or "V2", and mode "N". TR-31 key blocks with legacy key usage "A" (key block protected using the Key Variant Binding Method 2005 Edition) use the same mode "N" for PINGEN as well as PINVER keys. For usage "A" keys only, enabling a PINGEN and PINVER access-control point while enabling offset X'017C' (for mode "N") is NOT recommended. Failure to comply with this recommendation allows changing PINVER keys into PINGEN, and the other way around.

In addition to the required commands, the verb needs these additional commands to be enabled in the active role depending on the rule-array keyword provided:

Rule-array keyword	Offset	Command	Specific key usage, version ID, and mode values
"C0": TR-31 CVK card verification keys			
CVK-CSC	X'015B'	**TR31 Import - Permit C0 to MAC/MACVER:AMEXCSC**	See Table 188 on page 713.
CVK-CVV	X'015A'	**TR31 Import - Permit C0 to MAC/MACVER:CVKEY-A**	
"K0" and "K1": TR-31 key encryption or wrapping, or key block protection keys			
OKEYXLAT or **EXPORTER**	X'015C'	**TR31 Import - Permit K0:E to EXPORTER/ OKEYXLAT**	See Table 190 on page 715.
	X'015E'	**TR31 Import - Permit K0:B to EXPORTER/OKEYXLAT**	
	X'0160'	**TR31 Import - Permit K1:E to EXPORTER/OKEYXLAT**	
	X'0162'	**TR31 Import - Permit K1:B to EXPORTER/OKEYXLAT**	
IKEYXLAT or **IMPORTER**	X'015D'	**TR31 Import - Permit K0:D to IMPORTER/IKEYXLAT**	
	X'015F'	**TR31 Import - Permit K0:B to IMPORTER/IKEYXLAT**	
	X'0161'	**TR31 Import - Permit K1:D to IMPORTER/IKEYXLAT**	
	X'0163'	**TR31 Import - Permit K1:B to IMPORTER/IKEYXLAT**	
"M0", "M1", and "M3": TR-31 ISO MAC algorithm keys			
N/A	X'0164'	**TR31 Import - Permit M0/M1/M3 to MAC/MACVER:ANY-MAC**	See Table 191 on page 716.
	X'018C'	**TR31 Export - Permit MACVER/DATAMV to M0:V**	
"P0", "V0", "V1": TR-31 PIN encryption or PIN verification keys			
N/A	X'0165'	**TR31 Import - Permit P0:E to OPINENC**	See Table 192 on page 718.
	X'0166'	**TR31 Import - Permit P0:D to IPINENC**	
PINGEN	X'0167'	**TR31 Import - Permit V0 to PINGEN:NO-SPEC**	
	X'0169'	**TR31 Import - Permit V1 to PINGEN:IBM-PIN/IBMPINO**	
	X'016B'	**TR31 Import - Permit V2 to PINGEN:VISA-PVV**	
	X'017C'	**TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER**	
PINVER	X'0168'	**TR31 Import - Permit V0 to PINVER:NO-SPEC**	
	X'016A'	**TR31 Import - Permit V1 to PINVER:IBM-PIN/IBMPINO**	
	X'016C'	**TR31 Import - Permit V2 to PINVER:VISA-PVV**	
	X'017C'	**TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER**	
"E0", "E1", "E2", "E3", "E4", and "E5": TR-31 EMC/chip issuer master-key keys			

Rule-array keyword	Offset	Command	Specific key usage, version ID, and mode values
DKYL0	X'016D'	**TR31 Import - Permit E0 to DKYGENKY:DKYL0+DMAC**	See Table 194 on page 722.
	X'016E'	**TR31 Import - Permit E0 to DKYGENKY:DKYL0+DMV**	
	X'0171'	**TR31 Import - Permit E1 to DKYGENKY:DKYL0+DMPIN**	
	X'0172'	**TR31 Import - Permit E1 to DKYGENKY:DKYL0+DDATA**	
	X'0175'	**TR31 Import - Permit E2 to DKYGENKY:DKYL0+DMAC**	
	X'0179'	**TR31 Export - Permit DKYGENKY:DKYL1+DALL to E0**	
	X'017A'	**TR31 Import - Permit E5 to DKYGENKY:DKYL0+DDATA**	
	X'017B'	**TR31 Import - Permit E5 to DKYGENKY:DKYL0+DEXP**	
DKYL1	X'016F'	**TR31 Import - Permit E0 to DKYGENKY:DKYL1+DMAC**	See Table 194 on page 722.
	X'0170'	**TR31 Import - Permit E0 to DKYGENKY:DKYL1+DMV**	
	X'0173'	**TR31 Import - Permit E1 to DKYGENKY:DKYL1+DMPIN**	
	X'0174'	**TR31 Import - Permit E1 to DKYGENKY:DKYL1+DDATA**	
	X'0176'	**TR31 Import - Permit E2 to DKYGENKY:DKYL1+DMAC**	
DMAC	X'016D'	**TR31 Import - Permit E0 to DKYGENKY:DKYL0+DMAC**	See Table 194 on page 722.
	X'016F'	**TR31 Import - Permit E0 to DKYGENKY:DKYL1+DMAC**	
	X'0175'	**TR31 Import - Permit E2 to DKYGENKY:DKYL0+DMAC**	
	X'0176'	**TR31 Import - Permit E2 to DKYGENKY:DKYL1+DMAC**	
	X'0179'	**TR31 Import - Permit E5 to DKYGENKY:DKYL0+DMAC**	
DMV	X'016E'	**TR31 Import - Permit E0 to DKYGENKY:DKYL0+DMV**	See Table 194 on page 722.
	X'0170'	**TR31 Import - Permit E0 to DKYGENKY:DKYL1+DMV**	
DMPIN	X'0171'	**TR31 Import - Permit E1 to DKYGENKY:DKYL0+DMPIN**	
	X'0173'	**TR31 Import - Permit E1 to DKYGENKY:DKYL1+DMPIN**	
DDATA	X'0172'	**TR31 Import - Permit E1 to DKYGENKY:DKYL0+DDATA**	See Table 194 on page 722.
	X'0174'	**TR31 Import - Permit E1 to DKYGENKY:DKYL1+DDATA**	
	X'017A'	**TR31 Import - Permit E5 to DKYGENKY:DKYL0+DDATA**	
DEXP	X'017B'	**TR31 Import - Permit E5 to DKYGENKY:DKYL0+DEXP**	See Table 194 on page 722.
N/A	X'0177'	**TR31 Import - Permit E3 to ENCIPHER**	
	X'0178'	**TR31 Import - Permit E4 to DKYGENKY:DKYL0+DDATA**	

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBT31IJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void  CSNBT31IJ(
             hikmNativeInteger return_code,
             hikmNativeInteger reason_code,
             hikmNativeInteger exit_data_length,
             byte[]            exit_data,
             hikmNativeInteger rule_array_count,
             byte[]            rule_array,
             hikmNativeInteger tr31_key_block_length,
```

```
byte[]            tr31_key_block,
hikmNativeInteger unwrap_kek_identifier_length,
byte[]            unwrap_kek_identifier,
hikmNativeInteger wrap_kek_identifier_length,
byte[]            wrap_kek_identifier,
hikmNativeInteger output_key_identifier_length,
byte[]            output_key_identifier,
hikmNativeInteger num_opt_blocks,
hikmNativeInteger cv_source,
hikmNativeInteger protection_method);
```

TR31 Key Token Parse (CSNBT31P)

Use the TR31 Key Token Parse verb to disassemble the unencrypted header of an external TR-31 key block into separate pieces of information.

The part of the header that is optional, called optional blocks, is not disassembled. To obtain the contents of optional blocks, use the TR31 Optional Data Read verb. Neither verb performs any cryptographic services, and both disassemble a key block in application storage. The validity of the key block is verified as much as can be done without performing any cryptographic services.

The TR-31 header fields that are disassembled into separate pieces of information include a key block version ID, key block length, key usage, algorithm, mode of use, key version number, exportability, and number of optional blocks. Except for the two length values, which are returned as integers, the verb returns the field values as ASCII strings. This format is used in the TR-31 key block itself. For more information, see X9 TR-31 2010: *Interoperable Secure Key Exchange Block Specification for Symmetric Algorithms*.

The following table summarizes the key blocks fields returned by this verb:

TR-31 field name	Verb parameter	Field or buffer string length in bytes	Description of TR-31 field
Key block version ID	*key_block_version*	1	Identifies the method by which the key block is cryptographically protected and the content layout of the block.
Key block length	*key_block_length*	4 (integer)	Entire key block length after encoding (header, encrypted confidential data, and MAC).
Key usage	*key_usage*	2	Provides information about the intended function of the protected key/sensitive data, such as data encryption, PIN encryption, or key wrapping. Numeric values are reserved for proprietary use (that is, not defined by TR-31).
Algorithm	*algorithm*	1	The approved symmetric algorithm for which the protected key may be used. Numeric values are reserved for proprietary use.
Mode of use	*mode*	1	Defines the operation for which the protected key can perform. Numeric values are reserved for proprietary use.
Key version number	*key_version_number*	2	Version number to optionally indicate that the contents of the key block is a component (key part), or to prevent re-injection of an old key. This field is a tool for enforcement of local key change rules.

TR-31 field name	Verb parameter	Field or buffer string length in bytes	Description of TR-31 field
Exportability	*exportability*	1	Defines whether the protected key may be exported.
Number of optional blocks	*num_opt_blocks*	4 (integer)	Defines the number of optional blocks included in the key block. If this value is greater than zero, use the TR31 Optional Data Read verb to obtain the contents of the optional blocks.

This verb does not perform cryptographic services on any key value. You cannot use this verb to change a key or to change the control vector related to a key.

Format

The format of CSNBT31P.

```
CSNBT31P(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              rule_array_count,
              rule_array,
              tr31_key_length,
              tr31_key,
              key_block_version,
              key_block_length,
              key_usage,
              algorithm,
              mode,
              key_version_number,
              exportability,
              num_opt_blocks)
```

Parameters

The parameters for CSNBT31P.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. The value must be 0.

rule_array

Direction: Input
Type: String array

No rule array keywords are defined for this verb.

tr31_key_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *tr31_key* variable. Specify a length that is greater than or equal to the size of the key block. The verb determines the actual length of the key by parsing its contents.

tr31_key

Direction: Input
Type: String

A pointer to a string variable containing the TR-31 key block to be disassembled.

key_block_version

Direction: Output
Type: String

A pointer to a string variable. The verb copies the one byte found in the key block version ID field of the input key block to this variable.

Note that if the verb finds a proprietary key block version ID, the verb treats it as an invalid value, because the verb is not capable of disassembling a key block that has a proprietary ID. This variable is not updated if a processing error occurs.

key_block_length

Direction: Output
Type: Integer

A pointer to an integer variable. The verb parses the 2-byte numeric ASCII key block length field from the input key block, converts the string value into an integer, and returns the integer in this variable. This value must be less than or equal to the *tr31_key_length* input variable.

key_usage

Direction: Output
Type: String

A pointer to a string variable. The verb copies the two bytes found in the key usage field of the input key block to this variable.

algorithm

Direction: Output
Type: String

A pointer to a string variable. The verb copies the one byte found in the algorithm field of the input key block to this variable. **The verb does not treat a proprietary algorithm value as an error.**

mode

Direction: Output
Type: String

A pointer to a one-byte string variable containing the TR-31 mode of use for the key contained in the block. The value is obtained from the TR-31 header. The mode of use describes what operations the key can perform, within the

limitations specified with the key usage value. For example, a key with usage for data encryption can have a mode to indicate that it can be used only for encryption, decryption, or both.

This pointer must be non-NULL and point to application storage with at least the size given by the byte count noted. The storage is updated with the noted value on a successful return from this verb, and unchanged otherwise.

key_version_number

Direction: Output
Type: String

A pointer to a 2-byte string variable obtained from the TR-31 header, which can be used for one of three purposes, or can be unused.

- If both bytes are X'30' ("0"), then key versioning is unused for this key. In this case, the second byte is not examined and can contain any value.
- If the first byte is X'63' ("c"), then the block contains a component of a key which must be combined with other components in order to form the complete key. TR-31 does not define the method through which the components are combined. TR-31 specifies that local rules are used for that purpose.

  In this case, the second byte is not examined and can contain any value.
- If the first byte is anything other than the two values above, then the 2-byte key version value is an identifier of the version of the key that is carried in the block. This key version value can be used by an application, for example, to ensure that an old version of a key is not reentered into the system.

This pointer must be non-NULL and point to application storage with at least the size given by the byte count noted. The storage is updated with the noted value on a successful return from this verb, and unchanged otherwise.

exportability

Direction: Output
Type: String

A pointer to a one-byte string variable containing the key exportability value from the TR-31 header. This value indicates whether the key can be exported from this system, and if so specifies conditions under which export is permitted. The following three values are possible:

- If the value is X'4E' ("N"), then the key is not exportable.
- If the value is X'53' ("S"), then the key is exportable under any key-encrypting key.
- If the value is X'45' ("E"), then the key is exportable only under a trusted key-encrypting key. TR 31 defines such a trusted key as either one that is encrypted under the HSM master key or one that is itself contained in a TR-31 key block. CCA does not support KEKs that are wrapped in TR-31 key blocks.

This pointer must be non-NULL and point to application storage with at least the size given by the byte count noted. The storage is updated with the noted value on a successful return from this verb, and unchanged otherwise.

num_opt_block

Direction: Output
Type: Integer

A pointer to an integer value containing the number of optional blocks that are part of the TR-31 key block. Information about each optional block can be obtained using the TR31 Optional Data Read verb. In this verb, use the number of optional blocks acquired with this verb to obtain a list of the IDs and lengths for each optional block. Then, use those lists to read the data from each desired block.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBT31PJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBT31PJ(
              hikmNativeInteger return_code,
              hikmNativeInteger reason_code,
              hikmNativeInteger exit_data_length,
              byte[]            exit_data,
              hikmNativeInteger rule_array_count,
              byte[]            rule_array,
              hikmNativeInteger tr31_key_length,
              byte[]            tr31_key,
              byte[]            key_block_version,
              hikmNativeInteger key_block_length,
              byte[]            key_usage,
              byte[]            algorithm,
              byte[]            mode,
              byte[]            key_version_number,
              byte[]            exportability,
              hikmNativeInteger num_opt_blocks);
```

TR31 Optional Data Build (CSNBT31O)

Use the TR31 Optional Data Build verb to build a properly formatted TR-31 optional block from the data provided.

The newly constructed optional block can optionally be appended to an existing structure of optional blocks, or it can be returned as a new optional blocks structure. After the last optional block has been constructed, the completed structure containing the optional blocks can be included in a TR-31 key block during an export operation by the Key Export to TR31 verb by using its *opt_blocks* parameter. For information about TR-31, including the format of a TR-31 optional block, see X9 TR-31 2010: *Interoperable Secure Key Exchange Block Specification for Symmetric Algorithms*.

The TR-31 key block has an unencrypted header that can contain optional blocks. The header is securely bound to the key block using the integrated MAC. An optional block has a 2-byte ASCII block ID value that determines the use of the block. The ID of each block in an optional blocks structure must be unique.

The verb builds a structure of optional blocks by adding one optional block with each call. This process is repeated until the entire set of optional blocks has been added. For each call, provide the components for a single optional block. This includes the optional block ID, the optional block length in bytes, and the optional block data. In addition, provide an optional blocks buffer large enough to add the optional block being built.

There are two valid scenarios for the optional blocks buffer provided on input, as determined by the value of the *opt_blocks_length* variable:

1. The optional blocks buffer is empty. In this case, the newly constructed optional block is copied into the buffer.
2. The optional blocks buffer contains one or more existing optional blocks. In this case, the newly constructed optional block is appended to the existing optional blocks. No duplicate IDs are allowed.

Upon successful completion, the *opt_blocks_length* variable is updated to the length of the returned optional blocks structure.

This verb does not perform cryptographic services on any key value. You cannot use this verb to change a key or to change the control vector related to a key.

Format

The format of CSNBT31O.

```
CSNBT31O(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              rule_array_count,
              rule_array,
              opt_blocks_bfr_length,
              opt_blocks_length,
              opt_blocks,
              num_opt_blocks,
              opt_block_id,
              opt_block_data_length,
              opt_block_data)
```

Parameters

The parameters for CSNBT31O.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 0, because no keywords are currently defined for this verb.

rule_array

Direction: Input
Type: String array

No rule array keywords are defined for this verb.

opt_blocks_bfr_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the length in bytes of the buffer allocated for the *opt_blocks* variable. Set this length to at least the size of any optional blocks structure in the buffer plus the optional block being added.

opt_blocks_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the length in bytes of the data in the *opt_blocks* variable. This length must be less than or equal to the value of the *opt_blocks_bfr_length* variable. On input, set this variable to the length of the optional blocks structure being updated. Set this value to zero if it is the first optional block in the structure. On successful completion, this variable is updated with the length of the updated variable.

opt_blocks

Direction: Input/Output
Type: String

A pointer to a string variable containing a buffer for the optional blocks structure that the verb updates. In the first call to the verb, the buffer will generally be empty. The verb appends one optional block to the buffer with each call.

num_opt_blocks

Direction: Output
Type: Integer

The *num_opt_blocks* parameter is a pointer to an integer variable containing the number of optional blocks contained in the *opt_blocks* variable that is returned by the verb.

opt_block_id

Direction: Input
Type: String

A pointer to a string variable containing a 2-byte value that identifies the use of the optional block. Each ID must be unique, that is, no duplicates are allowed.

Note that a value of "PB" is not allowed. The Key Export to TR31 verb adds a padding block of the appropriate size as needed. Unlike the padding in the encryption key portion of the TR-31 key block, the padding block for optional blocks serves no security purpose.

opt_block_data_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the length in bytes of the data passed in the *opt_block_data* variable. Note that it is valid for this length to be zero, since an optional block can have an ID and a length, but no data.

opt_block_data

Direction: Input
Type: String

A pointer to a string variable containing the data for the optional block that is to be constructed.

Restrictions

The restrictions for CSNBT31O.

An optional block with an ID of "PB" (padding block) cannot be added by the user. The Key Export to TR31 verb adds a padding block of the appropriate size as needed when building the TR-31 key block. Unlike the padding within the encrypted key portion of the key block, the padding block for optional blocks serves no security purpose.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBT31OJ.

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBT31OJ(
              hikmNativeInteger return_code,
              hikmNativeInteger reason_code,
              hikmNativeInteger exit_data_length,
              byte[]            exit_data,
              hikmNativeInteger rule_array_count,
              byte[]            rule_array,
              hikmNativeInteger opt_blocks_bfr_length,
              hikmNativeInteger opt_blocks_length,
              byte[]            opt_blocks,
              hikmNativeInteger num_opt_blocks,
              byte[]            opt_block_id,
              hikmNativeInteger opt_block_data_length,
              byte[]            opt_block_data);
```

TR31 Optional Data Read (CSNBT31R)

Use the TR31 Optional Data Read verb to either obtain information about all of the optional blocks in the header of an external TR-31 key block, or obtain the length and data of the specified optional block.

To disassemble the part of the header that is not optional, use the TR31 Key Token Parse verb. Neither verb performs any cryptographic services, and both disassemble a key block in application storage. The validity of the key block is verified as much as can be done without performing any cryptographic services.

A TR-31 key block contains an unencrypted header that can include one or more optional blocks. All parts of the header are securely bound to the key block using the integrated MAC.

Optional blocks in a key block must each be identified by a unique 2-byte ID. The value of an ID must either be defined by TR-31 or be a numeric value, otherwise the key block is invalid. Numeric IDs are reserved for proprietary use. For more information, see X9 TR-31 2010: *Interoperable Secure Key Exchange Block Specification for Symmetric Algorithms*.

In order to obtain the data of a particular optional block from the header of an external TR-31 key block, perform the following steps:

1. Use the *tr31_key* parameter to identify the TR-31 key block that this verb is to process.
2. Call the TR31 Key Token Parse verb to parse the TR-31 key block. See "TR31 Key Token Parse (CSNBT31P)" on page 730. Upon successful completion:
   - Set the value of the *tr31_key_length* variable to the value returned in the *tr31_key_length* variable.
   - Set the value of the *num_opt_blocks* variable to the value returned in the *num_opt_blocks* variable.
   - Allocate a string buffer in bytes for the *opt_blocks_id* and an integer buffer in bytes for the *opt_blocks_length* variables. These buffers must be at least two times the value of the *num_opt_blocks* variable.
3. Specify a rule-array keyword of **INFO** to obtain information about the optional blocks in the key block. The *opt_block_id*, *opt_block_data_length*, and *opt_block_data* parameters are ignored.
4. Call the TR31 Optional Data Read verb to read data from the TR-31 key block. Upon successful completion, the verb returns an array of optional block IDs in the *opt_blocks_id* variable, and an array of lengths for the optional block IDs in the *opt_blocks_length* variable. The IDs and lengths are returned in same order as the optional blocks appear in the header of the TR-31 key block.
5. Determine which ID of the unique IDs contained in the *opt_blocks_id* variable is to be obtained from the TR-31 key block. Set the *opt_block_id* variable to this 2-byte value. Set the value of the *opt_block_data_length* variable to the corresponding length from the *opt_blocks_length* variable.

   Note: The offset used to locate the ID in the *opt_blocks_id* variable has the same value as the offset for the corresponding length in the *opt_blocks_length* variable.
6. Allocate a buffer in bytes for the *opt_block_data* variable that is at least the value of the *opt_block_data_length* variable.
7. Specify a rule-array keyword of **DATA** to obtain the length and data of the specified optional block. The *num_opt_blocks* and the *opt_blocks_id* parameters are ignored.
8. Call the TR31 Optional Data Read verb. Upon successful completion, the verb returns the data of the specified optional block in the *opt_block_data* variable. The verb updates the *opt_block_data_length* variable to the number of bytes returned in the *opt_block_data* variable.

This verb does not perform cryptographic services on any key value. You cannot use this verb to change a key or to change the control vector related to a key.

Format

The format of CSNBT31R.

```
CSNBT31R(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              rule_array_count,
              rule_array,
              tr31_key_length,
              tr31_key,
              opt_block_id,
              num_opt_blocks,
              opt_block_ids,
              opt_block_lengths,
              opt_block_data_length,
              opt_block_data)
```

Parameters

The parameters for CSNBT31R.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 1.

rule_array

Direction: Input
Type: String array

A pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The following *rule_array* keywords are defined for this verb:

Keyword	Meaning
Operation (one required)	
INFO	Return information about the optional blocks in the TR-31 key block.
DATA	Return the data contained in a specified optional block in the TR-31 key block.

tr31_key_length

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the *tr31_key* variable. Specify a length that is greater than or equal to the size of the key block. The verb determines the actual length of the key by parsing its contents.

tr31_key

Direction: Input
Type: String

A pointer to a string variable containing the TR-31 key block to be disassembled.

opt_block_id

Direction: Input
Type: String

This parameter is used when operation keyword **DATA** is specified, otherwise it is ignored. For keyword **DATA**, this parameter is a pointer to a string variable that identifies the 2-byte ID of the optional block to obtain from the TR-31 key block.

num_opt_blocks

Direction: Input
Type: Integer

This parameter is used when operation keyword **INFO** is specified, otherwise it is ignored. For keyword **INFO**, this parameter is a pointer to an integer variable that specifies the number of 2-byte optional block IDs that are allocated for (1) the *opt_block_id*s variable and (2) the number of 2-byte integers that are allocated for the *opt_block_lengths* variable. This the value must specify the *exact* number of optional blocks that are in the header of the TR-31 key block. Use the TR31 Key Token Parse verb to determine the number of optional blocks IDs in a TR-31 key block before calling this verb.

opt_block_ids

Direction: Output
Type: String array

This parameter is used when operation keyword **INFO** is specified, otherwise it is ignored. For keyword **INFO**, this parameter is a pointer to a string array of 2-byte values that lists the identifiers of each optional block contained in the header of the TR-31 key block. Each ID must be unique, that is, no duplicates are allowed. The IDs, along with the associated lengths listed in the *opt_block_lengths* variable, are returned in the order that the optional blocks appear in the header of the TR-31 key block. The size of the variable must be at least two times the value of the *num_opts_blocks* variable.

opt_block_lengths

Direction: Output
Type: String array

This parameter is used when operation keyword **INFO** is specified, otherwise it is ignored. For keyword **INFO**, this parameter is a pointer to an integer array of 2-byte values that are 16-bit unsigned integers corresponding to the associated length of the optional block identified in the *opt_block_ids* variable. The lengths, along with the associated IDs listed in the *opt_block_ids* variable, are returned in the order that the optional blocks appear in the header of the TR-31 key block. The size of the variable must be at least two times the value of the *num_opts_blocks* variable.

opt_block_data_length

Direction: Input/Output
Type: Integer

This parameter is used when operation keyword **DATA** is specified, otherwise it is ignored. For keyword **DATA**, this parameter is a pointer to an integer variable containing the length of the *opt_block_data* parameter. On input, this variable specifies the maximum permissible length of the result. On output, the verb updates the value to length of the returned optional block data.

opt_block_data

Direction: Output
Type: String

This parameter is used when operation keyword **DATA** is specified, otherwise it is ignored. For keyword **DATA**, this parameter is a pointer to a string variable. If the TR-31 key block is found to be valid and the TR-31 key block contains an optional block specified by the *optional_block_ID* variable, the optional block is copied into this variable if it is large enough. The *opt_block_data_length* variable is updated with the length of the data returned in the variable.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBT31RJ.

See "Building Java applications using the CCA JNI" on page 26.

Format

```
public native void CSNBT31RJ(
              hikmNativeInteger return_code,
              hikmNativeInteger reason_code,
              hikmNativeInteger exit_data_length,
              byte[]            exit_data,
              hikmNativeInteger rule_array_count,
              byte[]            rule_array,
              hikmNativeInteger tr31_key_length,
              byte[]            tr31_key,
              byte[]            opt_block_id,
              hikmNativeInteger num_opt_blocks,
              byte[]            opt_block_ids,
              byte[]            opt_block_lengths,
              hikmNativeInteger opt_block_data_length,
              byte[]            opt_block_data);
```

Chapter 15. Utility verbs

A utility verb is provided for code conversion.

- "Code Conversion (CSNBXEA)"

Code Conversion (CSNBXEA)

Use the Code Conversion utility to convert between ASCII data, EBCDIC data, or a custom 8-bit based conversion table.

Format

The format of CSNBXEA.

```
CSNBXEA(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             text_length,
             source_text,
             target_text,
             code_table_length,
             code_table)
```

Parameters

The parameters for CSNBXEA.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. This value must be 1.

rule_array

Direction: Input
Type: String array

An array of 8-byte keywords providing the processing control information. The keywords must be left-aligned and padded on the right with space characters. The rule_array keywords are described in Table 197.

Table 197. Keywords for the CSNBXEA utility

Keyword	Description
Service requested (One, required)	

Table 197. Keywords for the CSNBXEA utility (continued)

Keyword	Description
TOASCII	Converts the contents of **source_text** from EBCDIC character set to ASCII character set using the appropriate code table displayed in "Usage notes" on page 745. The result is placed in parameter **target_text**. The parameters **code_table** and **code_table_length** are ignored for this keyword.
TOEBCDIC	Converts the contents of **source_text** from ASCII character set to EBCDIC character set using the code table displayed in "Usage notes" on page 745. The result is placed in parameter **target_text**. The parameters **code_table** and **code_table_length** are ignored for this keyword.
USETABLE	Converts the contents of **source_text** from the original character set to a custom character set using the appropriate code table passed by the application in the **code_table** parameter. The result is placed in parameter **target_text**. Each byte of the **source_text** is used to index the **code_table** to discover the resulting byte for the **target_text**. This keyword assumes the conversion data is single-byte aligned: 1 byte of **source_text** is used to look up 1 byte of conversion value in the code table. Therefore the **code_table_length** must be at least 256 bytes. If the code_table_length is larger than 256 bytes the extra is ignored.

text_length

Direction: Input
Type: Integer

A pointer to an integer variable that contains the length of the **source_text** parameter. The length must be a positive nonzero value.

source_text

Direction: Input
Type: String

This parameter contains the string to be converted.

target_text

Direction: Output
Type: String

This parameter contains the converted text that is returned by this verb. The size of this buffer must be at least as long as indicated by **text_length**.

code_table_length

Direction: Input
Type: Integer

The size in bytes of the buffer passed as the **code_table** parameter.

code_table

Direction: Input
Type: String

A data conversion table specified by the user for converting the contents of **source_text** to **target_text**.

Restrictions

The restrictions for CSNBXEA.

None.

Required commands

The required commands for CSNBXEA.

None.

Usage notes

The usage notes for CSNBXEA. Also the conversion tables for EBCDIC to ASCII and for ASCII to EBCDIC are listed.

This service is built to provide exact correspondence to the functions provided by the ICSF verbs CSNBXEA and CSNBXAE. The default conversion tables for EBCDIC-to-ASCII and ASCII-to-EBCDIC (see Table 198 on page 746 and Table 199 on page 747) match those used for the ICSF verbs. The origin of these conversion tables was a comparison of the EBCDIC 1047 code page to a widely used mapping for Extended ASCII. Round-trip conversions (achievable with two calls to this service for ASCII-to-EBCDIC-to-ASCII or EBCDIC-to-ASCII-to-EBCDIC) provide the original data as output.

This service is structured differently than the other services. It runs in the caller's address space in the caller's key and mode. The adapter need not be active for you to run this service.

Table 198. EBCDIC to ASCII conversion table

EBC	ASC	EBC	ASC	EBC	ASC	EBC	ASC	EBC	ASC	EBC	ASC	EBC	ASC	EBC	ASC
00	00	20	81	40	20	60	2D	80	F8	A0	C8	C0	7B	E0	5C
01	01	21	82	41	A6	61	2F	81	61	A1	7E	C1	41	E1	E7
02	02	22	1C	42	E1	62	DF	82	62	A2	73	C2	42	E2	53
03	03	23	84	43	80	63	DC	83	63	A3	74	C3	43	E3	54
04	CF	24	86	44	EB	64	9A	84	64	A4	75	C4	44	E4	55
05	09	25	0A	45	90	65	DD	85	65	A5	76	C5	45	E5	56
06	D3	26	17	46	9F	66	DE	86	66	A6	77	C6	46	E6	57
07	7F	27	1B	47	E2	67	98	87	67	A7	78	C7	47	E7	58
08	D4	28	89	48	AB	68	9D	88	68	A8	79	C8	48	E8	59
09	D5	29	91	49	8B	69	AC	89	69	A9	7A	C9	49	E9	5A
0A	C3	2A	92	4A	9B	6A	BA	8A	96	AA	EF	CA	CB	EA	A0
0B	0B	2B	95	4B	2E	6B	2C	8B	A4	AB	C0	CB	CA	EB	85
0C	0C	2C	A2	4C	3C	6C	25	8C	F3	AC	DA	CC	BE	EC	8E
0D	0D	2D	05	4D	28	6D	5F	8D	AF	AD	5B	CD	E8	ED	E9
0E	0E	2E	06	4E	2B	6E	3E	8E	AE	AE	F2	CE	EC	EE	E4
0F	0F	2F	07	4F	7C	6F	3F	8F	C5	AF	F9	CF	ED	EF	D1
10	10	30	E0	50	26	70	D7	90	8C	B0	B5	D0	7D	F0	30
11	11	31	EE	51	A9	71	88	91	6A	B1	B6	D1	4A	F1	31
12	12	32	16	52	AA	72	94	92	6B	B2	FD	D2	4B	F2	32
13	13	33	E5	53	9C	73	B0	93	6C	B3	B7	D3	4C	F3	33
14	C7	34	D0	54	DB	74	B1	94	6D	B4	B8	D4	4D	F4	34
15	B4	35	1E	55	A5	75	B2	95	6E	B5	B9	D5	4E	F5	35
16	08	36	EA	56	99	76	FC	96	6F	B6	E6	D6	4F	F6	36
17	C9	37	04	57	E3	77	D6	97	70	B7	BB	D7	50	F7	37
18	18	38	8A	58	A8	78	FB	98	71	B8	BC	D8	51	F8	38
19	19	39	F6	59	9E	79	60	99	72	B9	BD	D9	52	F9	39
1A	CC	3A	C6	5A	21	7A	3A	9A	97	BA	8D	DA	A1	FA	B3
1B	CD	3B	C2	5B	24	7B	23	9B	87	BB	D9	DB	AD	FB	F7
1C	83	3C	14	5C	2A	7C	40	9C	CE	BC	BF	DC	F5	FC	F0
1D	1D	3D	15	5D	29	7D	27	9D	93	BD	5D	DD	F4	FD	FA
1E	D2	3E	C1	5E	3B	7E	3D	9E	F1	BE	D8	DE	A3	FE	A7
1F	1F	3F	1A	5F	5E	7F	22	9F	FE	BF	C4	DF	8F	FF	FF

Table 199. ASCII to EBCDIC conversion table

ASC	EBC	ASC	EBC	ASC	EBC	ASC	EBC	ASC	EBC	ASC	EBC	ASC	EBC	ASC	EBC
00	00	20	40	40	7C	60	79	80	43	A0	EA	C0	AB	E0	30
01	01	21	5A	41	C1	61	81	81	20	A1	DA	C1	3E	E1	42
02	02	22	7F	42	C2	62	82	82	21	A2	2C	C2	3B	E2	47
03	03	23	7B	43	C3	63	83	83	1C	A3	DE	C3	0A	E3	57
04	37	24	5B	44	C4	64	84	84	23	A4	8B	C4	BF	E4	EE
05	2D	25	6C	45	C5	65	85	85	EB	A5	55	C5	8F	E5	33
06	2E	26	50	46	C6	66	86	86	24	A6	41	C6	3A	E6	B6
07	2F	27	7D	47	C7	67	87	87	9B	A7	FE	C7	14	E7	E1
08	16	28	4D	48	C8	68	88	88	71	A8	58	C8	A0	E8	CD
09	05	29	5D	49	C9	69	89	89	28	A9	51	C9	17	E9	ED
0A	25	2A	5C	4A	D1	6A	91	8A	38	AA	52	CA	CB	EA	36
0B	0B	2B	4E	4B	D2	6B	92	8B	49	AB	48	CB	CA	EB	44
0C	0C	2C	6B	4C	D3	6C	93	8C	90	AC	69	CC	1A	EC	CE
0D	0D	2D	60	4D	D4	6D	94	8D	BA	AD	DB	CD	1B	ED	CF
0E	0E	2E	4B	4E	D5	6E	95	8E	EC	AE	8E	CE	9C	EE	31
0F	0F	2F	61	4F	D6	6F	96	8F	DF	AF	8D	CF	04	EF	AA
10	10	30	F0	50	D7	70	97	90	45	B0	73	D0	34	F0	FC
11	11	31	F1	51	D8	71	98	91	29	B1	74	D1	EF	F1	9E
12	12	32	F2	52	D9	72	99	92	2A	B2	75	D2	1E	F2	AE
13	13	33	F3	53	E2	73	A2	93	9D	B3	FA	D3	06	F3	8C
14	3C	34	F4	54	E3	74	A3	94	72	B4	15	D4	08	F4	DD
15	3D	35	F5	55	E4	75	A4	95	2B	B5	B0	D5	09	F5	DC
16	32	36	F6	56	E5	76	A5	96	8A	B6	B1	D6	77	F6	39
17	26	37	F7	57	E6	77	A6	97	9A	B7	B3	D7	70	F7	FB
18	18	38	F8	58	E7	78	A7	98	67	B8	B4	D8	BE	F8	80
19	19	39	F9	59	E8	79	A8	99	56	B9	B5	D9	BB	F9	AF
1A	3F	3A	7A	5A	E9	7A	A9	9A	64	BA	6A	DA	AC	FA	FD
1B	27	3B	5E	5B	AD	7B	C0	9B	4A	BB	B7	DB	54	FB	78
1C	22	3C	4C	5C	E0	7C	4F	9C	53	BC	B8	DC	63	FC	76
1D	1D	3D	7E	5D	BD	7D	D0	9D	68	BD	B9	DD	65	FD	B2
1E	35	3E	6E	5E	5F	7E	A1	9E	59	BE	CC	DE	66	FE	9F
1F	1F	3F	6F	5F	6D	7F	07	9F	46	BF	BC	DF	62	FF	FF

JNI version

This verb has a Java Native Interface (JNI) version, which is namedCSNBXEAJ .

See “Building Java applications using the CCA JNI” on page 26.

Format

```
public native void CSNBXEAJ(
           hikmNativeInteger    return_code,
           hikmNativeInteger    reason_code,
```

```
|                                hikmNativeInteger    exit_data_length,
|                                byte[]               exit_data,
|                                hikmNativeInteger    rule_array_count,
|                                byte[]               rule_array,
|                                hikmNativeInteger    text_length,
|                                byte[]               source_text,
|                                byte[]               target_text,
|                                hikmNativeInteger    code_table_length,
|                                byte[]               code_table);

|
```

Part 3. Reference information

The provided reference information informs about key formats, cryptographic algorithms, as well as return and reason codes when you program with CCA.

You can find the following reference information:

Chapter 16. Return codes and reason codes

Read the contained reference information that describes the return codes and reason codes reported at the conclusion of verb processing.

Reason code numbers narrow down the meaning of a return code. All reason code numbers are unique and associated with a single return code. Generally, you can base your application program design on the return codes.

Each verb supplies a return code and a reason code in the variables identified by the *return_code* and *reason_code* parameters. See "Parameters common to all verbs" on page 20.

Return codes

A return code provides a general indication of the results of verb processing.

A return code can have the values shown in Table 200.

Table 200. Return code values

Hex value	Decimal value	Description
00	00	This return code indicates a normal completion of verb processing. To provide additional information, there are also nonzero reason codes associated with this return code.
04	04	This return code is a warning indicating the verb completed processing; however, an unusual event occurred. The event is most likely related to a problem created by the user or is a normal occurrence based on the data supplied to the verb.
08	08	This return code indicates the verb prematurely stopped processing. Generally, the application programmer needs to investigate the significance of the associated reason code to determine the origin of the problem. In some cases, due to transient conditions, retrying the verb might produce different results.
0C	12	This return code indicates the verb prematurely stopped processing. Either a coprocessor is not available or a processing error occurred. The reason is most likely related to a problem in the set up of the hardware or in the configuration of the software.
10	16	This return code indicates the verb prematurely stopped processing. A processing error occurred. If these errors persist, a repair of the coprocessor hardware or a correction to the coprocessor software might be required.

Note: If an application receives a return code greater than 4, an error occurred. In the case of an error, assume any output variables other than the return code and reason code are not valid, unless otherwise indicated in the description of verb processing.

Reason codes

A reason code details the results of verb processing.

Every reason code is associated with a single return code. A nonzero reason code can be associated with a zero return code.

User Defined Extensions (UDX) return reason codes in the range of 20480 (X'5000') - 24575 (X'5FFF').

The remainder of this topic lists the reason codes that accompany each of the return codes. The return codes are shown in decimal form and the reason codes are shown in decimal and in hexadecimal (hex) form.

Reason codes that accompany return code 0

Reason codes that accompany return code 0.

These codes are listed in Table 201.

Table 201. Reason codes for return code 0

Return code, decimal	Reason code, decimal (hex)	Description
0	000 (000)	The verb completed processing successfully.
0	002 (002)	One or more bytes of a key do not have odd parity.
0	008 (008)	No value is present to be processed.
0	151 (097)	The key token supplies the MAC length or MACLEN4 is the default for key tokens that contain MAC or MACVER keys.
0	701 (2BD)	A new master-key value has duplicate thirds.
0	702 (2BE)	A provided master-key part does not have odd parity. See “Master Key Process (CSNBMKP)” on page 122 about parity requirements for master key parts.
0	2013 (7DD)	The Pending Change Buffer (PCB) is empty. This return code and reason code pair applies only to IBM z Systems.
0	2146 (862)	A weaker key was used to wrap a stronger key and the **Warn when weak wrap - Transport keys** command (offset X'032C') was enabled in the active role.
0	2173 (87D)	The specified payload format version for the output key token matches the payload format version of the input key token.
0	3010 (BC2)	This card is currently disabled. A card is placed in this state so that it can be moved from one piece of hardware to another, while keeping its secret keys and master keys intact. Normally, when a card has been moved a 'tamper' event is recorded and all secrets are erased. A TKE workstation is typically required to put a card in this state and to remove it from this state after the card is installed on the new hardware. This return code and reason code pair applies only to IBM z Systems.
0	10001 (2711)	A key encrypted under the old master key was used.
0	10 002 (2712)	A fully qualified dataset name is longer than 64 bytes and the environment variable CSU*xxx*LD is not defined (where *xxx* is either AES, DES, or PKA). The current directory has been abbreviated as a single dot (period).
0	10 003 (2713)	A fully qualified dataset name is longer than 64 bytes and the environment variable CSU*xxx*LD is defined (where *xxx* is either AES, DES, or PKA). Only the dataset name is returned. Use the CSU*xxx*LD environment variable to determine the fully qualified dataset name.

Reason codes that accompany return code 4

Reason codes that accompany return code 4

These codes are listed in Table 202.

Table 202. Reason codes for return code 4

Return code, decimal	Reason code, decimal (hex)	Description
4	001 (001)	The verification test failed.
4	013 (00D)	The key token has an initialization vector and the *initialization_vector* parameter value is nonzero. The verb uses the value in the key token.
4	016 (010)	The *rule_array* and the *rule_array_count* are too small to contain the complete result.
4	017 (011)	The requested ID is not present in any profile in the specified cryptographic hardware component.
4	019 (013)	The financial PIN in a PIN block is not verified.
4	158 (09E)	The verb did not process any key records.
4	166 (0A6)	The control-vector is not valid because of parity bits, anti-variant bits, inconsistent KEK bits or because bits 59 - 62 are not zero.
4	179 (0B3)	The control-vector keywords in the *rule_array* are ignored.
4	195 (C3)	The key or key-part rule keyword provided does not match the length of the key in the enciphered key token. The output is based on the length specified, and not on the actual key length.
4	283 (11B)	The coprocessor battery is low.
4	287 (11F)	The PIN-block format is not consistent.
4	429 (1AD)	The digital signature is not verified. The verb completed its processing normally.
4	1024 (400)	Sufficient shares have been processed to create a new master key.
4	2039 (7F7)	At least one control vector bit cannot be parsed.
4	2042 (7FA)	The supplied passphrase is not valid.
4	2133 (855)	The *verb_data* value identifies one or more PIN decimalization tables to be deleted that are not stored on the coprocessor. All PIN tables that were requested to be deleted are removed.
4	2162 (872)	At least two of the key parts of a new operational or master key have identical parts and a warning has been requested by the setting of an appropriate access control point.

Reason codes that accompany return code 8

Reason codes that accompany return code 8.

The codes are listed in Table 203.

Table 203. Reason codes for return code 8

Return code, decimal	Reason code, decimal (hex)	Description
8	012 (00C)	The token-validation value in an external key token is not valid.
8	022 (016)	The ID number in the request field is not valid.
8	023 (017)	An access to the data area is outside the data-area boundary.

Table 203. Reason codes for return code 8 (continued)

Return code, decimal	Reason code, decimal (hex)	Description
8	024 (018)	The master key verification pattern is not valid.
8	025 (019)	The value that the *text_length* parameter specifies is not valid.
8	026 (01A)	The value of the PIN is not valid.
8	029 (01D)	The token-validation value in an internal key token is not valid.
8	030 (01E)	No record with a matching key label is in key storage.
8	031 (01F)	The control vector does not specify a DATA key.
8	032 (020)	A key label format is not valid.
8	033 (021)	A *rule_array* or other parameter specifies a keyword that is not valid.
8	034 (022)	A *rule_array* keyword combination is not valid.
8	035 (023)	A *rule_array_count* is not valid.
8	036 (024)	The action command must be specified in the *rule_array*.
8	037 (025)	The object type must be specified in the *rule_array*.
8	039 (027)	A control vector violation occurred. Check all control vectors employed with the verb. For security reasons, no detail is provided.
8	040 (028)	The service code does not contain numerical character data.
8	041 (029)	The keyword supplied with the *key_form* parameter is not valid.
8	042 (02A)	The expiration date is not valid.
8	043 (02B)	The keyword supplied with the *key_length* or the *key_token_length* parameter is not valid.
8	044 (02C)	A record with a matching key label already exists in key storage.
8	045 (02D)	The input character string cannot be found in the code table.
8	046 (02E)	The card-validation value (CVV) is not valid.
8	047 (02F)	A source key token is unusable because it contains data that is not valid or is undefined.
8	048 (030)	One or more keys has a master key verification pattern that is not valid.
8	049 (031)	A key-token-version-number found in a key token is not supported.
8	050 (032)	The key-serial-number specified in the *rule_array* is not valid.
8	051 (033)	The value that the *text_length* parameter specifies is not a multiple of eight bytes.
8	054 (036)	The value that the *pad_character* parameter specifies is not valid.
8	055 (037)	The initialization vector in the key token is enciphered.
8	056 (038)	The master key verification pattern in the OCV is not valid.
8	058 (03A)	The parity of the operating key is not valid.
8	059 (03B)	Control information (for example, the processing method or the pad character) in the key token conflicts with that in the *rule_array*.
8	060 (03C)	A cryptographic request with the FIRST or MIDDLE keywords and a text length less than eight bytes is not valid.
8	061 (03D)	The keyword supplied with the *key_type* parameter is not valid.
8	062 (03E)	The source key is not present.
8	063 (03F)	A key token has an invalid token header (for example, not an internal token).

Table 203. Reason codes for return code 8 (continued)

Return code, decimal	Reason code, decimal (hex)	Description
8	064 (040)	The RSA key is not permitted to perform the requested operation. Likely cause is key distribution usage is not enabled for the key.
8	065 (041)	The key token failed consistency checking.
8	066 (042)	The recovered encryption block failed validation checking.
8	067 (043)	RSA encryption failed.
8	068 (044)	RSA decryption failed.
8	070 (046)	An invalid block identifier (identifier tag) was found. Either a block ID (identifier tag) that was proprietary was found, a reserved block ID was used, a duplicate block ID was found, or the specified optional block in the TR-31 key block could not be found.
8	072 (048)	The value that the size parameter specifies is not valid (too small, too large, negative, or zero).
8	085 (055)	The date or the time value is not valid.
8	090 (05A)	Access control checking failed. See the Required Commands descriptions for the failing verb.
8	091 (05B)	The time that was sent in your logon request was more than five minutes different from the clock in the secure module.
8	092 (05C)	The user profile is expired.
8	093 (05D)	The user profile has not yet reached its activation date.
8	094 (05E)	The authentication data (for example, passphrase) is expired.
8	095 (05F)	Access to the data is not authorized.
8	096 (060)	An error occurred reading or writing the secure clock.
8	100 (064)	The PIN length is not valid.
8	101 (065)	The PIN check length is not valid. It must be in the range from 4 to the PIN length inclusive.
8	102 (066)	The value of the decimalization table is not valid.
8	103 (067)	The value of the validation data is not valid.
8	104 (068)	The value of the customer-selected PIN is not valid or the PIN length does not match the value supplied with the *PIN_length* parameter or defined by the PIN-block format specified in the PIN profile.
8	105 (069)	The value of the *transaction_security* parameter is not valid.
8	106 (06A)	The PIN-block format keyword is not valid.
8	107 (06B)	The format control keyword is not valid.
8	108 (06C)	The value or the placement of the padding data is not valid.
8	109 (06D)	The extraction method keyword is not valid.
8	110 (06E)	The value of the PAN data is not numeric character data.
8	111 (06F)	The sequence number is not valid.
8	112 (070)	The PIN offset is not valid.
8	114 (072)	The PVV value is not valid.
8	116 (074)	The clear PIN value is not valid. For example, digits other than 0 - 9 were found.
8	118 (76)	The issuer domestic code is invalid. This value must be five alphanumeric characters.
8	120 (078)	An origin or destination identifier is not valid.

Table 203. Reason codes for return code 8 (continued)

Return code, decimal	Reason code, decimal (hex)	Description
8	121 (079)	The value of the *inbound_key* or *source_key* parameter is not valid.
8	122 (07A)	The value of the *inbound_KEK_count* or *outbound_count* parameter is not valid.
8	125 (07D)	A PKA92-encrypted key having the same Environment Identifier (EID) as the local node cannot be imported.
8	129 (081)	Required rule-array keyword not found.
8	153 (099)	The text length exceeds the system limits.
8	154 (09A)	The key token the *key_identifier* parameter specifies is not an internal key-token or a key label.
8	155 (09B)	The value that the *generated_key_identifier* parameter specifies is not valid or it is not consistent with the value that the *key_form* parameter specifies.
8	156 (09C)	A keyword is not valid with the specified parameters.
8	157 (09D)	The key-token type is not specified in the *rule_array*.
8	159 (09F)	The keyword supplied with the option parameter is not valid.
8	160 (0A0)	The key type and the key length are not consistent.
8	161 (0A1)	The value that the *dataset_name_length* parameter specifies is not valid.
8	162 (0A2)	The offset value is not valid.
8	163 (0A3)	The value that the *dataset_name* parameter specifies is not valid.
8	164 (0A4)	The starting address of the output area falls inside the input area.
8	165 (0A5)	The carry_over_character_count specified in the chaining vector is not valid.
8	168 (0A8)	A hexadecimal MAC value contains characters that are not valid or the MAC, on a request or reply failed, because the user session key in the host and the adapter card do not match.
8	169 (0A9)	Specific to MDC Generate, indicates that the length of the text supplied is not correct, either not long enough for the algorithm parameters used or not the correct multiple (must be multiple of eight bytes).
8	170 (0AA)	Special authorization through the operating system is required to use this verb.
8	171 (0AB)	The control_array_count value is not valid.
8	175 (0AF)	The key token cannot be parsed because no control vector is present.
8	180 (0B4)	A key token presented for parsing is null.
8	181 (0B5)	The key token is not valid. The first byte is not valid or an incorrect token type was presented.
8	183 (0B7)	The key type is not consistent with the key type of the control vector.
8	184 (0B8)	An input pointer is null.
8	185 (0B9)	A disk I/O error occurred: perhaps the file is in-use, does not exist, and so forth.
8	186 (0BA)	The key-type field in the control vector is not valid.
8	187 (0BB)	The requested MAC length (MACLEN4, MACLEN6, MACLEN8) is not consistent with the control vector (key-A, key-B).
8	191 (0BF)	The requested MAC length (MACLEN6, MACLEN8) is not consistent with the control vector (MAC-LN-4).
8	192 (0C0)	A key-storage record contains a record validation value that is not valid.
8	204 (0CC)	A memory allocation failed. This can occur in the host and in the coprocessor. Try closing other host tasks. If the problem persists, contact the IBM support center.

Table 203. Reason codes for return code 8 (continued)

Return code, decimal	Reason code, decimal (hex)	Description
8	205 (0CD)	The X9.23 ciphering method is not consistent with the use of the CONTINUE keyword.
8	323 (143)	The ciphering method the Decipher verb used does not match the ciphering method the Encipher verb used.
8	335 (14F)	Either the specified cryptographic hardware component or the environment cannot implement this function.
8	340 (154)	One of the input control vectors has odd parity.
8	343 (157)	Either the data block or the buffer for the block is too small or a variable has caused an attempt to create an internal data structure that is too large.
8	345 (159)	Insufficient storage space exists for the data in the data block buffer.
8	374 (176)	Less data was supplied than expected or less data exists than was requested.
8	377 (179)	A key-storage error occurred.
8	382 (17E)	A time-limit violation occurred.
8	385 (181)	The cryptographic hardware component reported that the data passed as part of a command is not valid for that command.
8	387 (183)	The cryptographic hardware component reported that the user ID or role ID is not valid.
8	393 (189)	The command was not processed because the profile cannot be used.
8	394 (18A)	The command was not processed because the expiration date was exceeded.
8	397 (18D)	The command was not processed because the active profile requires the user to be verified first.
8	398 (18E)	The command was not processed because the maximum PIN or password failure limit is exceeded.
8	407 (197)	There is a PIN-block consistency-check-error.
8	439 (1B7)	Key cannot be completed because all required key parts have not yet been accumulated, or key is already complete.
8	441 (1B9)	Key part cannot be added because key is complete. The key to be processed should be partial, but the key is not partial according to the control vector or other control bits of the key.
8	442 (1BA)	DES keys with replicated halves are not allowed.
8	605 (25D)	The number of output bytes is greater than the number that is permitted.
8	703 (2BF)	A new master-key value is one of the weak DES keys.
8	704 (2C0)	A new master key cannot have the same master key version number as the current master-key.
8	705 (2C1)	Both exporter keys specify the same key-encrypting key.
8	706 (2C2)	Pad count in deciphered data is not valid.
8	707 (2C3)	The master-key registers are not in the state required for the requested function.
8	714 (2CA)	A reserved parameter must be a null pointer or an expected value.
8	715 (2CB)	A parameter that must have a value of zero is not valid.
8	718 (2CE)	The hash value of the data block in the decrypted RSA-OAEP block does not match the hash of the decrypted data block.
8	719 (2CF)	The block format (BT) field in the decrypted RSA-OAEP block does not have the correct value.

Table 203. Reason codes for return code 8 (continued)

Return code, decimal	Reason code, decimal (hex)	Description
8	720 (2D0)	The initial byte (I) in the decrypted RSA-OAEP block does not have a valid value.
8	721 (2D1)	The V field in the decrypted RSA-OAEP does not have the correct value.
8	752 (2F0)	The key-storage file path is not usable.
8	753 (2F1)	Opening the key-storage file failed.
8	754 (2F2)	An internal call to the key_test command failed.
8	756 (2F4)	Creation of the key-storage file failed.
8	760 (2F8)	An RSA-key modulus length in bits or in bytes is not valid.
8	761 (2F9)	An RSA-key exponent length is not valid.
8	762 (2FA)	A length in the key value structure is not valid.
8	763 (2FB)	The section identification number within a key token is not valid.
8	770 (302)	The PKA key token has a field that is not valid.
8	771 (303)	The user is not logged on.
8	772 (304)	The requested role does not exist.
8	773 (305)	The requested profile does not exist.
8	774 (306)	The profile already exists.
8	775 (307)	The supplied data is not replaceable.
8	776 (308)	The requested ID is already logged on.
8	777 (309)	The authentication data is not valid.
8	778 (30A)	The checksum for the role is in error.
8	779 (30B)	The checksum for the profile is in error.
8	780 (30C)	There is an error in the profile data.
8	781 (30D)	There is an error in the role data.
8	782 (30E)	The function-control-vector header is not valid.
8	783 (30F)	The command is not permitted by the function-control-vector value.
8	784 (310)	The operation you requested cannot be performed because the user profile is in use.
8	785 (311)	The operation you requested cannot be performed because the role is in use.
8	816 (330)	The public-key certificate length is invalid.
8	817 (331)	The public key does not match.
8	818 (332)	The signature of the input public-key certificate does not verify.
8	819 (333)	The public-key certificate type is invalid or not allowed.
8	1025 (401)	The registered public key or retained private key name already exists.
8	1026 (402)	The key name (registered public key or retained private key) does not exist.
8	1027 (403)	Environment identifier data is already set.
8	1028 (404)	Master key share data is already set.
8	1029 (405)	There is an error in the Environment Identifier (EID) data.
8	1030 (406)	There is an error in using the master key share data.
8	1031 (407)	There is an error in using registered public key or retained private key data.
8	1032 (408)	There is an error in using registered public key hash data.

Table 203. Reason codes for return code 8 (continued)

Return code, decimal	Reason code, decimal (hex)	Description
8	1033 (409)	The public key hash was not registered.
8	1034 (40A)	The public key was not registered.
8	1035 (40B)	The public key certificate signature was not verified.
8	1037 (40D)	There is a master key shares distribution error.
8	1038 (40E)	The public key hash is not marked for cloning.
8	1039 (40F)	The registered public key hash does not match the registered hash.
8	1040 (410)	The master key share enciphering key failed encipher.
8	1041 (411)	The master key share enciphering key failed decipher.
8	1042 (412)	The master key share digital signature generate failed.
8	1043 (413)	The master key share digital signature verify failed.
8	1044 (414)	There is an error in reading VPD data from the adapter.
8	1045 (415)	Encrypting the cloning information failed.
8	1046 (416)	Decrypting the cloning information failed.
8	1047 (417)	There is an error loading the new master key from the master key shares.
8	1048 (418)	The clone information has one or more sections that are not valid.
8	1049 (419)	The master key share index is not valid.
8	1050 (41A)	The public-key encrypted-key is rejected because the Environment Identifier (EID) with the key is the same as the EID for this node.
8	1051 (41B)	The private key is rejected because the key is not flagged for use in master-key cloning.
8	1052 (41C)	The token identifier of the trusted block's header section is in the range X'20' - X'FF'. Check the token identifier of the trusted block.
8	1053 (41D)	The active flag in the trusted block's trusted block section X'14' is not disabled. Use the Trusted Block Create verb to create an inactive/external trusted block.
8	1054 (41E)	The token identifier of the trusted block's header section is not X'1E' (external). Use the Trusted Block Create verb to create an inactive/external trusted block.
8	1055 (41F)	The active flag of the trusted block's trusted block section X'14' is not enabled. Use the Trusted Block Create verb to create an active/external trusted block.
8	1056 (420)	The token identifier of the trusted block's header section is not X'1F' (internal). Use the PKA Key Import verb to import the trusted block.
8	1057 (421)	The trusted block rule section X'12' rule ID does not match input parameter rule ID. Verify that the trusted block used has the rule section specified.
8	1058 (422)	The trusted block contains a value that is too small or too large.
8	1059 (423)	A trusted block parameter that must have a value of zero (or a grouping of bits set to zero) is invalid.
8	1060 (424)	The trusted block public key section failed consistency checking.
8	1061 (425)	The trusted block contains extraneous sections or subsections (TLVs). Check the trusted block for undefined sections or subsections.
8	1062 (426)	The trusted block contains missing sections or subsections (TLVs). Check the trusted block for required sections and subsections applicable to the verb invoked.
8	1063 (427)	The trusted block contains duplicate sections or subsections (TLVs). Check the trusted block's sections and subsections for duplicates. Multiple rule sections are allowed.

Table 203. Reason codes for return code 8 (continued)

Return code, decimal	Reason code, decimal (hex)	Description
8	1064 (428)	The trusted block expiration date has expired (as compared to the IBM 4764 clock). Validate the expiration date in the trusted block's trusted information section's Activation and Expiration Date TLV object
8	1065 (429)	The trusted block expiration date is at a date prior to the activation date. Validate the expiration date in the trusted block's trusted information section's Activation and Expiration Date TLV object.
8	1066 (42A)	The trusted block public key modulus length in bits is not consistent with the byte length. The bit length must be less than or equal to byte length * 8 and greater than (byte length - 1) * 8.
8	1067 (42B)	The trusted block public key modulus length in bits exceeds the maximum allowed bit length, as defined by the Function Control Vector.
8	1068 (42C)	One or more trusted block sections or TLV objects contained data that is invalid (an example would be invalid label data in label section X'13').
8	1069 (42D)	Trusted block verification was attempted by a verb other than CSNDDSV, CSNDKTC, CSNDPKI, CSNDRKX, or CSNDTBC.
8	1070 (42E)	The trusted block rule ID contained within a rule section has invalid characters.
8	1071 (42F)	The source key's length or CV does not match what is expected by the rule section in the trusted block that was selected by the rule ID input parameter.
8	1072 (430)	The activation data is not valid. Validate the activation data in the trusted block's trusted information section's Activation and Expiration Date TLV object.
8	1073 (431)	The source-key label does not match the template in the export key DES token parameters TLV object of the selected trusted block rule section.
8	1074 (432)	The control-vector value specified in the common export key parameters TLV object in the selected rule section of the trusted block contains a control vector that is not valid.
8	1075 (433)	The source-key label template in the export key DES token parameters TLV object in the selected rule section of the trusted block contains a label template that is not valid.
8	1077 (435)	Key wrapping option input error.
8	1078 (436)	Key wrapping Security Relevant Data Item (SRDI) error.
8	1079 (437)	The format of the decrypted PIN block is not supported in this function.
8	1100 (44C)	There is a general hardware device driver execution error.
8	1101 (44D)	There is a hardware device driver parameter that is not valid.
8	1102 (44E)	There is a hardware device driver non-valid buffer length.
8	1103 (44F)	The hardware device driver has too many opens. The device cannot open now.
8	1104 (450)	The hardware device driver is denied access.
8	1105 (451)	The hardware device driver device is busy and cannot perform the request now.
8	1106 (452)	The hardware device driver buffer is too small and the received data is truncated.
8	1107 (453)	The hardware device driver request is interrupted and the request is aborted.
8	1108 (454)	The hardware device driver detected a security tamper event.
8	1114 (45A)	The communications manager detected that the host-supplied buffer for the reply control block is too small.
8	1115 (45B)	The communications manager detected that the host-supplied buffer for the reply data block is too small.
8	1117 (45D)	Hardware device driver operation not permitted.

Table 203. Reason codes for return code 8 (continued)

Return code, decimal	Reason code, decimal (hex)	Description
8	1118 (45E)	Hardware device driver received bad address.
8	1119 (45F)	Hardware device driver hardware error.
8	1121 (461)	Hardware device driver firmware error.
8	1122 (462)	Hardware device driver temperature of out range.
8	1123 (463)	Hardware device driver received bad request.
8	1125 (465)	Hardware device driver host timeout.
8	2034 (7F2)	The environment variable that was used to set the default coprocessor is not valid, or does not exist for a coprocessor in the system.
8	2036 (7F4)	The contents of a chaining vector are not valid. Ensure the chaining vector was not modified by your application program.
8	2038 (7F6)	No RSA private key information is provided.
8	2041 (7F9)	A default card environment variable is not valid.
8	2050 (802)	The current key serial number field in the PIN profile variable is not valid (not hexadecimal or too many one bits).
8	2051 (803)	There is a non-valid message length in the OAEP-decoded information.
8	2053 (805)	No message found in the OAEP-decoded data.
8	2054 (806)	There is a non-valid RSA Enciphered Key cryptogram: OAEP optional encoding parameters failed validation.
8	2055 (807)	Based on the hash method and size of the symmetric key specified, the RSA public key size is too small to format the symmetric key into a PKOAEP2 message.
8	2062 (80E)	The active role does not permit you to change the characteristic of a double-length key in the *key_Part_Import* parameter.
8	2065 (811)	The specified key token is not null.
8	2080 (820)	The group profile was not found.
8	2081 (821)	The group has duplicate elements.
8	2082 (822)	The group profile is not in the group.
8	2083 (823)	The group has the wrong user ID count.
8	2084 (824)	The group user ID failed.
8	2085 (825)	The profile is not in the specified group.
8	2086 (826)	The group role was not found.
8	2087 (827)	The group profile has not been activated.
8	2088 (828)	The expiration date of the group profile has been reached or exceeded.
8	2089 (829)	The verb contains multiple keywords or parameters that indicate the algorithm to be used, and at least one of these specifies a different algorithm from the others.
8	2090 (82A)	A required SRDI was not found.
8	2091 (82B)	A required CA SRDI was not found.
8	2093 (82D)	Specific toIBM z Systems - an AES key is encrypted under a DES master key, which is not acceptable for the requested operation.
8	2095 (82F)	The *key_form* is incompatible with the *key_type*.
8	2097 (831)	The *key_length* is incompatible with the *key_type*.

Table 203. Reason codes for return code 8 (continued)

Return code, decimal	Reason code, decimal (hex)	Description
8	2098 (832)	Either a key bit length that was not valid was found in an AES key token (length not 128, 192, or 256 bits) or a version X'01' DES token had a token-marks field that was not valid.
8	2099 (833)	Invalid encrypted key length in the AES token, when an encrypted key is present.
8	2106 (83A)	An input/output error occurred while accessing the logged on users table.
8	2110 (83E)	Invalid wrapping type.
8	2111 (83F)	Control vector enhanced bit (bit 56) conflicts with key wrapping keyword.
8	2113 (841)	A key token contains invalid payload.
8	2114 (842)	Clear-key bit length is out of range.
8	2115 (843)	Input key token cannot have a key present when importing the first key part; skeleton key token is required.
8	2118 (846)	One or more invalid values in the TR-31 key block header.
8	2119 (847)	The "mode" value in the TR-31 header is invalid or is not acceptable in the chosen operation.
8	2121 (849)	The "algorithm" value in the TR-31 header is invalid or is not acceptable in the chosen operation.
8	2122 (84A)	For import, the exportability byte in the TR-31 header contains a value that does not support import of the key into CCA. For export, the requested exportability does not match circumstances (for example, a 'B' Key Block Version ID key can be wrapped only by a KEK that is wrapped in CBC mode, the ECB mode KEK violates ANSI X9.24).
8	2123 (84B)	The length of the cleartext key in the TR-31 block is invalid (for example, the algorithm is 'D' for single-length DES, but the key length is not 64 bits).
8	2125 (84D)	The Key Block Version ID in the TR-31 header contains an invalid value.
8	2126 (84E)	The key-usage field in the TR-31 header contains a value that is not supported for import of the key into CCA.
8	2127 (84F)	The key-usage field in the TR-31 header contains a value that is not valid with the other parameters in the header.
8	2129 (851)	Either a parameter for building a TR-31 key block (a TR-31 key block or a component, such as a tag for an optional block) contains one or more ASCII characters that are not printable as described in TR-31, or a field contains ASCII characters that are not allowed for that field.
8	2130 (852)	The control vector carried in the optional blocks of the TR-31 key block is inconsistent with other attributes of the key.
8	2131 (853)	The TR-31 key-token failed the MAC validate step of the Key Block unwrap and verify steps (for either Key Block Version ID method). MAC validation failed for a parameter in a key block, such as a trusted block or a TR-31 key block. This might be the result of tampering, corruption, or using a validation key that is different from the one use to generate the MAC.
8	2134 (856)	No valid PIN decimalization tables are present.
8	2135 (857)	The PIN decimalization table provided as input is not allowed to be used because it does not match any of the active tables stored on the coprocessor.
8	2137 (859)	There is an error involving the PIN decimalization table input data. No PIN tables have been changed.
8	2138 (85A)	At least one of the PIN decimalization tables requested to be activated is empty or already in the active state (not in the loaded state). No PIN tables have been activated.

Table 203. Reason codes for return code 8 (continued)

Return code, decimal	Reason code, decimal (hex)	Description
8	2139 (85B)	At least one PIN decimalization table provided as input to be activated does not match the corresponding table that is loaded on the coprocessor. No PIN tables have been changed from the loaded state to the active state.
8	2141 (84D)	The key verification pattern for the key-encrypting key is not valid.
8	2142 (85E)	A key-usage field setting prevents operation.
8	2143 (85F)	A key-management field setting prevents operation.
8	2145 (861)	An attempt to wrap a stronger key with a weaker key was disallowed.
8	2147 (863)	The key type to be generated is not valid.
8	2149 (865)	The key to be generated is stronger than the input material.
8	2151 (867)	At least one PIN decimalization table identifier provided as input is out of range or is a duplicate. No PIN tables have been changed.
8	2153 (869)	The input token is incompatible with the service (that is, clear key when encrypted key was expected).
8	2154 (86A)	At least one key token does not have the required key type for the specified function.
8	2158 (86E)	There is a mismatch between ECC key tokens of curve types, key lengths, or both. Curve types and key lengths must match.
8	2159 (86F)	A key-encrypting key is invalid.
8	2161 (871)	A wrap type, either requested or default, is in conflict with one or more input tokens.
8	2163 (873)	At least two of the key parts of a new operational or master key have identical parts and an error has been requested by the setting of an appropriate access control point.
8	2165 (875)	An RSA key token contains a private section that is not valid with this command.
8	2167 (877)	Invalid hash type in certificate.
8	2169 (879)	Invalid signature type in certificate.
8	2170 (87A)	Translation of text using an outbound key that has an effective key strength weaker than the effective strength of the inbound key is not allowed.
8	2174 (87E)	The provided data was not hexadecimal digits.
8	2175 (87F)	A weak PIN was presented. The PIN change has been rejected.
8	2177 (881)	The PAN presented to the PAN change verb was the same as the PAN in the encrypted PIN block. The change has been rejected.
8	2178 (882)	The PAN provided is inconsistent with a PAN incorporated in another piece of data.
8	2181 (885)	
8	2183 (887)	There is an error in the weak PIN entry structure input header length. No entries have been changed.
8	2185 (889)	For at least one of the inputs, the weak PIN entry requested to be activated is not in the loaded state. No weak PIN entries have been activated.
8	2186 (88A)	For at least one of the inputs, the weak PIN entry requested to be activated did not match the weak PIN entry structure to be activated. No weak PIN entries have been activated.
8	2187 (88B)	One or more of the weak PIN entry ID numbers in the input verb data was invalid, out or range, or a duplicate. No weak PIN entries have been changed.
8	2189 (88D)	There is an error in the weak PIN entry structure input type. No entries have been changed.

Table 203. Reason codes for return code 8 (continued)

Return code, decimal	Reason code, decimal (hex)	Description
8	2190 (88E)	There is an error in the weak PIN entry structure input header version. No entries have been changed.
8	2191 (88F)	There is an error in the weak PIN entry structure input header count. No entries have been changed.
8	2193 (891)	The presented PIN is a duplicate of one already in the table. No entries have been changed.
8	2194 (892)	Invalid or out of range passphrase length.
8	2197 (895)	The presented PIN failed verification. No processing has been done.
8	2198 (896)	The presented CMAC failed verification. No processing has been done.
8	2199 (897)	A variable-length symmetric key-token (version X'05') contains invalid key-usage field data.
8	2201 (899)	A variable-length symmetric key-token (version X'05') contains invalid key-management field data.
8	2203 (89B)	RSA engine check-sum error.
8	2227 (8B3)	The triple-length key cannot be imported because the TR-31 key block does not include a CCA control vector.
8	2229 (8B5)	The type of the specified key is not valid because a diversified key-generating key must be used to derive this symmetric key type.
8	2231 (8B7)	There was a problem converting or formatting the PAN.
8	2232 (8B8)	There was a problem converting or formatting the cardholder name.
8	2233 (8B9)	There was a problem converting or formatting the track 1 data.
8	2235 (8BB)	There was a problem converting or formatting the track 2 data.
8	2237 (8BD)	Data presented for VFPE processing is not in VFPE enciphered.
8	2239 (8BF)	The check digit compliance indicator/keyword denotes compliant check digit but the input PAN does not have a compliant check digit.
8	2849 (B21)	A verb data keyword specifies a keyword that is not valid.
8	2850 (B22)	A verb data keyword combination is not valid.
8	2851 (B23)	The verb data length value is not valid.
8	2945 (B81)	A required verb data keyword is not found.
8	3001 (BB9)	The RSA-OAEP block contains a PIN block and the verb did not request PINBLOCK processing.
8	3002 (BBA)	Specific to IBM z Systems - UDX already authorized.
8	3005 (BBD)	Specific to IBM z Systems - UDX not in UDX Authorization Table (UAT).
8	3006 (BBE)	Specific to IBM z Systems - UDX not authorized.
8	3007 (BBF)	Specific to IBM z Systems - Failed to obtain semaphore that guards the UAT.
8	3009 (BC1)	Specific to IBM z Systems - UDX Password hash mismatch.
8	3013 (BC5)	The longitudinal redundancy check (LRC) checksum in the AES key-token does not match the LRC checksum of the clear key.
8	3047 (BE7)	Use of clear key provided is not allowed. A secure key is required.
8	6000 (1770)	The specified device is already allocated.
8	6001 (1771)	No device is allocated.

Table 203. Reason codes for return code 8 (continued)

Return code, decimal	Reason code, decimal (hex)	Description
8	6002 (1772)	The specified device does not exist.
8	6003 (1773)	The specified device is an improper type.
8	6013 (177D)	The length of the cryptographic resource name is not valid.
8	6014 (177E)	The cryptographic resource name is not valid or does not refer to a coprocessor that is available in the system.
8	6015 (177F)	An ECC curve type is invalid or its usage is inconsistent.
8	6017 (1781)	Curve size *p* is invalid or its usage is inconsistent.
8	6018 (1782)	Error returned from CLiC module.
8	10028 (272C)	Specific to IBM z Systems - Invalid control vector in key token supplied.
8	10036 (2734)	Specific to IBM z Systems - Invalid control vectors (L-R) in key token supplied.
8	10044 (273C)	Specific to IBM z Systems - The *key_type* parameter and the CV key type for the supplied key token do not match.
8	10056 (2748)	Specific to IBM z Systems - The *key_type* parameter contains TOKEN, which is invalid for the requested operation.
8	10124 (278C)	Specific to IBM z Systems - The key id cannot be exported because of prohibit export restriction in the token supplied.
8	10128 (2790)	Specific to IBM z Systems - The NOCV-KEK *rule_array* keyword does not apply in this case.
8	10129 (2791)	Specific to IBM z Systems - The NOCV-KEK importer key or transport key is not allowed in the Remote Key Export operation requested.

Reason codes that accompany return code 12

Reason codes that accompany return code 12

The codes are listed in Table 204.

Table 204. Reason codes for return code 12

Return code, decimal	Reason code, decimal (hex)	Description
12	097 (061)	File space in key storage is insufficient to complete the operation.
12	196 (0C4)	The device driver, the security server, or the directory server is not installed or is not active. File permissions are not valid for your application.
12	197 (0C5)	There is a key-storage file I/O error or the file is not found.
12	206 (0CE)	The key-storage file is not valid or the master-key verification failed. There is an unlikely, but possible, synchronization problem with the Master Key Process verb.
12	207 (0CF)	The verification method flags in the profile are not valid.
12	319 (13F)	Passed to the CVV Verify or CVV Generate verb, the Verb Unique data corresponds to a PAN length of 19, but the overall length is wrong. This indicates that the host code is out of date.
12	324 (144)	There is insufficient memory available to process your request, either memory in the host computer or memory inside the coprocessor including the flash EPROM used to store keys, profiles, and other application data.

Table 204. Reason codes for return code 12 (continued)

Return code, decimal	Reason code, decimal (hex)	Description
12	338 (152)	This cryptographic hardware device driver is not installed or is not responding, or the CCA code is not loaded in the coprocessor.
12	764 (2FC)	The master keys are not loaded and, therefore, a key cannot be recovered or enciphered.
12	768 (300)	One or more paths for key-storage directory operations are improperly specified.
12	769 (301)	An internal error has occurred with the parameters to a cryptographic algorithm.
12	2007 (7D7)	The change type in the Pending Change Buffer is not recognized.
12	2015 (7DF)	The domain stored in the domain mask does not match what was included as the domain in the CPRB.
12	2017 (7E1)	The operation is attempting to call 'SET' for a master key, but has passed an invalid Master Key Verification Pattern.
12	2021 (7E5)	The card is disabled in the TKE path.
12	2037 (7F5)	Invalid domain specified.
12	2043 (7FB)	In the course of TKE communication through the host library to an adapter, a particular requested OA certificate was not found. A small number of these errors are typical when communication with a TKE is initiated.
12	2045 (7FD)	The CCA software is unable to claim a semaphore. The system might be short of resources.
12	2046 (7FE)	The CCA software is unable to list all the keys. The limit of 500,000 keys might have been reached.
12	2049 (801)	An error occurred while unlocking a semaphore in order to release the exclusive control of that semaphore.
12	2073 (819)	TKE command received when TKE disabled.
12	2074 (81A)	Invalid version found in Connectivity Programming Request/Reply Block (CPRB).
12	2101 (835)	Invalid AES flags in the function control vector (FCV).
12	2117 (845)	Thread specific CLiC objects are not in proper state.
12	2155 (86B)	The length of the fully qualified dataset name exceeds the maximum size that the verb can process.
12	2225 (8B1)	An internal outbound authentication manager error occurred, or the OA manager is disabled.
12	3046 (BE6)	The wrong usage was attempted in an operation with a retained key.
12	2242 (8C2)	The reply message block is too long for the host buffer.

Reason codes that accompany return code 16

Reason codes that accompany return code 16.

These codes are listed in Table 205.

Table 205. Reason codes for return code 16

Return code, decimal	Reason code, decimal (hex)	Description
16	099 (063)	An unrecoverable error occurred in the security server; contact the IBM support center.

Table 205. Reason codes for return code 16 (continued)

Return code, decimal	Reason code, decimal (hex)	Description
16	336 (150)	An error occurred in a cryptographic hardware or software component.
16	337 (151)	A device software error occurred.
16	339 (153)	A system error occurred in the interprocess communication routine.
16	444 (1BC)	The verb-unique-data has an invalid length.
16	556 (22C)	The request parameter block failed consistency checking.
16	708 (2C4)	The cryptographic engine is returning inconsistent data.
16	709 (2C5)	Cryptographic engine internal error. Could not access the master-key data.
16	710 (2C6)	An unrecoverable error occurred while attempting to update master-key data items.
16	712 (2C8)	An unexpected error occurred in the master-key manager.
16	800 (320)	A problem occurred in internal SHA operation processing.
16	2022 (7E6)	TKE-related internal file open error.
16	2047 (7FF)	Unable to transfer request data from host to coprocessor.
16	2057 (809)	Internal error: memory allocation failure.
16	2058 (80A)	Internal error: unexpected return code from OAEP routines.
16	2059 (80B)	Internal error: OAEP SHA-1 request failure.
16	2061 (80D)	Internal error in Symmetric Key Import, OAEP-decode: enciphered message too long.
16	2063 (80F)	The reply message too long for the requestor's command reply buffer.
16	2107 (83B)	Internal files failed verification check when loading from encrypted storage.
16	2150 (866)	An error occurred while attempting to open or save the DECTABLE SRDI that is stored on the coprocessor.
16	2195 (893)	An error occurred reading the weak PIN file stored on the coprocessor.

Chapter 17. Key token formats

The key token formats can be useful for debugging purposes.

This information unit provides the formats for:

AES internal key token

The format for an AES internal key token.

Table 206 shows the format for an AES internal key token.

CCA AES key-token data structures are 64 bytes in length, and are made up of an internal key-token identifier and a token version number, reserved fields, a flag byte containing various flag bits, and a token-validation value.

Depending on the flag byte, the key token either contains an encrypted key, a clear key, or the key is absent. An encrypted key is encrypted under an AES master key identified by a master-key verification pattern (MKVP) in the key token. The key token contains a two-byte integer that specifies the length of the clear-key value in bits, valued to 0, 128, 192, or 256, and a two-byte integer that specifies the length of the encrypted-key value in bytes, valued to 0 or 32. An LRC checksum byte of the clear-key value is also in the key token.

All AES keys are DATA keys. If the flag byte indicates a control vector (CV) is present, it must be all binary zeros. An all-zero CV represents the CV value of an AES DATA key. If a key is present without a control vector in a key token, that is accepted and the key is interpreted as an AES DATA key. The AES internal key-token is the structure used to hold AES keys that are either encrypted with the AES master-key, or in cleartext format.

Table 206. AES Internal key token format, version X'04'

Bytes	Description
0	X'01' (flag indicating this is an internal key token)
1 - 3	Implementation-dependent bytes, must be X'000000'.

Table 206. AES Internal key token format, version X'04' (continued)

Bytes	Description
4	Key token version number, X'04'
5	Reserved (X'00')
6	Flag byte. See "AES internal key-token flag byte."
7	Longitudinal redundancy check (LRC) checksum of clear-key value (LRC is the XOR of each byte in the clear-key value).
8 - 15	Master key verification pattern (MKVP) Contains the master-key verification pattern of the AES master-key used to encrypt the key contained in the token, or binary zeros if the token does not contain a key or the key is in the clear. The MKVP is calculated as the leftmost eight bytes of the SHA-256 hash of the string formed by pre-pending the byte X'01' to the cleartext master-key value.
16 - 47	Key value, if present. Contains either: • A 256-bit encrypted-key value. The clear key value is padded on the right with binary zeros, and the entire 256-bit value is encrypted under the AES master-key using AES CBC mode with an initialization vector of binary zeros. • A 128-bit, 192-bit, or 256-bit clear-key value left-aligned and padded on the right with binary zeros for the entire 256-bit field.
48 - 55	Control Vector (CV) This value must be binary zeros for all AES key tokens that have a control vector present.
56 - 57	Clear-key bit length An integer specifying the length in bits of the clear-key value. If no key is present in a completed token, this length is zero. In a skeleton token, this is the length of the key to be created in the token when used as input to the Key Generate verb.
58 - 59	Encrypted-key byte length An integer specifying the length in bytes of the encrypted-key value. This value is zero if the token does not contain a key or the key is in the clear.
60 - 63	Token validation value (TVV).

AES internal key-token flag byte

The format for an AES internal key token flag byte.

Table 207 shows the format for an AES internal key token flag byte.

Table 207. AES internal key-token flag byte

Bits (MSB...LSB)[1]	Description
1*xxx xxxx*	Key is encrypted under the AES master-key (ignored if no key present).
0*xxx xxxx*	Key is in the clear (ignored if no key present).
*x*1*xx xxxx*	Control vector (CV) is present.
*xx*1*x xxxx*	No key and no MKVP present.
*xx*0*x xxxx*	Encrypted or clear key present, MKVP present if key is encrypted.
Note: All undefined bits are reserved and must be 0.	

Token Validation Value

CCA uses the *Token Validation Value (TVV)* to verify that a token is valid.

The TVV prevents a key token that is not valid or that is overlaid from being accepted by CCA. It provides a checksum to detect a corruption in the key token.

When an CCA verb generates a key token, it generates a TVV and stores the TVV in bytes 60-63 of the key token. When an application program passes a key token to a verb, CCA checks the TVV. To generate the TVV, CCA performs a twos complement ADD operation (ignoring carries and overflow) on the key token, operating on four bytes at a time, starting with bytes 0-3 and ending with bytes 56-59.

Format of the clear key token

The format for a clear internal key token.

Table 208 shows the format for a clear internal key token.

Table 208. Internal clear key token format

Bytes	Description
0	X'01' (flag indicating this is an internal key token)
1 - 3	Implementation-dependent bytes (X'000000' for ICSF)
4	Key token version number (X'00' or X'01')
5	Reserved (X'00')
6	Flag byte **Bit** **Meaning When Set On** **0** Encrypted key and master key verification pattern (MKVP) are present. This will be off for clear keys. **1** Control vector (CV) value in this token has been applied to the key. This will be off for clear keys. **2 - 7** reserved
7 - 15	Reserved (X'00')
16 - 23	A single-length key, the left half of a double-length key, or Part A of a triple-length key.
24 - 31	X'0000000000000000' if a single-length key, the right half of a double-length operational key, or Part B of a triple-length operational key.
32 - 47	Reserved for clear key tokens (X'00's')
48 - 55	X'0000000000000000' if a single-length key or double-length key, or Part C of a triple-length operational key.
56 - 58	Reserved (X'000000')
59 bits 0 and 1	B'00' reserved
59 bits 2 and 3	**Value** **Description** **B'00'** Indicates single-length key (version 0 only). **B'01'** Indicates double-length key (version 1 only). **B'10'** Indicates triple-length key (version 1 only).
59 bits 4 - 7	B'0000'
60 - 63	Token validation value (TVV).

DES internal key token

The format for a DES internal key token.

Table 209 shows the format for a DES internal key token.

Table 209. DES internal key token format

Bytes	Description
0	X'01' (flag indicating this is an internal key token)
1 - 3	Implementation-dependent bytes (X'000000' for ICSF)
4	Key token version number (X'00' or X'01')
5	Reserved (X'00')
6	Flag byte **Bit** **Meaning When Set On** **0** Encrypted key and master key verification pattern (MKVP) are present. **1** Control vector (CV) value in this token has been applied to the key. **2** Key is used for no control vector (NOCV) processing. Valid for transport keys only. **3** Key is an ANSI key-encrypting key (AKEK). **4** AKEK is a double-length key (16 bytes). **Note:** When bit 3 is on and bit 4 is off, AKEK is a single-length key (eight bytes). **5** AKEK is partially notarized. **6** Key is an ANSI partial key. **7** Export prohibited.
7	Reserved (X'00')
8 - 15	Master key verification pattern (MKVP)
16 - 23	A single-length key, the left half of a double-length key, or Part A of a triple-length key. The value is encrypted under the master key.
24 - 31	X'0000000000000000' if a single-length key, the right half of a double-length operational key, or Part B of a triple-length operational key. The right half of the double-length key or Part B of the triple-length key is encrypted under the master key.
32 - 39	The control vector (CV) for a single-length key or the left half of the control vector for a double-length key.
40 - 47	X'0000000000000000' if a single-length key or the right half of the control vector for a double-length operational key.
48 - 55	X'0000000000000000' if a single-length key or double-length key, or Part C of a triple-length operational key. Part C of a triple-length key is encrypted under the master key.
56 - 58	Reserved (X'000000')
59 bits 0 and 1	**Value** **Description** **B'10'** Indicates KEK. **B'00'** Indicates DES for DATA keys or the system default algorithm for a KEK. **B'01'** Indicates DES for a KEK.
59 bits 2 and 3	**Value** **Description** **B'00'** Indicates single-length key (version 0 only). **B'01'** Indicates double-length key (version 1 only). **B'10'** Indicates triple-length key (version 1 only).
59 bits 4 - 7	B'0000'
60 - 63	Token validation value (TVV).

Note: AKEKs are not supported by this version of CCA. Key tokens from other CCA systems, however, could have the AKEK flag bits set in a key token.

DES external key token

The format for a DES external key token.

Table 210 shows the format for a DES external key token.

Table 210. DES external key token format

Bytes	Description
0	X'02' (flag indicating an external key token)
1	Reserved (X'00')
2 - 3	Implementation-dependent bytes (X'0000' for CCA)
4	Key token version number (X'00' or X'01')
5	Reserved (X'00')
6	Flag byte **Bit** **Meaning When Set On** **0** Encrypted key is present. **1** Control vector (CV) value has been applied to the key. Other bits are reserved and are binary zeros.
7	Reserved (X'00')
8 - 15	Reserved (X'0000000000000000')
16 - 23	Single-length key or left half of a double-length key, or Part A of a triple-length key. The value is encrypted under a transport key.
24 - 31	X'0000000000000000' if a single-length key or right half of a double-length key, or Part B of a triple-length key. The right half of a double-length key or Part B of a triple-length key is encrypted under a transport (key-encrypting key) for export or import.
32 - 39	Control vector (CV) for single-length key or left half of CV for double-length key
40 - 47	X'0000000000000000' if single-length key or right half of CV for double-length key
48 - 55	X'0000000000000000' if a single-length key, double-length key, or Part C of a triple-length key.
56 - 58	Reserved (X'000000')
59 bits 0 and 1	B'00'
59 bits 2 and 3	**Value** **Description** **B'00'** Indicates single-length key (version 0 only). **B'01'** Indicates double-length key (version 1 only). **B'10'** Indicates triple-length key (version 1 only).
59 bits 4 - 7	B'0000'
60 - 63	Token validation value (see “Token Validation Value” on page 770 for a description).

External RKX DES key tokens

The Remote Key Export (CSNDRKX) verb and DES key-storage verbs use a special RKX key token.

Table 211 on page 774 defines an external fixed-length DES key-token called an *RKX key-token*. An RKX key-token is a special token used exclusively by the Remote Key Export (CSNDRKX) verb and DES key-storage verbs (for example, DES Key Record Write). No other verbs use or reference an RKX key-token or key-token record. For additional information about the usage of RKX key tokens,

see “Remote key loading” on page 46. Verbs other than Remote Key Export and the DES key-storage do not support RKX key tokens or RKX key token records.

As can be seen in the table, RKX key tokens are 64 bytes in length, have a token identifier flag (X'02'), a token version number (X'10'), and room for encrypted keys, same as normal fixed-length DES key tokens. Unlike normal fixed-length DES key tokens, RKX key tokens do not have a control vector, flag bits, and a token-validation value. In addition, RKX key tokens have a confounder value, a MAC value, and room for a third encrypted key.

Table 211. External RKX DES key-token format, version X'10'

Offset	Length	Description
00	1	X'02' (a token identifier flag that indicates an external key-token)
01	3	Reserved, binary zero
04	1	Token version number (X'10')
05	2	Reserved, binary zero
07	1	Key length in bytes, including confounder
08	8	Confounder
16	8	Key left
24	8	Key middle (binary zero if not used)
32	8	Key right (binary zero if not used)
40	8	Rule ID The trusted block rule identifier used to create this key token. A subsequent call to Remote Key Export (CSNDRKX) can use this token with a trusted block rule that references the rule ID that was used to create this token. The trusted block rule can be compared with this rule ID for verification purposes. The Rule ID is an 8-byte string of ASCII characters, left-aligned and padded on the right with space characters. Acceptable characters are A...Z, a...z, 0...9, - (X'2D'), and _ (X'5F'). All other characters are reserved for future use.
48	8	Reserved, binary zero
56	8	MAC value ISO 16609 CBC-mode Triple-DES MAC, computed over the 56 bytes starting at offset 0 and including the encrypted key value and the rule ID using the same MAC key that is used to protect the trusted block itself. This MAC value guarantees that the key and the rule ID cannot be modified without detection, providing integrity and binding the rule ID to the key itself. This MAC value must verify with the same trusted block used to create the key, thus binding the key structure to that specific trusted block.

Note:

1. A fixed, randomly derived variant is exclusive-ORed with the MAC key before it is used to encipher the generated or exported key and confounder.
2. The MAC key is located within a trusted block (internal format) and can be recovered by decipherment under a variant of the PKA master key.
3. The trusted block is originally created in external form by the Trusted Block Create verb, and then converted to internal form by the PKA Key Import verb prior to the Remote Key Export call.

DES null key token

The format for a DES null key token.

Table 212 shows the format for a DES null key token.

Table 212. DES null key token format

Bytes	Description
0	X'00' (flag indicating this is a null key token).
1 - 15	Reserved (set to binary zeros).
16 - 23	Single-length encrypted key, left half of double-length encrypted key, or Part A of triple-length encrypted key.
24 - 31	X'0000000000000000' if a single-length encrypted key, the right half of double-length encrypted key, or Part B of triple-length encrypted key.
32 - 39	X'0000000000000000' if a single-length encrypted key or double-length encrypted key.
40 - 47	Reserved (set to binary zeros).
48 - 55	Part C of a triple-length encrypted key.
56 - 63	Reserved (set to binary zeros).

RSA public key token

The sections of an RSA public key token.

An RSA public key token contains the following sections.

- A required token header, starting with the token identifier X'1E'
- A required RSA public key section, starting with the section identifier X'04'

Table 213 presents the format of an RSA public key token. All length fields are in binary. All binary fields (exponents, lengths, and so on) are stored with the high-order byte first (left, low-address, S/390® format).

Table 213. RSA Public Key Token format

Offset (decimal)	Length (bytes)	Description
Token Header **(Required)**		
000	001	Token identifier. X'1E' indicates an external token.
001	001	Version, X'00'.
002	002	Length of the key token structure.
004	004	Ignored. Should be 0.
RSA Public Key Section (Required)		
000	001	X'04', section identifier, RSA public key.
001	001	X'00', version.
002	002	Section length, 12 + xxx + yyy
004	002	Reserved field.
006	002	RSA public key exponent field length in bytes, “xxx”.
008	002	Public key modulus length in bits.
010	002	RSA public key modulus field length in bytes, “yyy”.

Table 213. RSA Public Key Token format (continued)

Offset (decimal)	Length (bytes)	Description
***Token Header* (Required)**		
012	xxx	Public key exponent (this is generally a 1, 3, or 64 - 256-byte quantity), named e. e must be odd and 1 < e < n. (Frequently, the value of e is $2^{16} + 1$). **Note:** You can import an RSA public key having an exponent valued to two (2). Such a public key can correctly validate an ISO 9796-1 digital signature. However, the current product implementation does not generate an RSA key with a public exponent valued to two (a Rabin key).
12 + xxx	yyy	Modulus, n.

RSA private external key token

The sections of an RSA private external key.

An RSA private external key token contains the following sections:

- A required PKA token header starting with the token identifier X'1E'
- A required RSA private key section starting with one of the section identifiers shown in Table 214.
- A required RSA public key section, starting with the section identifier X'04'
- An optional private key name section, starting with the section identifier X'10'

Table 214 presents the basic record format of an RSA private external key token. All length fields are in binary. All binary fields (exponents, lengths, and so on) are stored with the high-order byte first (left, low-address, S/390 format). All binary fields (exponents, modulus, and so on) in the private sections of tokens are right-aligned and padded with zeros to the left.

Table 214. RSA private external key token basic record format

Offset (decimal)	Length (bytes)	Description
Token Header (Required)		
000	001	Token identifier. X'1E' indicates an external token. The private key is either in cleartext or enciphered with a transport key-encrypting key.
001	001	Version, X'00'.
002	002	Length of the key token structure.
004	004	Ignored. Should be zero.
RSA Private Key Section (Required) See the following sections: • "RSA private key token, 1024-bit Modulus-Exponent external format" on page 777 • "RSA private key token, 4096-bit Modulus-Exponent external form" on page 778 • "RSA private key, 4096-bit Modulus-Exponent format with AES encrypted OPK section external form" on page 779 • "RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section external form" on page 781 • "RSA private key, 4096-bit Chinese Remainder Theorem external form" on page 783		
RSA Public Key Section (Required)		
000	001	X'04', section identifier, RSA public key.

Table 214. RSA private external key token basic record format (continued)

Offset (decimal)	Length (bytes)	Description
001	001	X'00', version.
002	002	Section length, 12 + xxx.
004	002	Reserved field.
006	002	RSA public key exponent field length in bytes, “xxx”.
008	002	Public key modulus length in bits.
010	002	RSA public key modulus field length in bytes, which is zero for a private token. **Note:** In an RSA private key token, this field should be zero. The RSA private key section contains the modulus.
012	xxx	Public key exponent, e (this is generally a 1, 3, or 64 - 256-byte quantity). e must be odd and 1 < e < n. (Frequently, the value of e is 2^{16} + 1 (= 65,537). **Note:** You can import an RSA public key having an exponent valued to two (2). Such a public key can correctly validate an ISO 9796-1 digital signature. However, the current product implementation does not generate an RSA key with a public exponent valued to two (a Rabin key).
Private Key Name (Optional)		
000	001	X'10', section identifier, private key name.
001	001	X'00', version.
002	002	Section length, X'0044' (68 decimal).
004	064	Private key name (in ASCII), left-aligned, padded with space characters (X'20'). An access control system can use the private key name to verify the calling application is entitled to use the key.

RSA private key token, 1024-bit Modulus-Exponent external format

This RSA private key token and the external X'02' token is supported starting with CEX3C.

Table 215 shows the format.

Table 215. RSA private key token, 1024-bit Modulus-Exponent external format

Offset (decimal)	Length (bytes)	Description
000	001	X'02', section identifier, RSA private key, Modulus-Exponent format (RSA-PRIV)
001	001	X'00', version.
002	002	Length of the RSA private key section X'016C' (364 decimal).
004	020	SHA-1 hash value of the private key subsection clear text, offset 28 to the section end. This hash value is checked after an enciphered private key is deciphered for use.
024	004	Reserved; set to binary zero.
028	001	Key format and security: **Value** **Description** **X'00'** Unencrypted RSA private key subsection identifier. **X'82'** Encrypted RSA private key subsection identifier.
029	001	Reserved, binary zero.
030	020	SHA-1 hash of the optional key-name section. If there is no key-name section, then 20 bytes of X'00'.

Table 215. RSA private key token, 1024-bit Modulus-Exponent external format (continued)

Offset (decimal)	Length (bytes)	Description
050	004	Key use flag bits. **B'11xx xxxx'** Only key unwrapping (KM-ONLY) **B'10xx xxxx'** Both signature generation and key unwrapping (KEY-MGMT) **B'01xx xxxx'** Undefined **B'00xx xxxx'** Only signature generation (SIG-ONLY) All other bits reserved, set to binary zero.
054	006	Reserved; set to binary zero.
060	024	Reserved; set to binary zero.
084	Start of the optionally-encrypted secure subsection.	
084	024	Random number, confounder.
108	128	Private-key exponent, d. $d = e^{-1} \bmod((p-1)(q-1))$, and $1 < d < n$ where e is the public exponent.
	End of the optionally-encrypted subsection; the confounder field and the private-key exponent field are enciphered for key confidentiality when the key format and security flags (offset 28) indicate the private key is enciphered. They are enciphered under a double-length transport key using the ede2 algorithm.	
236	128	Modulus, n. n = pq where p and q are prime and $1 < n < 2^{1024}$.

RSA private key token, 4096-bit Modulus-Exponent external form

This RSA private key token is supported on CCA Crypto Express coprocessors.

This RSA private key token and the external X'09' token is supported on a CCA Crypto Express coprocessor.

Table 216. RSA Private Key Token, 4096-bit Modulus-Exponent External Format

Offset (decimal)	Number of bytes	Description
000	001	X'09', section identifier, RSA private key, modulus-exponent format (RSAMEVAR).
001	001	X'00', version.
002	002	Length of the RSA private key section 132+ddd+nnn+xxx.
004	020	SHA-1 hash value of the private key subsection cleartext, offset 28 to the section end. This hash value is checked after an enciphered private key is deciphered for use.
024	002	Length of the encrypted private key section 8+ddd+xxx.
026	002	Reserved; set to binary zero.
028	001	Key format and security: **X'00'** Unencrypted RSA private key subsection identifier. **X'82'** Encrypted RSA private key subsection identifier.
029	001	Reserved, set to binary zero.
030	020	SHA-1 hash of the optional key-name section. If there is no key-name section, then 20 bytes of X'00'.

Table 216. RSA Private Key Token, 4096-bit Modulus-Exponent External Format (continued)

Offset (decimal)	Number of bytes	Description
050	001	Key use flag bits. **Bit** **Meaning When Set On** **0** Key management usage permitted. **1** Signature usage not permitted. **6** The key is translatable All other bits reserved, set to binary zero.
051	001	Reserved; set to binary zero.
052	048	Reserved; set to binary zero.
100	016	Reserved; set to binary zero.
116	002	Length of private exponent, d, in bytes: ddd.
118	002	Length of modulus, n, in bytes: nnn.
120	002	Length of padding field, in bytes: xxx.
122	002	Reserved; set to binary zero.
		Start of the optionally-encrypted secure subsection.
124	008	Random number, confounder.
132	ddd	Private-key exponent, $d=e^{-1} \bmod((p-1)(q-1))$, and $1<d<n$ where e is the public exponent.
132+ddd	xxx	X'00' padding of length xxx bytes such that the length from the start of the random number above to the end of the padding field is a multiple of eight bytes.
		End of the optionally-encrypted subsection; the confounder field and the private-key exponent field are enciphered for key confidentiality when the key format and security flags (offset 28) indicate that the private key is enciphered. They are enciphered under a double-length transport key using the ede2 algorithm.
132+ddd+xxx	nnn	Modulus, n. n=pq where p and q are prime and $1<n<2^{4096}$.

RSA private key, 4096-bit Modulus-Exponent format with AES encrypted OPK section external form

This RSA private key token is supported on CCA Crypto Express coprocessors.

Table 217. RSA private key, 4096-bit Modulus-Exponent format with AES encrypted OPK section (X'30') external form

Offset (bytes)	Length (bytes)	Description
000	001	Section identifier: **X'30'** RSA private key, ME format with AES encrypted OPK.
001	001	Section version number (X'00').
002	002	Section length: 122 + nnn + ppp
004	002	Length of Associated Data section
006	002	Length of payload data: ppp
008	002	Reserved, binary zero.
		Start of Associated Data
010	001	Associated Data Version: **X'02'** Version 2

Table 217. RSA private key, 4096-bit Modulus-Exponent format with AES encrypted OPK section (X'30') external form (continued)

Offset (bytes)	Length (bytes)	Description
011	001	Key format and security flag: **X'00'** Unencrypted ME RSA private-key subsection identifier **X'82'** Encrypted ME RSA private-key subsection identifier
012	001	Key source flag: Reserved, binary zero.
013	001	Reserved, binary zeroes.
014	001	Hash type: **X'00'** Clear key **X'02'** SHA-256
015	032	SHA-256 hash of all optional sections that follow the public key section, if any; else 32 bytes of X'00'.
047	003	Reserved, binary zero.
050	001	Key-usage flag: **B'11xx xxxx'** Only key unwrapping (KM-ONLY) **B'10xx xxxx'** Both signature generation and key unwrapping (KEY-MGMT) **B'01xx xxxx'** Undefined **B'00xx xxxx'** Only signature generation (SIG-ONLY) Translation control: **B'xxxx xx1x'** Private key translation is allowed (XLATE-OK) **B'xxxx xx0x'** Private key translation is not allowed (NO-XLATE)
051	001	Reserved, binary zero.
052	002	Length of modulus: nnn bytes
054	002	Length of private exponent: ddd bytes
		End of Associated Data
056	048	16 byte confounder + 32-byte Object Protection Key. OPK used as an AES key. encrypted with an AES KEK.
104	016	Key verification pattern • For an encrypted private key, KEK verification pattern (KVP) • For a clear private key, binary zeros • For a skeleton, binary zeros
120	002	Reserved, binary zeros.
122	nnn	Modulus

Table 217. RSA private key, 4096-bit Modulus-Exponent format with AES encrypted OPK section (X'30') external form (continued)

Offset (bytes)	Length (bytes)	Description
122+nnn	ppp	Payload starts here and includes: When this section is unencrypted: • Clear private exponent d. • Length ppp bytes : ddd + 0 When this section is encrypted: • Private exponent d within the AESKW-wrapped payload. • Length ppp bytes : ddd + AESKW format overhead

RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section external form

This RSA private key token is supported on CCA Crypto Express Coprocessors.

Table 218. RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section (X'31') external form

Offset (bytes)	Length (bytes)	Description
000	001	Section identifier: **X'31'** RSA private key, CRT format with AES encrypted OPK
001	001	Section version number (X'00').
002	002	Section length: 134 + nnn + xxx
004	002	Length of Associated Data section
006	002	Length of payload data: xxx
008	002	Reserved, binary zero.
		Start of Associated Data
010	001	Associated Data Version: **X'03'** Version 3
011	001	Key format and security flag: **X'40'** Unencrypted RSA private-key subsection identifier **X'42'** Encrypted RSA private-key subsection identifier
012	001	Key source flag: Reserved, binary zero.
013	001	Reserved, binary zeroes.
014	001	Hash type: **X'00'** Clear key **X'01'** SHA-256
015	032	SHA-256 hash of all optional sections that follow the public key section, if any; else 32 bytes of X'00'.
047	003	Reserved, binary zero.

Table 218. RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section (X'31') external form (continued)

Offset (bytes)	Length (bytes)	Description
050	001	Key-usage flag: **B'11xx xxxx'** Only key unwrapping (KM-ONLY) **B'10xx xxxx'** Both signature generation and key unwrapping (KEY-MGMT) **B'01xx xxxx'** Undefined **B'00xx xxxx'** Only signature generation (SIG-ONLY) Translation control: **B'xxxx xx1x'** Private key translation is allowed (XLATE-OK) **B'xxxx xx0x'** Private key translation is not allowed (NO-XLATE)
051	001	Reserved, binary zero.
052	002	Length of the prime number, p, in bytes: ppp.
054	002	Length of the prime number, q, in bytes: qqq
056	002	Length of dp : rrr.
058	002	Length of dq : sss.
060	002	Length of U: uuu.
062	002	Length of modulus, nnn.
064	002	Reserved, binary zero.
066	002	Reserved, binary zero.
		End of Associated Data
068	048	16 byte confounder + 32-byte Object Protection Key. OPK used as an AES key. External tokens: encrypted with an AES KEK. Internal tokens: encrypted with the ECC master key.
116	016	Key verification pattern • For an encrypted private key, KEK verification pattern (KVP) • For a clear private key, binary zeros • For a skeleton, binary zeros
132	002	Reserved, binary zeros
134	nnn	Modulus, n, n=pq, where p and q are prime.

Table 218. RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section (X'31') external form (continued)

Offset (bytes)	Length (bytes)	Description
134+nnn	xxx	Payload starts here and includes: When this section is unencrypted: • Clear prime number p • Clear prime number q • Clear dp • Clear dq • Clear U • Length xxx bytes: ppp + qqq + rrr + sss +uuu + 0 When this section is encrypted: • prime number p • prime number q • dp • dq • U • within the AESKW-wrapped payload. Length xxx bytes : ppp + qqq + rrr + sss +uuu + AESKW format overhead

RSA private key, 4096-bit Chinese Remainder Theorem external form

This RSA private key token with up to 2048-bit modulus is supported on all CCA Crypto Express coprocessors.

The modulus size is increased to 4096-bit on the z9 EC, z9 BC, z10 EC, z10 BC, or later machines with the Nov. 2007 or later version of the licensed internal code installed on the CCA Crypto Express coprocessor.

Table 219. RSA Private Key Token, 4096-bit Chinese Remainder Theorem External Format

Offset (decimal)	Number of bytes	Description
000	001	X'08', section identifier, RSA private key, CRT format (RSA-CRT)
001	001	X'00', version.
002	002	Length of the RSA private-key section, 132 + ppp + qqq + rrr + sss + uuu + xxx + nnn.
004	020	SHA-1 hash value of the private key subsection cleartext, offset 28 to the end of the modulus.
024	004	Reserved; set to binary zero.
028	001	Key format and security: **X'40'** Unencrypted RSA private-key subsection identifier, Chinese Remainder form. **X'42'** Encrypted RSA private-key subsection identifier, Chinese Remainder form.
029	001	Reserved; set to binary zero.

Table 219. RSA Private Key Token, 4096-bit Chinese Remainder Theorem External Format (continued)

Offset (decimal)	Number of bytes	Description
030	020	SHA-1 hash of the optional key-name section and any following optional sections. If there are no optional sections, then 20 bytes of X'00'.
050	004	Key use flag bits. **Bit** **Meaning When Set On** **0** Key management usage permitted. **1** Signature usage not permitted. **6** The key is translatable. All other bits reserved, set to binary zero.
054	002	Length of prime number, p, in bytes: ppp.
056	002	Length of prime number, q, in bytes: qqq.
058	002	Length of d_p, in bytes: rrr.
060	002	Length of d_q, in bytes: sss.
062	002	Length of U, in bytes: uuu.
064	002	Length of modulus, n, in bytes: nnn.
066	004	Reserved; set to binary zero.
070	002	Length of padding field, in bytes: xxx.
072	004	Reserved, set to binary zero.
076	016	Reserved, set to binary zero.
092	032	Reserved; set to binary zero.
124	Start of the optionally-encrypted secure subsection.	
124	008	Random number, confounder.
132	ppp	Prime number, p.
132 + ppp	qqq	Prime number, q
132 + ppp + qqq	rrr	d_p = d mod(p - 1)
132 + ppp + qqq + rrr	sss	d_q = d mod(q - 1)
132 + ppp + qqq + rrr + sss	uuu	U = q**-1 mod(p).
132 + ppp + qqq + rrr + sss + uuu	xxx	X'00' padding of length xxx bytes such that the length from the start of the random number above to the end of the padding field is a multiple of eight bytes.
	End of the optionally-encrypted secure subsection; all of the fields starting with the confounder field and ending with the variable length pad field are enciphered for key confidentiality when the key format-and-security flags (offset 28) indicate that the private key is enciphered. They are enciphered under a double-length transport key using the TDES (CBC outer chaining) algorithm.	
132 + ppp + qqq + rrr + sss + uuu + xxx	nnn	Modulus, n. n = pq where p and q are prime and 1**<n<2**2048.

RSA private internal key token

The sections of the RSA private internal key.

An RSA private internal key token contains the following sections:

- A required PKA token header, starting with the token identifier X'1F'
- Basic record format of an RSA private internal key token. All length fields are in binary. All binary fields (exponents, lengths, and so on) are stored with the high-order byte first (left, low-address, S/390 format). All binary fields (exponents, modulus, and so on) in the private sections of tokens are right-aligned and padded with zeros to the left.

Table 220 shows the format.

Table 220. RSA private internal key token basic record format

Offset (decimal)	Length (bytes)	Description
Token Header (Required)		
000	001	Token identifier. X'1F' indicates an internal token. The private key is enciphered with a PKA master key.
001	001	Version, X'00'.
002	002	Length of the key token structure excluding the internal information section.
004	004	Ignored; should be zero.
RSA Private Key Section and Secured Subsection (Required) See the following sections: • "RSA private key token, 1024-bit Modulus-Exponent internal format for cryptographic coprocessor feature" on page 786. • "RSA private key token, 1024-bit Modulus-Exponent internal format" on page 787. • "RSA private key token, 4096-bit Modulus-Exponent internal format" on page 788. • "RSA private key, 4096-bit Modulus-Exponent format with AES encrypted OPK section internal form" on page 790 • "RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section internal form" on page 791 • "RSA private key, 4096-bit Chinese Remainder Theorem internal form" on page 793		
RSA Public Key Section (Required)		
000	001	X'04', section identifier, RSA public key.
001	001	X'00', version.
002	002	Section length, 12 + xxx.
004	002	Reserved field.
006	002	RSA public key exponent field length in bytes, "xxx".
008	002	Public key modulus length in bits.
010	002	RSA public key modulus field length in bytes, which is zero for a private token.
012	xxx	Public key exponent (this is generally a 1, 3, or 64 - 256-byte quantity), e. e must be odd and $1 < e < n$. (Frequently, the value of e is $2^{16} + 1$ (= 65,537). **Note:** You can import an RSA public key having an exponent valued to two (2). Such a public key can correctly validate an ISO 9796-1 digital signature. However, the current product implementation does not generate an RSA key with a public exponent valued to two (a Rabin key).
Private Key Name (Optional)		

Table 220. RSA private internal key token basic record format (continued)

Offset (decimal)	Length (bytes)	Description
000	001	X'10', section identifier, private key name.
001	001	X'00', version.
002	002	Section length, X'0044' (68 decimal).
004	064	Private key name (in ASCII), left-aligned, padded with space characters (X'20'). An access control system can use the private key name to verify the calling application is entitled to use the key.
Internal Information Section (Required)		
000	004	Eye catcher 'PKTN'.
004	004	PKA token type. **Bit** **Meaning When Set On** **0** RSA key. **2** Private key. **3** Public key. **4** Private key name section exists. **5** Private key unenciphered. **6** Blinding information present. **7** Retained private key.
008	004	Address of token header.
012	002	Total length of total structure including this information section.
014	002	Count of number of sections.
016	016	PKA master key hash pattern.
032	001	Domain of retained key.
033	008	Serial number of processor holding retained key.
041	007	Reserved.

RSA private key token, 1024-bit Modulus-Exponent internal format for cryptographic coprocessor feature

The format of the RSA private key token, 1024-bit Modulus-Exponent internal format for cryptographic coprocessor feature.

Table 221 shows the format of the RSA private key token, 1024-bit Modulus-Exponent internal format for cryptographic coprocessor feature.

Table 221. RSA private internal key token, 1024-bit Modulus-Exponent format for cryptographic coprocessor feature

Offset (decimal)	Length (bytes)	Description
000	001	X'02', section identifier, RSA private key.
001	001	X'00', version.
002	002	Length of the RSA private key section X'016C' (364 decimal).
004	020	SHA-1 hash value of the private key subsection cleartext, offset 28 to the section end. This hash value is checked after an enciphered private key is deciphered for use.
024	004	Reserved; set to binary zero.
028	001	Key format and security: **X'02'** RSA private key.

Table 221. RSA private internal key token, 1024-bit Modulus-Exponent format for cryptographic coprocessor feature (continued)

Offset (decimal)	Length (bytes)	Description
029	001	Format of external key from which this token was derived: **Value** **Description** **X'21'** External private key was specified in the clear. **X'22'** External private key was encrypted.
030	020	SHA-1 hash of the key token structure contents that follow the public key section. If no sections follow, this field is set to binary zeros.
050	001	Key use flag bits. **B'11xx xxxx'** Only key unwrapping (KM-ONLY) **B'10xx xxxx'** Both signature generation and key unwrapping (KEY-MGMT) **B'01xx xxxx'** Undefined **B'00xx xxxx'** Only signature generation (SIG-ONLY) All other bits reserved, set to binary zero.
051	009	Reserved; set to binary zero.
060	048	Object Protection Key (OPK) encrypted under a PKA master key—can be under the Signature Master Key (SMK) or Key Management Master Key (KMMK) depending on key use.
108	128	Secret key exponent d, encrypted under the OPK. $d = e^{-1} \bmod((p-1)(q-1))$
236	128	Modulus, n. $n = pq$ where p and q are prime and $1 < n < 2^{1024}$.

RSA private key token, 1024-bit Modulus-Exponent internal format

The format for the RSA private key token, 1024-bit Modulus-Exponent internal format starting with CEX3C.

Table 222 shows the format for the RSA private key token, 1024-bit Modulus-Exponent internal format starting with CEX3C.

Table 222. RSA private internal key token, 1024-bit Modulus-Exponent starting with CEX3C

Offset (decimal)	Length (bytes)	Description
000	001	X'06', section identifier, RSA private key Modulus-Exponent format (RSA-PRIV).
001	001	X'00', version.
002	002	Length of the RSA private key section X'0198' (408 decimal) + rrr + iii + xxx.
004	020	SHA-1 hash value of the private key subsection cleartext, offset 28 to and including the modulus at offset 236.
024	004	Reserved; set to binary zero.
028	001	Key format and security: **X'02'** RSA private key.

Table 222. RSA private internal key token, 1024-bit Modulus-Exponent starting with CEX3C (continued)

Offset (decimal)	Length (bytes)	Description
029	001	Format of external key from which this token was derived: **X'21'** External private key was specified in the clear. **X'22'** External private key was encrypted. **X'23'** Private key was generated using regeneration data. **X'24'** Private key was randomly generated.
030	020	SHA-1 hash of the optional key-name section and any following optional sections. If there are no optional sections, this field is set to binary zeros.
050	004	Key use flag bits. **B'11xx xxxx'** Only key unwrapping (KM-ONLY) **B'10xx xxxx'** Both signature generation and key unwrapping (KEY-MGMT) **B'01xx xxxx'** Undefined **B'00xx xxxx'** Only signature generation (SIG-ONLY) All other bits reserved, set to binary zero.
054	006	Reserved; set to binary zero.
060	048	Object Protection Key (OPK) encrypted under the Asymmetric Keys Master Key using the ede3 algorithm.
108	128	Private key exponent d, encrypted under the OPK using the ede5 algorithm. $d = e^{-1} \bmod((p-1)(q-1))$, and $1 < d < n$ where e is the public exponent.
236	128	Modulus, n. $n = pq$ where p and q are prime and $2^{512} < n < 2^{1024}$.
364	016	Asymmetric-Keys Master Key hash pattern.
380	020	SHA-1 hash value of the blinding information subsection cleartext, offset 400 to the end of the section.
400	002	Length of the random number r, in bytes: *rrr*
402	002	Length of the random number r^{-1}, in bytes: *iii*
404	002	Length of the padding field, in bytes: *xxx*
406	002	Reserved; set to binary zeros.
408	Start of the encrypted blinding subsection	
408	*rrr*	Random number r (used in blinding).
408 + *rrr*	*iii*	Random number r^{-1} (used in blinding).
408 + *rrr* + *iii*	*xxx*	X'00' padding of length *xxx* bytes such that the length from the start of the encrypted blinding subsection to the end of the padding field is a multiple of eight bytes.
	End of the encrypted blinding subsection; all of the fields starting with the random number r and ending with the variable length pad field are encrypted under the OPK using TDES (CBC outer chaining) algorithm.	

RSA private key token, 4096-bit Modulus-Exponent internal format

The format of an RSA private key, 4096-bit Modulus-Exponent token.

Table 223 on page 789 shows the format of an RSA private key, 4096-bit Modulus-Exponent token.

Table 223. Private key, 4096-bit Modulus-Exponent format section (X'09')

Offset (bytes)	Length (bytes)	Description
000	001	Section identifier: **X'09'** RSA private key, 4096-bit maximum Modulus-Exponent format (RSAMEVAR). This format is used for a clear or an encrypted RSA private-key in an external key-token up to a modulus size of 4096 bits. This section type is not created in releases before Release 4.1.
001	001	Section version number (X'00').
002	002	Section length (132 + *ddd* + *nnn* + *xxx*).
004	020	SHA-1 hash value of the private-key subsection cleartext, offset 28 to the end of the modulus.
024	002	Length in bytes of the optionally encrypted secure subsection, or X'0000' if the subsection is not encrypted.
026	002	Reserved (X'0000').
028	001	Key format and security flags: External token: **X'00'** Unencrypted RSA private-key subsection identifier **X'82'** Encrypted RSA private-key subsection identifier
029	001	Private key source flag: **X'00'** Generation method unknown
030	020	SHA-1 hash of all optional sections that follow the public-key section, if any, else 20 bytes of X'00'.
050	001	Key-usage flag: **B'11xx xxxx'** Only key unwrapping (KM-ONLY) **B'10xx xxxx'** Both signature generation and key unwrapping (KEY-MGMT) **B'01xx xxxx'** Undefined **B'00xx xxxx'** Only signature generation (SIG-ONLY) All other bits are reserved and must be zero.
051	065	Reserved, binary zeros.
116	002	Private-key exponent field length, in bytes: *ddd*.
118	002	Private-key modulus field length, in bytes: *nnn*.
120	002	Length of padding field, in bytes: *xxx*. Padding of X'00' bytes for a length of *xxx* bytes such that the length from the start of the confounder at offset 124 to the end of the padding field is a multiple of 8 bytes.
122	002	Reserved, binary zeros.
Start of the (optionally) encrypted subsection; all of the fields starting with the confounder field and ending with the variable-length pad field are enciphered for key confidentiality when the key format and security flags (offset 28) indicate that the private key is enciphered.		
124	008	Confounder. This is an eight-byte random number. Data encrypted with two-part key-encrypting key.

Table 223. Private key, 4096-bit Modulus-Exponent format section (X'09') (continued)

Offset (bytes)	Length (bytes)	Description
132	*ddd*	Private-key exponent, *d*: $d = e^{-1} \bmod((p - 1)(q - 1))$ where $1 < d < n$, and *e* is the public exponent. The transport key encrypts the private key exponent using the EDE2 algorithm.
132 + *ddd*	*xxx*	Pad of X'00' bytes.
End of the optionally encrypted subsection.		
132 + *ddd* + *xxx*	*nnn*	Private-key modulus.

RSA private key, 4096-bit Modulus-Exponent format with AES encrypted OPK section internal form

This RSA private key token is supported on CCA Crypto Express coprocessors.

Table 224. RSA private key, 4096-bit Modulus-Exponent format with AES encrypted OPK section (X'30') internal form

Offset (bytes)	Length (bytes)	Description
000	001	Section identifier: **X'30'** RSA private key, ME format with AES encrypted OPK.
001	001	Section version number (X'00').
002	002	Section length: 122 + nnn + ppp
004	002	Length of Associated Data section
006	002	Length of payload data: ppp
008	002	Reserved, binary zero.
		Start of Associated Data
010	001	Associated Data Version: **X'02'** Version 2
011	001	Key format and security flag: **X'02'** Encrypted ME RSA private-key subsection identifier
012	001	Key source flag: Internal tokens: **X'21'** Imported from clear text **X'22'** Imported from cipher text **X'23'** Generated using regeneration data **X'24'** Randomly generated
013	001	Reserved, binary zeroes.
014	001	Hash type: **X'00'** Clear key **X'02'** SHA-256
015	032	SHA-256 hash of all optional sections that follow the public key section, if any; else 32 bytes of X'00'.
047	003	Reserved, binary zero.

Table 224. RSA private key, 4096-bit Modulus-Exponent format with AES encrypted OPK section (X'30') internal form (continued)

Offset (bytes)	Length (bytes)	Description
050	001	Key-usage flag: **B'11xx xxxx'** Only key unwrapping (KM-ONLY) **B'10xx xxxx'** Both signature generation and key unwrapping (KEY-MGMT) **B'01xx xxxx'** Undefined **B'00xx xxxx'** Only signature generation (SIG-ONLY) Translation control: **B'xxxx xx1x'** Private key translation is allowed (XLATE-OK) **B'xxxx xx0x'** Private key translation is not allowed (NO-XLATE)
051	001	Reserved, binary zero.
052	002	Length of modulus: nnn bytes
054	002	Length of private exponent: ddd bytes
		End of Associated Data
056	048	16 byte confounder + 32-byte Object Protection Key. OPK used as an AES key. encrypted with the ECC master key.
104	016	Key verification pattern • For an encrypted private key, ECC master-key verification pattern (MKVP) • For a skeleton, binary zeros
120	002	Reserved, binary zeros.
122	nnn	Modulus
122+nnn	ppp	Payload starts here and includes: When this section is unencrypted: • Clear private exponent d. • Length ppp bytes : ddd + 0 When this section is encrypted: • Private exponent d within the AESKW-wrapped payload. • Length ppp bytes : ddd + AESKW format overhead

RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section internal form

The format of an RSA private key token, 4096-bit Chinese Remainder Theorem, with AES encrypted OPK section (X'31'), in internal format.

Table 225. RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section (X'31') internal form

Offset (bytes)	Length (bytes)	Description
000	001	Section identifier: **X'31'** RSA private key, CRT format with AES encrypted OPK
001	001	Section version number (X'00').
002	002	Section length: 134 + nnn + xxx
004	002	Length of Associated Data section
006	002	Length of payload data: xxx
008	002	Reserved, binary zero.
		Start of Associated Data
010	001	Associated Data Version: **X'03'** Version 3
011	001	Key format and security flag: **X'08'** Unencrypted RSA private-key subsection identifier
012	001	Key source flag: **X'21'** Imported from cleartext **X'22'** Imported from ciphertext **X'23'** Generated using regeneration data **X'24'** Randomly generated
013	001	Reserved, binary zeroes.
014	001	Hash type: **X'00'** Clear key **X'01'** SHA-256
015	032	SHA-256 hash of all optional sections that follow the public key section, if any; else 32 bytes of X'00'.
047	003	Reserved, binary zero.
050	001	Key-usage flag: **B'11xx xxxx'** Only key unwrapping (KM-ONLY) **B'10xx xxxx'** Both signature generation and key unwrapping (KEY-MGMT) **B'01xx xxxx'** Undefined **B'00xx xxxx'** Only signature generation (SIG-ONLY) Translation control: **B'xxxx xx1x'** Private key translation is allowed (XLATE-OK) **B'xxxx xx0x'** Private key translation is not allowed (NO-XLATE)
051	001	Reserved, binary zero.
052	002	Length of the prime number, p, in bytes: ppp.
054	002	Length of the prime number, q, in bytes: qqq

Table 225. RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section (X'31') internal form (continued)

Offset (bytes)	Length (bytes)	Description
056	002	Length of dp : rrr.
058	002	Length of dq : sss.
060	002	Length of U: uuu.
062	002	Length of modulus, nnn.
064	002	Reserved, binary zero.
066	002	Reserved, binary zero.
		End of Associated Data
068	048	16 byte confounder + 32-byte Object Protection Key. OPK used as an AES key. encrypted with the ECC master key.
116	016	Key verification pattern • For an encrypted private key, ECC master-key verification pattern (MKVP) • For a skeleton, binary zeros
132	002	Reserved, binary zeros
134	nnn	Modulus, n, n=pq, where p and q are prime.
134+nnn	xxx	Payload starts here and includes: When this section is unencrypted: • Clear prime number p • Clear prime number q • Clear dp • Clear dq • Clear U • Length xxx bytes: ppp + qqq + rrr + sss +uuu + 0 When this section is encrypted: • prime number p • prime number q • dp • dq • U • within the AESKW-wrapped payload. Length xxx bytes : ppp + qqq + rrr + sss +uuu + AESKW format overhead

RSA private key, 4096-bit Chinese Remainder Theorem internal form

This RSA private key token with up to 2048-bit modulus is supported on all CCA coprocessors.

The modulus size is increased to 4096-bit on the z9 EC, z9 BC, z10 EC, z10 BC, or later machines with the Nov. 2007 or later version of the licensed internal code installed on the CCA Crypto Express coprocessor.

Table 226. RSA Private Internal Key Token, 4096-bit Chinese Remainder Theorem Internal Format

Offset (decimal)	Number of bytes	Description
000	001	X'08', section identifier, RSA private key, CRT format (RSA-CRT)
001	001	X'00', version.
002	002	Length of the RSA private-key section, 132 + ppp + qqq + rrr + sss + uuu + ttt + iii + xxx + nnn.
004	020	SHA-1 hash value of the private-key subsection cleartext, offset 28 to the end of the modulus.
024	004	Reserved; set to binary zero.
028	001	Key format and security: **X'08'** Encrypted RSA private-key subsection identifier, Chinese Remainder form.
029	001	Key derivation method: **X'21'** External private key was specified in the clear. **X'22'** External private key was encrypted. **X'23'** Private key was generated using regeneration data. **X'24'** Private key was randomly generated.
030	020	SHA-1 hash of the optional key-name section and any following sections. If there are no optional sections, then 20 bytes of X'00'.
050	004	Key use flag bits: **Bit** **Meaning When Set On** **0** Key management usage permitted. **1** Signature usage not permitted. All other bits reserved, set to binary zero.
054	002	Length of prime number, p, in bytes: ppp.
056	002	Length of prime number, q, in bytes: qqq.
058	002	Length of d_p, in bytes: rrr.
060	002	Length of d_q, in bytes: sss.
062	002	Length of U, in bytes: uuu.
064	002	Length of modulus, n, in bytes: nnn.
066	002	Length of the random number r, in bytes: ttt.
068	002	Length of the random number r**-1, in bytes: iii.
070	002	Length of padding field, in bytes: xxx.
072	004	Reserved, set to binary zero.
076	016	RSA Master Key hash pattern.
092	032	Object Protection Key (OPK) encrypted under the RSA Master Key using the TDES (CBC outer chaining) algorithm.
124	Start of the encrypted secure subsection, encrypted under the OPK using TDES (CBC outer chaining).	
124	008	Random number, confounder.
132	ppp	Prime number, p.
132 + ppp	qqq	Prime number, q

Table 226. RSA Private Internal Key Token, 4096-bit Chinese Remainder Theorem Internal Format (continued)

Offset (decimal)	Number of bytes	Description
132 + ppp + qqq	rrr	d_p = d mod(p - 1)
132 + ppp + qqq + rrr	sss	d_q = d mod(q - 1)
132 + ppp + qqq + rrr + sss	uuu	U = $^{-1}$mod(p). q**-1 mod(p).
132 + ppp + qqq + rrr + sss + uuu	ttt	Random number r (used in blinding).
132 + ppp + qqq + rrr + sss + uuu + ttt	iii	Random number $^{-1}$ r**-1 (used in blinding).
132 + ppp + qqq + rrr + sss + uuu + ttt + iii	xxx	X'00' padding of length xxx bytes such that the length from the start of the confounder at offset 124 to the end of the padding field is a multiple of eight bytes.
	End of the encrypted secure subsection; all of the fields starting with the confounder field and ending with the variable length pad field are encrypted under the OPK using TDES (CBC outer chaining) for key confidentiality.	
132 + ppp + qqq + rrr + sss + uuu + ttt + iii + xxx	nnn	Modulus, n. n = pq where p and q are prime and 1<n<2**2048.

RSA variable Modulus-Exponent token

A description of the fields in the new variable length Modulus-Exponent token. RSA variable Modulus-Exponent token.

Table 227 describes the fields in the new variable length Modulus-Exponent token. Currently, only the external form of the token will be used. There are no blinding values for the token. The latest level hardware makes this unnecessary.

Table 227. RSA variable Modulus-Exponent token format

Number	If External Key	New version '09' field	If Internal Key	Length in bytes
1	'09'	sectionId	'09'	1
2	'00'	version	'00'	1
3	132 + dLength + nLength + padLength	sectionLength	132 + dLength + nLength + padLength	2
4	Hash over fields 7 - end of section (clear values)	sha1Hash	Hash over fields 7 - end of section	20
5	8 + dLength + padLength	encrypted sectionLength	8 + dLength + padLength	2
6	This is actually a reserved field, not a pad '0000'	pad	'0000'	2
7	'82' encrypted external key or '00' clear external key	keyFormat	'02' encrypted operational key	1
8	'00'	pedigree	'21', '22', '23', or '24' as '06' token	1
9	Hash over sections which follow the public key section, or '00'	sha1Key NameHash	Hash over sections which follow the public key section, or '00'	20
10	'02' indicates that the key is translatable	keyUsageFlag	same as in '06'	1
11	'00'	reserved1	'00'	1

Table 227. RSA variable Modulus-Exponent token format (continued)

Number	If External Key	New version '09' field	If Internal Key	Length in bytes
12	Binary zeroes	OPK	8 byte confounder + 40-byte (5-part) DES key, encrypted with the PKA master key	48
13	Binary zeroes	mkHash Pattern	16 byte MKVP	16
14	Length of private exponent	dLength	Length of private exponent	2
15	Length of modulus	nLength	Length of modulus	2
16	Length required to pad dLength to a multiple of 8	padLength	Length required to pad dLength to a multiple of 8	2
17	'0000'	reserved2	'0000'	2
18	Random value - encrypted data (with PKA MK) begins here	confounder	encrypted data (with 5-part OPK) begins here	8
19		<d follows, then pad, then n>		1

ECC key token

The format of ECC public and private key tokens.

Table 231 on page 798 and Table 230 on page 797 show the format of ECC public and private key tokens.

CCA allows a choice between two types of elliptic curves when generating an ECC key. One is Brainpool, and the other is Prime. Table 228 and Table 229 show the size and name of each supported elliptic curve, along with its object identifier (OID) in dot notation.

Table 228. Supported Prime elliptic curves by size, name, and object identifier

Size of prime p in bits (key length)	OID in dot notation	ANSI X9.62 ECDSA prime curve ID	NIST-recommended elliptic curve ID	SEC 2 recommended elliptic curve domain parameter
192	1.2.840.10045.3.1.1	prime192v1	P-192	secp192r1
224	1.3.132.0.33	N/A	P-224	secp224r1
256	1.2.840.10045.3.1.7	prime256v1	P-256	secp256r1
384	1.3.132.0.34	N/A	P-384	secp384r1
521	1.3.132.0.35	N/A	P-521	secp521r1

Table 229. Supported Brainpool elliptic curves by size, name, and object identifier

Size of prime p in bits (key length)	OID in dot notation	Brainpool elliptic curve ID
160	1.3.36.3.3.2.8.1.1.1	brainpoolP160r1
192	1.3.36.3.3.2.8.1.1.3	brainpoolP192r1
224	1.3.36.3.3.2.8.1.1.5	brainpoolP224r1
256	1.3.36.3.3.2.8.1.1.7	brainpoolP256r1
320	1.3.36.3.3.2.8.1.1.9	brainpoolP320r1
384	1.3.36.3.3.2.8.1.1.11	brainpoolP384r1

Table 229. Supported Brainpool elliptic curves by size, name, and object identifier (continued)

Size of prime *p* in bits (key length)	OID in dot notation	Brainpool elliptic curve ID
512	1.3.36.3.3.2.8.1.1.13	brainpoolP512r1

Table 230. ECC private-key section

Offset (bytes)	Length (bytes)	Description
000	001	Section identifier: **X'20'** ECC private key
001	001	Section version number (X'00').
002	002	Length of section in bytes.
004	001	Wrapping method: **X'00'** Section is unencrypted (clear) **X'01'** AESKW **X'02'** CBC
005	001	Hash method used for wrapping: **X'00'** None (clear key) **X'01'** SHA-224 **X'02'** SHA-256
006	002	Reserved, binary zero.
008	001	Key-usage flag. Management of symmetric keys and generation of digital signatures: **B'11xx xxxx'** Only key establishment (KM-ONLY) **B'10xx xxxx'** Both signature generation and key establishment (KEY-MGMT) **B'01xx xxxx'** Undefined **B'00xx xxxx'** Only signature generation (SIG-ONLY) Translation control: **B'xxxx xx1x'** Private key translation is allowed (XLATE-OK) **B'xxxx xx0x'** Private key translation is not allowed (NO-XLATE)
009	001	Curve type: **X'00'** Prime **X'01'** Brainpool
010	001	Key format and security flag: **X'08'** Encrypted internal ECC private key **X'40'** Unencrypted external ECC private key **X'42'** Encrypted external ECC private key
011	001	Reserved, binary zero
012	002	Length of prime *p* in bits. See Table 228 on page 796 and Table 229 on page 796. **X'00A0'** 160 (Brainpool) **X'00C0'** 192 (Brainpool, Prime) **X'00E0'** 224 (Brainpool, Prime) **X'0100'** 256 (Brainpool, Prime) **X'0140'** 320 (Brainpool) **X'0180'** 384 (Brainpool, Prime) **X'0200'** 512 (Brainpool) **X'0209'** 521 (Prime)

Table 230. ECC private-key section (continued)

Offset (bytes)	Length (bytes)	Description
014	002	Length in bytes of IBM associated data.
016	008	Key verification pattern. External key-token • For an encrypted private key, KEK verification pattern (KVP) • For a clear private key, binary zeros • For a skeleton, binary zeros Internal key-token • For encrypted private key, master-key verification pattern (MKVP) • For a skeleton, binary zeros
024	048	Object Protection Key (OPK). For an internal key token: OPK, integrity check value (ICV), 8-byte confounder, and a 256-bit AES key used with the AESKW algorithm to encrypt the ECC private key, otherwise binary zeros. **Note:** The OPK is encrypted by the APKA master key using AESKW.
072	002	Length in bytes of associated data.
074	002	Length in bytes of formatted section, *bb*.
Associated data section		
Start of IBM associated data		
076	001	Associated data section version (X'00'). Includes IBM associated data and user-definable associated data.
077	001	Length in bytes of the key label: *kl* (0 - 64).
078	002	Length in bytes of the IBM associated data, including key label and IBM extended associated data (≥ 16).
080	002	Length in bytes of the IBM extended associated data (0).
082	001	Length in bytes of the user-definable associated data: *uad* (0 - 100).
083	001	Curve type (see offset 009).
084	002	Length of *p* in bits (see offset 012).
086	001	Key-usage flag (see offset 008).
087	001	Key format and security flag (see 010).
088	004	Reserved, binary zero.
092	*kl*	Optional key label.
092 + *kl*	*iead*	Optional IBM extended associated data.
End of IBM associated data		
092 + *kl* + *iead*	*uad*	Optional user-definable associated data.
End of associated data section		
092 + *kl* + *iead* + *uad*	*bb*	Formatted section (payload), which includes private key *d*: • Clear-key section contains *d*. • Encrypted-key section contains *d* within the AESKW-wrapped payload.

Table 231. ECC public-key section

Offset (bytes)	Length (bytes)	Description
000	001	Section identifier: **X'21'** ECC public key

Table 231. ECC public-key section (continued)

Offset (bytes)	Length (bytes)	Description
001	001	Section version number (X'00').
002	002	Section length.
004	004	Reserved, binary zero.
008	001	Curve type: **X'00'** Prime **X'01'** Brainpool
009	001	Reserved, binary zero
010	002	Length of prime *p* in bits. See Table 228 on page 796 and Table 229 on page 796. **X'00A0'** 160 (Brainpool) **X'00C0'** 192 (Brainpool, Prime) **X'00E0'** 224 (Brainpool, Prime) **X'0100'** 256 (Brainpool, Prime) **X'0140'** 320 (Brainpool) **X'0180'** 384 (Brainpool, Prime) **X'0200'** 512 (Brainpool) **X'0209'** 521 (Prime)
012	002	Length of public key *q* in bytes. Value includes key material length plus a one-byte flag to indicate if the key material is compressed.
014	*cc*	Public key *q*.

PKA null key token

The format for a PKA null key token.

Table 232 shows the format for a PKA null key token.

Table 232. PKA null key token format

Bytes	Description
0	X'00' Token identifier (indicates that this is a null key token).
1	Version, X'00'.
2 - 3	X'0008' Length of the key token structure.
4 - 7	Ignored (should be zero).

HMAC key token

The two formats of the HMAC key token.

HMAC key tokens have two formats, “HMAC MAC variable-length symmetric key token” on page 830 and “HMAC symmetric null key token.”

HMAC symmetric null key token

Table 233 shows the format of the HMAC symmetric null key token.

Table 233. HMAC symmetric null key token format

Offset (bytes))	Length (bytes)	Description
Header		

Table 233. HMAC symmetric null key token format (continued)

Offset (bytes))	Length (bytes)	Description
0	1	X'00' Token identifier, which indicates that this is a null key token.
1	1	X'00' Version
2 - 3	2	X'0008' Length of the key token structure.
4 - 7	4	Ignored (zero).

Variable-length symmetric key tokens

Beginning with Release 4.1, CCA supports a variable-length symmetric key-token. This key token has a version number of X'05' (offset 4). Use the Key Token Build2 (CSNBKTB2) verb to build skeleton variable-length symmetric key tokens used as input by the Key Generate2 (CSNBKGN2) or Key Part Import2 (CSNBKPI2) verbs, which return these key tokens with encrypted keys in the key-token payload.

Table 234 on page 801 shows the general format of the token version number X'05' key token.

Table 235 on page 814 shows the format of the CIPHER variable-length symmetric key-token that, beginning with Release 4.2, can be used with the AES algorithm. An AES CIPHER key-token is used by the Symmetric_Algorithm_Decipher (CSNBSAD) and Symmetric_Algorithm_Encipher (CSNBSAE) verbs to decipher or encipher data with the AES algorithm.

Table 236 on page 822 shows the format of the MAC variable-length symmetric key-token that, beginning with Release 4.4, can be used with the AES algorithm.

Table 237 on page 830 shows the format of the MAC variable-length symmetric key-token that, beginning with Release 4.1, can be used with the HMAC algorithm. An HMAC MAC key-token is used by the HMAC_Generate (CSNBHMG) and HMAC_Verify (CSNBHMV) verbs to generate or verify keyed hash message authentication codes.

Table 238 on page 837 shows the format of the EXPORTER and IMPORTER variable-length symmetric key tokens that can be used with the AES algorithm. An EXPORTER operational key-token is used by the Symmetric_Key_Export (CSNDSYX) verb to export an internal AES or HMAC variable-length symmetric key-token into an external variable-length symmetric key-token, either into an AESKW or PKOAEP2 wrapped payload. An IMPORTER operational key-token is used by the Symmetric_Key_Import2 (CSNDSYI2) verb to import an external AES or HMAC variable-length symmetric key-token, containing either an AESKW or PKOAEP2 wrapped payload, into an internal variable-length symmetric key-token.

Table 239 on page 847 shows the format of the PINPROT, PINCALC, and PINPRW variable-length symmetric key tokens that can be used with the AES algorithm.

Table 240 on page 856 shows the format of the DESUSECV variable-length symmetric key tokens that can be used with the AES algorithm.

"AES DKYGENKY variable-length symmetric key token" on page 860 shows the format of the DKYGENKY variable-length symmetric key tokens that can be used with the AES algorithm.

Table 242 on page 869 shows the format of the SECMSG variable-length symmetric key tokens that can be used with the AES algorithm.

General format of a variable-length symmetric key-token

View a table showing the general format of a variable-length symmetric key-token.

Table 234. General format of a variable-length symmetric key-token, version X'05'

Offset (bytes)	Length (bytes)	Description
Header		
000	01	Token identifier: **Value** **Meaning** **X'00'** Null key-token. **X'01'** Internal key-token (encrypted key is wrapped with the master key, the key is clear, or there is no payload). **X'02'** External key-token (encrypted payload is wrapped with a transport key, the payload is clear, or there is no payload). A transport key can be a key-encrypting key or an RSA public-key. All unused values are reserved and undefined.
001	01	Reserved, binary zero.
002	02	Length in bytes of the overall token structure. For a null key-token, the value is 8. Otherwise, the length is calculated as: 46 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead* + *uad* + *tlvs* + ((*pl* + 7) / 8) **Value** **Meaning** *kuf* See key usage fields count at offset 44. *kmf* See key management fields count at offset 45 + (2 * *kuf*). *kl* See key label length at offset 46 + (2 * *kuf*) + (2 * *kmf*). *iead* See IBM extended associated data length at offset 46 + (2 * *kuf*) + (2 * *kmf*) + *kl*. *uad* See user associated data length at offset 46 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead*. *tlvs* This value is currently 0. Tag-length-value data is defined for future use. *pl* See payload length in bits at offset 38.
004	01	Token version number (identifies the format of this key token): **Value** **Meaning** **X'05'** Version 5 format of the key token (variable-length symmetric key-token)
005	03	Reserved, binary zero.
End of header		
Wrapping information section (all data related to wrapping the key)		

Table 234. General format of a variable-length symmetric key-token, version X'05' (continued)

Offset (bytes)	Length (bytes)	Description
008	01	Key material state: **Value** **Meaning** **X'00'** No key is present. This is called a skeleton key-token. The key token is external or internal. **X'01'** Key is clear. Only valid with AES CIPHER or HMAC MAC. The key token is external or internal. **X'02'** Key is wrapped with a transport key. When the encrypted section key-wrapping method is AESKW (value at offset 26 is X'02'), the transport key is an AES key-encrypting key. When it is PKOAEP2 (value at offset 26 is X'03'), the transport key is an RSA public-key. The key token is external. **X'03'** Key is wrapped with the AES master-key. The encrypted section key-wrapping method is AESKW. The key token is internal. All unused values are reserved and undefined.
009	01	Key verification pattern (KVP) type: **Value** **Meaning** **X'00'** No KVP (no key present, key is clear, or key is wrapped with an RSA public-key). The key token is external or internal. **X'01'** AESMK (8 leftmost bytes of SHA-256 hash: X'01 ‖ clear AES MK). The key token is internal. **X'02'** Key-encrypting key (8 leftmost bytes of SHA-256 hash: X'01 ‖ clear KEK). The key token is external. All unused values are reserved and undefined.
010	16	KVP of the key used to wrap the payload (value depends on value of key material state, that is, the value at offset 8): **Value at offset** **Value of KVP** **X'00'** The key-material state is no key present. The field should be filled with binary zeros. The key token is external or internal. **X'01'** The key-material state is key is clear. The field should be filled with binary zeros. The key token is external or internal. **X'02'** The key material state is the key is wrapped with a transport key. The value of the KVP depends on the value of the encrypted section key-wrapping method: • When the key-wrapping method is AESKW (value at offset 26 is X'02'), the field contains the KVP of the key-encrypting key used to wrap the key. The 8-byte KEK KVP is left-aligned in the field and padded on the right low-order bytes with binary zeros. • When the key-wrapping method is PKOAEP2 (value at offset 26 is X'03'), the value should be filled with binary zeros. The encoded message, which contains the key, is wrapped with an RSA public-key. The key token is external. **X'03'** The key-material state is the key is wrapped with the AES master-key. The field contains the MKVP of the AES master-key used to wrap the key. The 8-byte MKVP is left-aligned in the field and padded on the right low-order bytes with binary zeros. The key token is internal.

Table 234. General format of a variable-length symmetric key-token, version X'05' (continued)

Offset (bytes)	Length (bytes)	Description
026	01	Encrypted section key-wrapping method (indicates the key-wrapping method used to protect the data in the encrypted section): **Value** **Meaning** **X'00'** No key-wrapping method (no key present or key is clear). The key token is external or internal. **X'02'** AESKW (ANS X9.102). The key token is external with a key wrapped by an AES key-encrypting key, or the key token is internal with a key wrapped by the AES master-key. **X'03'** PKOAEP2. Message *M*, which contains the key, is encoded using the RSAES-OAEP scheme of the RSA PKCS #1 v2.1 standard. The encoded message (*EM*) is produced using the given hash algorithm by encoding message *M* using the Bellare and Rogaway Optimal Asymmetric Encryption Padding (OAEP) method for encoding messages. For PKAOEP2, *M* is defined as follows: *M* = [32 bytes: *hAD*] ‖ [2 bytes: bit length of the clear key] ‖ [clear key] where *hAD* is the message digest of the associated data, and is calculated using the SHA-256 algorithm on the data starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). *EM* is wrapped with an RSA public-key. The key token is external. All unused values are reserved and undefined.

Table 234. General format of a variable-length symmetric key-token, version X'05' (continued)

Offset (bytes)	Length (bytes)	Description
027	01	Hash algorithm used for wrapping key or encoding message. Meaning depends on whether the encrypted section key-wrapping method (value at offset 26) is no key-wrapping method, AESKW, or PKOAEP2: No key-wrapping method (value at offset 26 is X'00') Hash algorithm used for wrapping key when encrypted section key-wrapping method is no key-wrapping method: **Value** **Meaning** **X'00'** No hash (no key present or key is clear). All unused values are reserved and undefined. The key token is external or internal. AESKW key-wrapping method (value at offset 26 is X'02') Hash algorithm used for wrapping key when encrypted section key-wrapping method is AESKW. The value indicates the algorithm used to calculate the message digest of the associated data. The message digest is included in the wrapped payload and is calculated starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). **Value** **Meaning** **X'01'** SHA-224 (not used) **X'02'** SHA-256 **X'04'** SHA-384 (defined for future use) **X'08'** SHA-512 (defined for future use) All unused values are reserved and undefined. The key token is external or internal. PKOAEP2 key-wrapping method (value at offset 26 is X'03') Hash algorithm used for encoding message when encrypted section key-wrapping method is PKOAEP2. The value indicates the given hash algorithm used for encoding message *M* using the RSAES-OAEP scheme of the RSA PKCS #1 v2.1 standard. **Value** **Meaning** **X'01'** SHA-1 **X'02'** SHA-256 **X'04'** SHA-384 **X'08'** SHA-512 All unused values are reserved and undefined. The key token is external.

Table 234. General format of a variable-length symmetric key-token, version X'05' (continued)

Offset (bytes)	Length (bytes)	Description
028	01	Payload format version (identifies format of the payload). Release 4.4 or later, otherwise undefined: **Value** **Meaning** **X'00'** V0 payload (V0PYLD). V0 payload is only supported on HMAC MAC and AES CIPHER, EXPORTER, and IMPORTER key types. The payload format depends on the encrypted section key-wrapping method (value at offset 26): **Value at offset 26** **Meaning** **X'00'** There is no key-wrapping method. When no key is present, there is no payload. When the key is clear (AES CIPHER and HMAC MAC only), the payload is unformatted. The key token is external or internal. **X'02'** The key-wrapping method is AESKW and the payload is variable length. The payload is formatted with the minimum size possible to contain the key material. The payload length varies for a given algorithm and key type. The key length can be inferred by the size of the payload. The key token is external or internal. **X'03'** The key-wrapping method is PKOAEP2 and the payload length is equal to the modulus size in bits of the RSA transport key used to wrap the encoded message. The key token is external. When the external key is exported, the internal target key will have the same V0 payload format. **X'01'** V1 payload (V1PYLD). V1 payload is supported on all key types except HMAC MAC. Release 4.4 or later. The payload format depends on the encrypted section key-wrapping method (value at offset 26): **Value at offset 26** **Meaning** **X'00'** There is no key-wrapping method. When no key is present, there is no payload. When the key is clear (AES CIPHER only), the payload is unformatted. The key token is external or internal. **X'02'** The key-wrapping method is AESKW and the payload is fixed length based on the maximum possible key size of the algorithm for the key. The key is padded with random data to the size of the largest key for that algorithm. This helps to deter attacks on keys known to be weaker. The key length cannot be inferred by the size of the payload. The key token is external or internal. **X'03'** The key-wrapping method is PKOAEP2 and the payload length is equal to the modulus size in bits of the RSA transport key used to wrap the encoded message. The key token is external. When the external key is exported, the internal target key will have the same V1 payload format. All unused values are reserved and undefined.
029	01	Reserved, binary zero.
End of wrapping information section		
Clear key, AESKW, or PKOAEP2 components: (1) associated data sections and (2) optional clear key payload, wrapped AESKW formatted payload, or wrapped PKOAEP2 encoded payload (no payload if no key present)		
Associated data sections: (1) required associated data section and (2) optional associated data sections		
Required associated data section		

Table 234. General format of a variable-length symmetric key-token, version X'05' (continued)

Offset (bytes)	Length (bytes)	Description
030	01	Associated data section version: **Value** **Meaning** **X'01'** Version 1 format of associated data
031	01	Reserved, binary zero.
032	02	Length in bytes of all the associated data for the key token: ≥ 16.
034	01	Length in bytes of the optional key label (*kl*): 0 or 64.
035	01	Length in bytes of the optional IBM extended associated data (*iead*): 0 - 255.
036	01	Length in bytes of the optional user-definable associated data (*uad*): 0 - 255.
037	01	Reserved, binary zero.
038	02	Length in *bits* of the clear or wrapped payload (*pl*): ≥ 0. • For no key-wrapping method (no key present or key is clear), *pl* is the length in bits of the key. For no key present, *pl* is 0. For key is clear (AES CIPHER and HMAC MAC only), *pl* can be 128, 192, or 256 for an AES key, or 80 - 2048 for an HMAC key. • For PKOAEP2 encoded payloads, *pl* is the length in bits of the modulus size of the RSA key used to wrap the payload. This can be 512 - 4096. • For an AESKW formatted payload, *pl* is based on the key size of the algorithm type (DES, AES, or HMAC) and the payload format version: DES algorithm (value at offset 41 is X'01') A DES key can have a length of 8, 16, or 24 bytes (64, 128, 192 bits). A DES key in an AESKW formatted payload is always wrapped with a V1 payload and has a fixed length payload of 576 bits. • AES algorithm (value at offset 41 is X'02'). An AES key can have a length of 16, 24, or 32 bytes (128, 192, or 256 bits). A V0 payload is only valid for HMAC MAC and AES CIPHER, EXPORTER, and EXPORTER keys. A V1 payload is not valid for an HMAC MAC key. The following table shows the payload length for a given AES key size and payload format: **AES key size** / **Bit length of V0 payload (value at offset 28 is X'00')** / **Bit length of V1 payload (value at offset 28 is X'01')** 16 bytes (128 bits) / 512 / 640 24 bytes (192 bits) / 576 / 640 32 bytes (256 bits) / 640 / 640 HMAC algorithm (value at offset 41 is X'03'). An HMAC key can have a length of 80 - 2048 bits. An HMAC key in an AESKW formatted payload is always wrapped with a V0 payload.
040	01	Reserved, binary zero.

Table 234. General format of a variable-length symmetric key-token, version X'05' (continued)

Offset (bytes)	Length (bytes)	Description
041	01	Algorithm type (algorithm for which the key can be used): **Value** **Meaning** **Supported key types by release** **X'01'** DES Release 4.4 or later: DESUSECV **X'02'** AES Release 4.2 or later: CIPHER Release 4.4 or later: MAC Release 4.2 or later: EXPORTER Release 4.2 or later: IMPORTER Release 4.4 or later: PINPROT Release 4.4 or later: PINCALC Release 4.4 or later: PINPRW Release 4.4 or later: DKYGENKY **X'03'** HMAC Release 4.1 or later: MAC All unused values are reserved and undefined.
042	02	Key type (general class of the key): **Value** **Meaning** **X'0001'** CIPHER **X'0002'** MAC **X'0003'** EXPORTER **X'0004'** IMPORTER **X'0005'** PINPROT **X'0006'** PINCALC **X'0007'** PINPRW **X'0008'** DESUSECV **X'0009'** DKYGENKY All unused values are reserved and undefined.
044	01	Key usage fields count (*kuf*): 0 - 255. Key-usage field information defines restrictions on the use of the key. Each key type can have a variable number of key usage fields from none to a maximum of 255. Each key-usage field is 2 bytes in length. The value in this field indicates how many 2-byte key usage fields follow.
045, for *kuf* > 0	01	Optional key-usage field 1, high-order byte. Defined based on algorithm type (value at offset 41) and key type (value at offset 42). All unused bits are reserved and must be zero.
046, for *kuf* > 0	01	Optional key-usage field 1, low-order byte (user-defined extension control): **Value** **Meaning** **B'xxxx 1xxx'** Key can only be used in UDXs (UDX-ONLY). **B'xxxx 0xxx'** Key can be used in UDXs and CCA. **B'xxxx x1uu'** UDX-defined bit reserved for UDXs (UDX-100). **B'xxxx xu1u'** UDX-defined bit reserved for UDXs (UDX-010). **B'xxxx xuu1'** UDX-defined bit reserved for UDXs (UDX-001). **Note:** This byte is common for all key types except for DES DESUSECV in which case this byte is reserved and must zero. All unused bits are reserved and must be zero.

Table 234. General format of a variable-length symmetric key-token, version X'05' (continued)

Offset (bytes)	Length (bytes)	Description
047, for *kuf* > 1	01	Optional key-usage field 2, high-order byte.
048, for *kuf* > 1	01	Optional key-usage field 2, low-order byte.
		. . .
043 + (2 * *kuf*), for *kuf* > 0	01	Optional key-usage field *kuf*, high-order byte.
044 + (2 * *kuf*), for *kuf* > 0	01	Optional key-usage field *kuf*, low-order byte.
045 + (2 * *kuf*)	01	Key management fields count (*kmf*): 0 - 255. Key-management field information describes how the data is to be managed or helps with management of the key material. Each key type can have a variable number of key management fields from none to a maximum of 255. Each key-management field is 2 bytes in length. The value in this field indicates how many 2-byte key management fields follow.
046 + (2 * *kuf*), for *kmf* > 0	01	Optional key-management field 1, high-order byte (export control): Symmetric-key export control: **Value Meaning** **B'1xxx xxxx'** Allow export using symmetric key (XPRT-SYM). **B'0xxx xxxx'** Prohibit export using symmetric key (NOEX-SYM). Unauthenticated asymmetric-key export control: **Value Meaning** **B'x1xx xxxx'** Allow export using unauthenticated asymmetric key (XPRTUASY). **B'x0xx xxxx'** Prohibit export using unauthenticated asymmetric key (NOEXUASY). Authenticated asymmetric-key export control: **Value Meaning** **B'xx1x xxxx'** Allow export using authenticated asymmetric key (XPRTAASY). **B'xx0x xxxx'** Prohibit export using authenticated asymmetric key (NOEXAASY). RAW-key export control: **Value Meaning** **B'xxx1 xxxx'** Allow export using RAW key (XPRT-RAW). Defined for future use. Currently ignored. **B'xxx0 xxxx'** Prohibit export using RAW key (NOEX-RAW). Defined for future use. Currently ignored. **Note:** This byte is common for all key types except for DES DESUSECV in which case this byte is reserved and must zero. All unused bits are reserved and must be zero.

Table 234. General format of a variable-length symmetric key-token, version X'05' (continued)

Offset (bytes)	Length (bytes)	Description
047 + (2 * *kuf*), for *kmf* > 0	01	Optional key-management field 1, low-order byte (export control by algorithm): DES-key export control: **Value Meaning** **B'1xxx xxxx'** Prohibit export using DES key (NOEX-DES). **B'0xxx xxxx'** Allow export using DES key (XPRT-DES). AES-key export control: **Value Meaning** **B'x1xx xxxx'** Prohibit export using AES key (NOEX-AES). **B'x0xx xxxx'** Allow export using AES key (XPRT-AES). RSA-key export control: **Value Meaning** **B'xxxx 1xxx'** Prohibit export using RSA key (NOEX-RSA). **B'xxxx 0xxx'** Allow export using RSA key (XPRT-RSA). **Note:** This byte is common for all key types except for DES DESUSECV in which case this byte is undefined. All unused bits are reserved and must be zero.
048 + (2 * *kuf*), for *kmf* > 1	01	Optional key-management field 2, high-order byte (key completeness): **Value Meaning** **B'11xx xxxx'** Key is incomplete. Key requires at least 2 more parts (MIN3PART). **B'10xx xxxx'** Key is incomplete. Key requires at least 1 more part (MIN2PART). **B'01xx xxxx'** Key is incomplete. Key can be completed or have more parts added (MIN1PART). **B'00xx xxxx'** Key is complete or no key present. If key is present, no more parts can be added (KEYCMPLT). **Note:** This byte is common for all key types except for DES DESUSECV in which case this byte is undefined. All unused bits are reserved and must be zero.

Table 234. General format of a variable-length symmetric key-token, version X'05' (continued)

Offset (bytes)	Length (bytes)	Description
049 + (2 * *kuf*), for *kmf* > 1	01	Optional key-management field 2, low-order byte (security history). Used to reflect the overall history of the key, not just the history at the most recent import or other operation. Release 4.2 or later: **Value Meaning** **B'xxx1 xxxx'** Previously encrypted with an untrusted KEK (UNTRUSTD). **B'xxxx 1xxx'** Previously in a format without type or usage attributes (WOTUATTR). **B'xxxx x1xx'** Previously encrypted with a key weaker than itself (WWEAKKEY). **B'xxxx xx1x'** Previously in a non-CCA format (NOTCCAFM). **B'xxxx xxx1'** Previously encrypted in ECB mode (WECBMODE). **Note:** This byte is common for all key types except for DES DESUSECV in which case this byte is undefined. All unused bits are reserved and must be zero.
050 + (2 * *kuf*), for *kmf* > 2	01	Optional key-management field 3, high-order byte(pedigree original). Used to indicate how the key was originally created and how it got into the current system. Release 4.2 or later. **Value Meaning** **X'00'** Unknown (POUNKNWN). **X'01'** Other method than those defined here, probably used in UDX (POOTHER). **X'02'** Randomly generated (PORANDOM). **X'03'** Established by key agreement such as Diffie-Hellman (POKEYAGR). **X'04'** Created from cleartext key components (POCLRKC). **X'05'** Entered as a cleartext key value (POCLRKV). **X'06'** Derived from another key (PODERVD). **X'07'** Cleartext keys or key parts that were entered at a TKE and secured from there to the target card (POKPSEC). **Note:** This byte is common for all key types except for DES DESUSECV in which case this byte is undefined. All unused values are reserved and must be zero.

Table 234. General format of a variable-length symmetric key-token, version X'05' (continued)

Offset (bytes)	Length (bytes)	Description
051 + (2 * *kuf*), for *kmf* > 2	01	Optional key-management field 3, low-order byte(pedigree current). Used to indicate how the key was originally created and how it got into the current system. **Value** **Meaning** **X'00'** Unknown (PCUNKNWN). **X'01'** Other method than those defined here, probably used in UDX (PCOTHER). **X'02'** Randomly generated (PCRANDOM). **X'03'** Established by key agreement such as Diffie-Hellman (PCKEYAGR). **X'04'** Created from cleartext key components (PCCLCOMP). **X'05'** Entered as a cleartext key value (PCCLVAL). **X'06'** Derived from another key (PCDERVD). **X'07'** Imported from CCA Version X'05' variable-length symmetric key-token with pedigree field (PCMVARWP). **X'08'** Imported from CCA Version X'05' variable-length symmetric key-token with no pedigree field (PCMVARNP). **X'09'** Imported from CCA key-token that contained a nonzero control vector (PCMWCV). **X'0A'** Imported from CCA key-token that either had no control vector or contained a zero control vector (PCMNOCV). **X'0B'** Imported from a TR-31 key block that contained a control vector (ATTR-CV option) (PCMT31WC). **X'0C'** Imported from a TR-31 key block that did not contain a control vector (PCMT31NC). **X'0D'** Imported using PKCS 1.2 RSA encryption (PCMPK1-2). **X'0E'** Imported using PKCS OAEP encryption (PCMOAEP). **X'0F'** Imported using PKA92 RSA encryption (PCMPKA92). **X'10'** Imported using RSA ZERO-PAD encryption (PCMZ-PAD). **X'11'** Converted from a CCA key-token that contained a nonzero control vector (PCCNVTWC). **X'12'** Converted from a CCA key-token that either had no control vector or contained a zero control vector (PCCNVTNC). **X'13'** Cleartext keys or key parts that were entered at a TKE and secured from there to the target card (PCKPSEC). **X'14'** Exported from CCA Version X'05' variable-length symmetric key-token with pedigree field (PCXVARWP). **X'15'** Exported from CCA Version X'05' variable-length symmetric key-token with no pedigree field (PCXVARNP). **X'16'** Exported using PKCS OAEP encryption (PCXOAEP). **Note:** This byte is common for all key types except for DES DESUSECV in which case this byte is undefined. All unused values are reserved and must be zero.
		. . .
044 + (2 * *kuf*) + (2 * *kmf*), for *kmf* > 0	01	Optional key-usage field *kmf*, high-order byte.
045 + (2 * *kuf*) + (2 * *kmf*), for *kmf* > 0	01	Optional key-usage field *kmf*, low-order byte.
046 + (2 * *kuf*) + (2 * *kmf*)	*kl*	Optional key label.
046 + (2 * *kuf*) + (2 * *kmf*) + *kl*	*iead*	Optional IBM extended associated data.

Table 234. General format of a variable-length symmetric key-token, version X'05' (continued)

Offset (bytes)	Length (bytes)	Description
046 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead*	*uad*	Optional user-defined associated data.
End of required associated data section		
Optional associated data sections (defined for future use)		
Optional tag-length-value (TLV) fields. The length of *tlv*n is in bytes and is calculated as follows: *tlv*n = size of TLV*n* tag + size of *tlv*n length field + size of the TLV *n* value in bytes for $n > 0$, where n = number of TLV fields. The summation of TLV lengths is in bytes and is calculated as follows: $tlvs = \sum_{i=1}^{n} tlv_i$ for $n > 0$, where n = number of TLV fields. For $n = 0$, *tlvs* = 0.		
046 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead* + *uad*, for *tlv1* > 0	01	Optional tag-length-value 1 (TLV1) tag.
047 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead* + *uad*, for *tlv1* > 0	02	Optional TLV1 length: *tlv1*.
049 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead* + *uad*, for *tlv1* > 0	*tlv1* - 3	Optional TLV1 value.
046 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead* + *uad* + *tlv1*, for *n* > 1	01	Optional tag-length-value 2 (TLV2) tag.
047 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead* + *uad* + *tlv1*, for *n* > 1	02	Optional TLV2 length: *tlv2*.
049 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead* + *uad* + *tlv1*, for *n* > 1	*tlv2* - *3*	Optional TLV2 value.
		. . .
046 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead* + *uad* + *tlvs* - *tlv* n, for *n* > 0	01	Optional TLV*n* tag.
047 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead* + *uad* + *tlvs* - *tlv* n, for *n* > 0	02	Optional TLV*n* length: *tlv* n
049 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead* + *uad* + *tlvs* - *tlv* n, for *n* > 0	*tlv* n - *3*	Optional TLV*n* value.

Table 234. General format of a variable-length symmetric key-token, version X'05' (continued)

Offset (bytes)	Length (bytes)	Description		
End of TLV fields				
End of optional associated data sections				
End of associated data sections				
Optional clear key payload, wrapped AESKW formatted payload, or wrapped PKOAEP2 encoded payload (no payload if no key present)				
046 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead* + *uad* + *tlvs*	(*pl* + 7) / 8	Contents of payload (*pl* is in *bits*) depending on the encrypted section key-wrapping method (value at offset 26):		
		Value at offset 26	**Encrypted section key-wrapping method**	**Meaning**
		X'00'	No key-wrapping method. Only applies when key is clear, that is, when key material state (value at offset 8) is X'01'.	Only the key material will be in the payload. The key token is external or internal.
		X'02'	AESKW	An encrypted AESKW payload which the Segment 2 code creates by wrapping the unencrypted AESKW formatted payload. The payload is made up of the integrity check value, pad length, length of hash options and hash, hash options, hash of the associated data, key material, and padding. The key token is internal.
		X'03'	PKOAEP2	An encrypted PKOAEP2 payload which the Segment 2 code creates using the RSAES-OAEP scheme of the PKCS #1 v2.1 standard. The message *M* is encoded for a given hash algorithm using the Bellare and Rogaway Optimal Asymmetric Encryption Padding (OAEP) method for encoding messages. For PKAOEP2, *M* is defined as follows: *M* = [32 bytes: *hAD*] ‖ [2 bytes: bit length of the clear key] ‖ [clear key] where *hAD* is the message digest of the associated data, and is calculated using the SHA-256 algorithm starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). The encoded message is wrapped with an RSA public-key according to the standard. The key token is external.
End of optional clear key payload, wrapped AESKW formatted payload, or wrapped PKOAEP2 encoded payload				
End of clear key, AESKW, or PKOAEP2 components				
Note: All numbers are in big endian format.				

AES CIPHER variable-length symmetric key token

View a table showing the format of the CIPHER variable-length symmetric key-token that, beginning with Release 4.2, can be used with the AES algorithm. An AES CIPHER key-token is used by the Symmetric_Algorithm_Decipher (CSNBSAD) and Symmetric_Algorithm_Encipher (CSNBSAE) verbs to decipher or encipher data with the AES algorithm.

Table 235. AES CIPHER variable-length symmetric key-token, version X'05'.

Offset (bytes)	Length (bytes)	Description
000	01	Token identifier: **Value** **Meaning** **X'01'** Internal key-token (encrypted key is wrapped with the master key, the key is clear, or there is no payload). **X'02'** External key-token (encrypted payload is wrapped with a transport key, the payload is clear, or there is no payload). A transport key can be a key-encrypting key or an RSA public-key. All unused values are reserved and undefined.
001	01	Reserved, binary zero.
002	02	Length in bytes of the overall token structure: 46 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead* + *uad* + ((*pl* + 7) / 8) **Key token** **Minimum token length** **Skeleton** 46 + (2 * 2) + (2 * 3) + 0 + 0 + 0 + 0 = 56 **Clear V0 payload** 46 + (2 * 2) + (2 * 3) + 0 + 0 + 0 + ((128 + 7) / 8) = 72 **Encrypted V0 payload** 46 + (2 * 2) + (2 * 3) + 0 + 0 + 0 + ((512 + 7) / 8) = 120 **Encrypted V1 payload** 46 + (2 * 2) + (2 * 3) + 0 + 0 + 0 + ((640 + 7) / 8) = 136 **Key token** **Maximum token length** **External*** 46 + (2 * 2) + (2 * 3) + 64 + 0 + 255 + ((4096 + 7) / 8) = 887 **Internal** 46 + (2 * 2) + (2 * 3) + 64 + 0 + 255 + ((640 + 7) / 8) = 455 *This assumes a PKOAEP2 key-wrapping method using a 4096-bit RSA transport key.
004	01	Token version number (identifies the format of this key token): **Value** **Meaning** **X'05'** Version 5 format of the key token (variable-length symmetric key-token)
005	03	Reserved, binary zero.
End of header		
Wrapping information section (all data related to wrapping the key)		

Table 235. AES CIPHER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
008	01	Key material state: **Value** **Meaning** **X'00'** No key is present. This is called a skeleton key-token. The key token is external or internal. **X'01'** Key is clear. The key token is external or internal. **X'02'** Key is wrapped with a transport key. When the encrypted section key-wrapping method is AESKW (value at offset 26 is X'02'), the transport key is an AES key-encrypting key. When it is PKOAEP2 (value at offset 26 is X'03'), the transport key is an RSA public-key. The key token is external. **X'03'** Key is wrapped with the AES master-key. The encrypted section key-wrapping method is AESKW. The key token is internal. All unused values are reserved and undefined.
009	01	Key verification pattern (KVP) type: **Value** **Meaning** **X'00'** No KVP (no key present, key is clear, or key is wrapped with an RSA public-key). The key token is external or internal. **X'01'** AESMK (8 leftmost bytes of SHA-256 hash: X'01 ‖ clear AES MK). The key token is internal. **X'02'** KEK (8 leftmost bytes of SHA-256 hash: X'01 ‖ clear KEK). The key token is external. All unused values are reserved and undefined.
010	16	KVP (value depends on value of key material state, that is, the value at offset 8): **Value at offset 8** **Value of KVP** **X'00'** The key-material state is no key present. The field should be filled with binary zeros. The key token is external or internal. **X'01'** The key-material state is key is clear. The field should be filled with binary zeros. The key token is external or internal. **X'02'** The key material state is the key is wrapped with a transport key. The value of the KVP depends on the value of the encrypted section key-wrapping method: • When the key-wrapping method is AESKW (value at offset 26 is X'02'), the field contains the KVP of the key-encrypting key used to wrap the key. The 8-byte KEK KVP is left-aligned in the field and padded on the right low-order bytes with binary zeros. • When the key-wrapping method is PKOAEP2 (value at offset 26 is X'03'), the value should be filled with binary zeros. The encoded message, which contains the key, is wrapped with an RSA public-key. The key token is external. **X'03'** The key-material state is the key is wrapped with the AES master-key. The field contains the MKVP of the AES master-key used to wrap the key. The 8-byte MKVP is left-aligned in the field and padded on the right low-order bytes with binary zeros. The key token is internal.

Table 235. AES CIPHER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
026	01	Encrypted section key-wrapping method (how data in the encrypted section is protected): **Value** **Meaning** **X'00'** No key-wrapping method (no key present or key is clear). The key token is external or internal. **X'02'** AESKW (ANS X9.102). The key token is external with a key wrapped by an AES key-encrypting key, or the key token is internal with a key wrapped by the AES master-key. **X'03'** PKOAEP2. Message *M*, which contains the key, is encoded using the RSAES-OAEP scheme of the RSA PKCS #1 v2.1 standard. The encoded message (*EM*) is produced using the given hash algorithm by encoding message *M* using the Bellare and Rogaway Optimal Asymmetric Encryption Padding (OAEP) method for encoding messages. For PKAOEP2, *M* is defined as follows: *M* = [32 bytes: *hAD*] ‖ [2 bytes: bit length of the clear key] ‖ [clear key] where *hAD* is the message digest of the associated data, and is calculated using the SHA-256 algorithm on the data starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). *EM* is wrapped with an RSA public-key. The key token is external. All unused values are reserved and undefined.

Table 235. AES CIPHER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
027	01	Hash algorithm used for wrapping key or encoding message. Meaning depends on whether the encrypted section key-wrapping method (value at offset 26) is no key-wrapping method, AESKW, or PKOAEP2: No key-wrapping method (value at offset 26 is X'00') Hash algorithm used for wrapping key when encrypted section key-wrapping method is no key-wrapping method: **Value** **Meaning** **X'00'** No hash (no key present or key is clear). All unused values are reserved and undefined. The key token is external or internal. AESKW key-wrapping method (value at offset 26 is X'02') Hash algorithm used for wrapping key when encrypted section key-wrapping method is AESKW. The value indicates the algorithm used to calculate the message digest of the associated data. The message digest is included in the wrapped payload and is calculated starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). **Value** **Meaning** **X'02'** SHA-256 All unused values are reserved and undefined. The key token is external or internal. PKOAEP2 key-wrapping method (value at offset 26 is X'03') Hash algorithm used for encoding message when encrypted section key-wrapping method is PKOAEP2. The value indicates the given hash algorithm used for encoding message *M* using the RSAES-OAEP scheme of the RSA PKCS #1 v2.1 standard. **Value** **Meaning** **X'01'** SHA-1 **X'02'** SHA-256 **X'04'** SHA-384 **X'08'** SHA-512 All unused values are reserved and undefined. The key token is external.

Table 235. AES CIPHER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
028	01	Payload format version (identifies format of the payload). Release 4.4 or later, otherwise undefined. **Value** **Meaning** **X'00'** V0 payload (V0PYLD). The payload format depends on the encrypted section key-wrapping method (value at offset 26): **Value at offset 26** **Meaning** **X'00'** There is no key-wrapping method. When no key is present, there is no payload. When the key is clear, the payload is unformatted. The key token is external or internal. **X'02'** The key-wrapping method is AESKW and the payload is variable length. The payload is formatted with the minimum size possible to contain the key material. The payload length varies for a given algorithm and key type. The key length can be inferred by the size of the payload. The key token is external or internal. **X'03'** The key-wrapping method is PKOAEP2 and the payload length is equal to the modulus size in bits of the RSA transport key used to wrap the encoded message. The key token is external. When the external key is exported, the internal target key will have the same V0 payload format. **X'01'** V1 payload (Release 4.4 or later). The payload format depends on the encrypted section key-wrapping method (value at offset 26): **Value at offset 26** **Meaning** **X'00'** There is no key-wrapping method. When no key is present, there is no payload. When the key is clear, the payload is unformatted. The key token is external or internal. **X'02'** The key-wrapping method is AESKW and the payload is fixed length based on the maximum possible key size of the algorithm for the key. The key is padded with random data to the size of the largest key for that algorithm. This helps to deter attacks on keys known to be weaker. The key length cannot be inferred by the size of the payload. The key token is external or internal. **X'03'** The key-wrapping method is PKOAEP2 and the payload length is equal to the modulus size in bits of the RSA transport key used to wrap the encoded message. The key token is external. When the external key is exported, the internal target key will have the same V1 payload format. All unused values are reserved and undefined.
029	01	Reserved, binary zero.
End of wrapping information section		
Clear key, AESKW, or PKOAEP2 components: (1) associated data section and (2) optional clear key payload, wrapped AESKW formatted payload, or wrapped PKOAEP2 encoded payload (no payload if no key present)		
Associated data section		
030	01	Associated data section version: **Value** **Meaning** **X'01'** Version 1 format of associated data
031	01	Reserved, binary zero.
032	02	Length in bytes of all the associated data for the key token: 26 - 345.
034	01	Length in bytes of the optional key label (*kl*): 0 or 64.
035	01	Length in bytes of the optional IBM extended associated data (*iead*): 0.
036	01	Length in bytes of the optional user-definable associated data (*uad*): 0 - 255.

Table 235. AES CIPHER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
037	01	Reserved, binary zero.
038	02	Length in bits of the clear or wrapped payload (*pl*): 0, 128, 192, 256, 512 - 4096. • For no key-wrapping method (no key present or key is clear), *pl* is the length in bits of the key. For no key present, *pl* is 0. For key is clear, *pl* can be 128, 192, or 256. • For PKOAEP2 encoded payloads, *pl* is the length in bits of the modulus size of the RSA key used to wrap the payload. This can be 512 - 4096. • For an AESKW formatted payload, *pl* is based on the key size of the algorithm type and the payload format version: AES algorithm (value at offset 41 is X'02') An AES key can have a length of 16, 24, or 32 bytes (128, 192, or 256 bits). The following table shows the payload length for a given AES key size and payload format: **AES key size** / **Bit length of V0 payload (value at offset 28 is X'00'** / **Bit length of V1 payload (value at offset 28 is X'01')** 16 bytes (128 bits) / 512 / 640 24 bytes (192 bits) / 576 / 640 32 bytes (256 bits) / 640 / 640
040	01	Reserved, binary zero.
041	01	Algorithm type (algorithm for which the key can be used): **Value** **Meaning** **X'02'** AES All unused values are reserved and undefined.
042	02	Key type (general class of the key): **Value** **Meaning** **X'0001'** CIPHER All unused values are reserved and undefined.
044	01	Key usage fields count (*kuf*): 2. Key-usage field information defines restrictions on the use of the key. Each key-usage field is 2 bytes in length. The value in this field indicates how many 2-byte key usage fields follow.

Table 235. AES CIPHER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
045	01	Key-usage field 1, high-order byte (encryption and translation control): **Value Meaning** **B'1xxx xxxx'** Key can be used for encryption (ENCRYPT). **B'0xxx xxxx'** Key cannot be used for encryption. **B'x1xx xxxx'** Key can be used for decryption (DECRYPT). **B'x0xx xxxx'** Key cannot be used for decryption. **B'xx1x xxxx'** Key can be used for data translate (C-XLATE) (Release 4.3 or later). Undefined (releases before Release 4.4). **B'xx0x xxxx'** Key cannot be used for data translate (Release 4.4 or later). Undefined (releases before Release 4.4). **B'00xx xxxx'** Undefined or not used (releases before Release 4.4). **B'000x xxxx'** Undefined or not used (Release 4.4 or later). All unused bits are reserved and must be zero.
046	01	Key-usage field 1, low-order byte (user-defined extension control). Refer to Table 234 on page 801.
047	01	Key-usage field 2, high-order byte (encryption mode): **Value Meaning** **X'00'** Key can be used for Cipher Block Chaining (CBC). **X'01'** Key can be used for Electronic Code Book (ECB). **X'02'** Key can be used for Cipher Feedback (CFB). **X'03'** Key can be used for Output Feedback (OFB). **X'04'** Key can be used for Galois/Counter Mode (GCM). **X'05'** Key can be used for Xor-Encrypt-Xor-based Tweaked Stealing (XTS). All unused values are reserved and undefined.
048	01	Key-usage field 2, low-order byte (reserved). All bits are reserved and must be zero.
049	01	Key management fields count (*kmf*): 3. Key-management field information describes how the data is to be managed or helps with management of the key material. Each key-management field is 2 bytes in length. The value in this field indicates how many 2-byte key management fields follow.
050	01	Key-management field 1, high-order byte (symmetric-key export control). Refer to Table 234 on page 801.
051	01	Key-management field 1, low-order byte (export control by algorithm). Refer to Table 234 on page 801.
052	01	Key-management field 2, high-order byte (key completeness). Refer to Table 234 on page 801.
053	01	Key-management field 2, low-order byte (security history). Refer to Table 234 on page 801.
054	01	Key-management field 3, high-order byte (pedigree original). Refer to Table 234 on page 801.
055	01	Key-management field 3, low-order byte (pedigree current). Refer to Table 234 on page 801.
056	*kl*	Optional key label.
056 + *kl*	*iead*	Optional IBM extended associated data (unused).

Table 235. AES CIPHER variable-length symmetric key-token, version X'05' (continued).

<table>
<tr><th>Offset (bytes)</th><th>Length (bytes)</th><th colspan="3">Description</th></tr>
<tr><td>056 + kl + iead</td><td>uad</td><td colspan="3">Optional user-defined associated data.</td></tr>
<tr><td colspan="5">End of associated data section</td></tr>
<tr><td colspan="5">Optional clear key payload, wrapped AESKW formatted payload, or wrapped PKOAEP2 encoded payload (no payload if no key present)</td></tr>
<tr><td rowspan="4">056 + kl + iead + uad</td><td rowspan="4">(pl + 7) / 8</td><td colspan="3">Contents of payload (pl is in bits) depending on the encrypted section key-wrapping method (value at offset 26):</td></tr>
<tr><td>Value at offset 26</td><td>Encrypted section key-wrapping method</td><td>Meaning</td></tr>
<tr><td>X'00'</td><td>No key-wrapping method. Only applies when key is clear, that is, when key material state (value at offset 8) is X'01'.</td><td>Only the key material will be in the payload. The key token is external or internal.</td></tr>
<tr><td>X'02'</td><td>AESKW</td><td>An encrypted payload which the Segment 2 code creates by wrapping the unencrypted AESKW formatted payload. The payload is made up of the integrity check value, pad length, length of hash options and hash, hash options, hash of the associated data, key material, and padding. The key token is internal.</td></tr>
<tr><td>X'03'</td><td>PKOAEP2</td><td>An encrypted PKOAEP2 encoded payload created using the RSAES-OAEP scheme of the PKCS #1 v2.1 standard. The message M is encoded for a given hash algorithm using the Bellare and Rogaway Optimal Asymmetric Encryption Padding (OAEP) method for encoding messages. For PKAOEP2, M is defined as follows:

M = [32 bytes: hAD] ‖ [2 bytes: bit length of the clear key] ‖ [clear key]

where hAD is the message digest of the associated data, and is calculated using the SHA-256 algorithm starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). The encoded message is wrapped with an RSA public-key according to the standard. The key token is external.</td></tr>
<tr><td colspan="5">End of optional clear key payload, wrapped AESKW formatted payload, or wrapped PKOAEP2 encoded payload</td></tr>
<tr><td colspan="5">End of clear key, AESKW, or PKOAEP2 components</td></tr>
<tr><td colspan="5">Note: All numbers are in big endian format.</td></tr>
</table>

AES MAC variable-length symmetric key token

View a table showing the format of an AES MAC variable-length symmetric key token.

Table 236. AES MAC variable-length symmetric key-token, version X'05'.

Offset (bytes)	Length (bytes)	Description
Header		
000	01	Token identifier: **Value** **Meaning** **X'01'** Internal key-token (encrypted key is wrapped with the master key or there is no payload). **X'02'** External key-token (encrypted payload is wrapped with a transport key or there is no payload). A transport key can be a key-encrypting key or an RSA public-key. All unused values are reserved and undefined.
001	01	Reserved, binary zero.
002	02	Length in bytes of the overall token structure: 46 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead* + *uad* + ((*pl* + 7) / 8) **Key token** **Minimum token length without DK enabled** **Skeleton** 46 + (2 * 2) + (2 * 3) + 0 + 0 + 0 + 0 = 56 **Encrypted V1 payload** 46 + (2 * 2) + (2 * 3) + 0 + 0 + 0 + ((640 + 7) / 8) = 136 **Key token** **Minimum token length with DK enabled** **Skeleton** 46 + (2 * 3) + (2 * 3) + 0 + 0 + 0 + 0 = 58 **Encrypted V1 payload** 46 + (2 * 3) + (2 * 3) + 0 + 0 + 0 + ((640 + 7) / 8) = 138 **Key token** **Maximum token length without DK enabled** **External*** 46 + (2 * 2) + (2 * 3) + 64 + 0 + 255 + ((4096 + 7) / 8) = 887 **Internal** 46 + (2 * 2) + (2 * 3) + 64 + 0 + 255 + ((640 + 7) / 8) = 455 **Key token** **Maximum token length with DK enabled** **External*** 46 + (2 * 3) + (2 * 3) + 64 + 0 + 255 + ((4096 + 7) / 8) = 889 **Internal** 46 + (2 * 3) + (2 * 3) + 64 + 0 + 255 + ((640 + 7) / 8) = 457 *This assumes a PKOAEP2 key-wrapping method using a 4096-bit RSA transport key.
004	01	Token version number (identifies the format of this key token): **Value** **Meaning** **X'05'** Version 5 format of the key token (variable-length symmetric key-token)
005	03	Reserved, binary zero.
End of header		
Wrapping information section (all data related to wrapping the key)		

Table 236. AES MAC variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
008	01	Key material state: **Value** **Meaning** **X'00'** No key is present. This is called a skeleton key-token. The key token is external or internal. **X'02'** Key is wrapped with a transport key. When the encrypted section key-wrapping method is AESKW (value at offset 26 is X'02'), the transport key is an AES key-encrypting key. When it is PKOAEP2 (value at offset 26 is X'03'), the transport key is an RSA public-key. The key token is external. **X'03'** Key is wrapped with the AES master-key. The encrypted section key-wrapping method is AESKW. The key token is internal. All unused values are reserved and undefined.
009	01	Key verification pattern (KVP) type: **Value** **Meaning** **X'00'** No KVP (no key present or key is wrapped with an RSA public-key). The key token is external or internal. **X'01'** AESMK (8 leftmost bytes of SHA-256 hash: X'01 ‖ clear AES MK). The key token is internal. **X'02'** KEK (8 leftmost bytes of SHA-256 hash: X'01 ‖ clear KEK). The key token is external. All unused values are reserved and undefined.
010	16	KVP (value depends on value of key material state, that is, the value at offset 8): **Value at offset 8** **Value of KVP** **X'00'** The key-material state is no key present. The field should be filled with binary zeros. The key token is external or internal. **X'02'** The key material state is the key is wrapped with a transport key. The value of the KVP depends on the value of the encrypted section key-wrapping method: • When the key-wrapping method is AESKW (value at offset 26 is X'02'), the field contains the KVP of the key-encrypting key used to wrap the key. The 8-byte KEK KVP is left-aligned in the field and padded on the right low-order bytes with binary zeros. • When the key-wrapping method is PKOAEP2 (value at offset 26 is X'03'), the value should be filled with binary zeros. The encoded message, which contains the key, is wrapped with an RSA public-key. The key token is external. **X'03'** The key-material state is the key is wrapped with the AES master-key. The field contains the MKVP of the AES master-key used to wrap the key. The 8-byte MKVP is left-aligned in the field and padded on the right low-order bytes with binary zeros. The key token is internal. All unused values are reserved and undefined.

Table 236. AES MAC variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
026	01	Encrypted section key-wrapping method (how data in the encrypted section is protected): **Value** **Meaning** **X'00'** No key-wrapping method (no key present). The key token is external or internal. **X'02'** AESKW (ANS X9.102). The key token is external with a key wrapped by an AES key-encrypting key, or the key token is internal with a key wrapped by the AES master-key. **X'03'** PKOAEP2. Message *M*, which contains the key, is encoded using the RSAES-OAEP scheme of the RSA PKCS #1 v2.1 standard. The encoded message (*EM*) is produced using the given hash algorithm by encoding message *M* using the Bellare and Rogaway Optimal Asymmetric Encryption Padding (OAEP) method for encoding messages. For PKAOEP2, *M* is defined as follows: *M* = [32 bytes: *hAD*] ‖ [2 bytes: bit length of the clear key] ‖ [clear key] where *hAD* is the message digest of the associated data, and is calculated using the SHA-256 algorithm on the data starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). *EM* is wrapped with an RSA public-key. The key token is external. All unused values are reserved and undefined.

Table 236. AES MAC variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
027	01	Hash algorithm used for wrapping key or encoding message. Meaning depends on whether the encrypted section key-wrapping method (value at offset 26) is no key-wrapping method, AESKW, or PKOAEP2: No key-wrapping method (value at offset 26 is X'00') Hash algorithm used for wrapping key when encrypted section key-wrapping method is no key-wrapping method: **Value** **Meaning** **X'00'** No hash (no key present) All unused values are reserved and undefined. The key token is external or internal. AESKW key-wrapping method (value at offset 26 is X'02') Hash algorithm used for wrapping key when encrypted section key-wrapping method is AESKW. The value indicates the algorithm used to calculate the message digest of the associated data. The message digest is included in the wrapped payload and is calculated starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). **Value** **Meaning** **X'02'** SHA-256 All unused values are reserved and undefined. The key token is external or internal. PKOAEP2 key-wrapping method (value at offset 26 is X'03') Hash algorithm used for encoding message when encrypted section key-wrapping method is PKOAEP2. The value indicates the given hash algorithm used for encoding message *M* using the RSAES-OAEP scheme of the RSA PKCS #1 v2.1 standard. **Value** **Meaning** **X'01'** SHA-1 **X'02'** SHA-256 **X'04'** SHA-384 **X'08'** SHA-512 All unused values are reserved and undefined. The key token is external.

Table 236. AES MAC variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
028	01	Payload format version (identifies format of the payload): **Value** **Meaning** **X'01'** V1 payload (V1PYLD). The payload format depends on the encrypted section key-wrapping method (value at offset 26): **Value at offset 26** **Meaning** **X'00'** There is no key-wrapping method. When no key is present, there is no payload. The key token is external or internal. **X'02'** The key-wrapping method is AESKW and the payload is fixed length based on the maximum possible key size of the algorithm for the key. The key is padded with random data to the size of the largest key for that algorithm. This helps to deter attacks on keys known to be weaker. The key length cannot be inferred by the size of the payload. The key token is external or internal. **X'03'** The key-wrapping method is PKOAEP2 and the payload length is equal to the modulus size in bits of the RSA transport key used to wrap the encoded message. The key token is external. When the external key is exported, the internal target key will have the same V1 payload format. All unused values are reserved and undefined.
029	01	Reserved, binary zero.
End of wrapping information section		
AESKW or PKOAEP2 components: (1) associated data section and (2) optional wrapped AESKW formatted payload or wrapped PKOAEP2 encoded payload (no payload if no key present)		
Associated data section		
030	01	Associated data section version: **Value** **Meaning** **X'01'** Version 1 format of associated data
031	01	Reserved, binary zero.
032	02	Length in bytes of all the associated data for the key token: 26 - 347.
034	01	Length in bytes of the optional key label (*kl*): 0 or 64.
035	01	Length in bytes of the optional IBM extended associated data (*iead*): 0.
036	01	Length in bytes of the optional user-definable associated data (*uad*): 0 - 255.
037	01	Reserved, binary zero.

Table 236. AES MAC variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
038	02	Length in *bits* of the wrapped payload (*pl*): 0, 512 - 4096. • For no key-wrapping method (no key present), *pl* is 0. • For PKOAEP2 encoded payloads, *pl* is the length in bits of the modulus size of the RSA key used to wrap the payload. This can be 512 - 4096. • For an AESKW formatted payload, *pl* is based on the key size of the algorithm type and the payload format version: AES algorithm (value at offset 41 is X'02') An AES key can have a length of 16, 24, or 32 bytes (128, 192, or 256 bits). The following table shows the payload length for a given AES key size and payload format: **AES key size** / **Bit length of V0 payload (value at offset 28 is X'00')** / **Bit length of V1 payload (value at offset 28 is X'01')** 16 bytes (128 bits) / Not applicable / 640 24 bytes (192 bits) / Not applicable / 640 32 bytes (256 bits) / Not applicable / 640
040	01	Reserved, binary zero.
041	01	Algorithm type (algorithm for which the key can be used): **Value** **Meaning** **X'02'** AES All unused values are reserved and undefined.
042	02	Key type (general class of the key): **Value** **Meaning** **X'0002'** MAC All unused values are reserved and undefined.
044	01	Key usage fields count (*kuf*): 2 - 3. Key-usage field information defines restrictions on the use of the key. Count is based on whether the key is DK enabled or not: **DK enabled** **kuf** **No** 2 **Yes** 3 Each key-usage field is 2 bytes in length. The value in this field indicates how many 2-byte key usage fields follow. Key-usage field information defines restrictions on the use of the key.

Table 236. AES MAC variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
045	01	**Key-usage field 1, high-order byte** (MAC operation). **Value Meaning** **B'11xx xxxx'** For *kuf* > 2, if DK enabled (value at offset 50 = X'01') then this value is undefined or not used. Otherwise, key can be used for generate; key can be used for verify (GENERATE). **B'10xx xxxx'** Key can be used for generate; key cannot be used for verify (GENONLY). **B'01xx xxxx'** Key cannot be used for generate; key can be used for verify (VERIFY). **B'00xx xxxx'** Undefined or not used. All unused bits are reserved and must be zero.
046	01	**Key-usage field 1, low-order byte** (user-defined extension control).
047	01	**Key-usage field 2, high-order byte** (MAC mode). **Value Meaning** **X'01'** CMAC mode (CMAC). NIST SP 800-38B. All unused values are reserved and undefined.
048	01	**Key-usage field 2, low-order byte** (reserved). All bits are reserved and must be zero.
049, for *kuf* > 2	01	**Key-usage field 3, high-order byte**. The meaning is determined by the field format identifier (value at offset 50). Currently the only field format identifier is DK enabled: DK enabled (value at offset 50 is X'01') Common control: **Value Meaning** **X'01'** PIN_OP (DKPINOP) **X'03'** PIN_AD1 (DKPINAD1) **X'04'** PIN_AD2 (DKPINAD2) All unused values are reserved and undefined.
050, for *kuf* > 2	01	**Key-usage field 3, low-order byte** (field format identifier). Identifies the format of key-usage field 3, high-order byte (value at offset 49): **Value Meaning** **X'01'** DK enabled (set when **DKPINOP**, **DKPINAD1**, or **DKPINAD2** keyword is used) All unused values are reserved and undefined.
045 + (2 * *kuf*)	01	Key management fields count (*kmf*): 3. Key-management field information describes how the data is to be managed or helps with management of the key material. Each key-management field is 2 bytes in length. The value in this field indicates how many 2-byte key management fields follow.
046 + (2 * *kuf*)	01	Key-management field 1, high-order byte (symmetric-key export control).
047 + (2 * *kuf*)	01	Key-management field 1, low-order byte (export control by algorithm).
048 + (2 * *kuf*)	01	Key-management field 2, high-order byte (key completeness).

Table 236. AES MAC variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description		
049 + (2 * *kuf*)	01	Key-management field 2, low-order byte (security history).		
050 + (2 * *kuf*)	01	Key-management field 3, high-order byte (pedigree original).		
051 + (2 * *kuf*)	01	Key-management field 3, low-order byte (pedigree current).		
052 + (2 * *kuf*)	*kl*	Optional key label.		
052 + (2 * *kuf*) + *kl*	*iead*	Optional IBM extended associated data (unused).		
*052 + (2 * kuf) + kl + iead*	*uad*	Optional user-defined associated data.		
End of associated data section				
Optional wrapped AESKW formatted payload or wrapped PKOAEP2 encoded payload (no payload if no key present)				
*052 + (2 * kuf) + kl + iead + uad*	(*pl* + 7) / 8	Contents of payload (*pl* is in *bits*) depending on the encrypted section key-wrapping method (value at offset 26):		
		Value at offset 26	**Encrypted section key-wrapping method**	**Meaning**
		X'02'	AESKW	An encrypted payload which the Segment 2 code creates by wrapping the unencrypted AESKW formatted payload. The payload is made up of the integrity check value, pad length, length of hash options and hash, hash options, hash of the associated data, key material, and padding. The key token is internal.
		X'03'	PKOAEP2	An encrypted PKOAEP2 encoded payload created using the RSAES-OAEP scheme of the PKCS #1 v2.1 standard. The message *M* is encoded for a given hash algorithm using the Bellare and Rogaway Optimal Asymmetric Encryption Padding (OAEP) method for encoding messages. For PKAOEP2, *M* is defined as follows: *M* = [32 bytes: *hAD*] ‖ [2 bytes: bit length of the clear key] ‖ [clear key] where *hAD* is the message digest of the associated data, and is calculated using the SHA-256 algorithm starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). The encoded message is wrapped with an RSA public-key according to the standard. The key token is external.
End of optional wrapped AESKW formatted payload or wrapped PKAOEP2 encoded payload				
End of AESKW or PKOAEP2 components				
Note: All numbers are in big endian format.				

HMAC MAC variable-length symmetric key token

View a table showing the format of the HMAC variable-length symmetric key-token.

Table 237 shows the format of the HMAC MAC variable-length symmetric key-token. An HMAC token is used by the HMAC Generate(CSNBHMG) and HMAC Verify(CSNBHMV) verbs to generate and verify keyed hash Message Authentication Codes.

Table 237. HMAC MAC variable-length symmetric key-token, version X'05' (CCA 4.1.0 or later)

Offset (bytes)	Length (bytes)	Description
Header		
000	01	Token identifier: **Value** **Meaning** **X'01'** Internal key-token (encrypted key is wrapped with the master key, the key is clear, or there is no payload). **X'02'** External key-token (encrypted payload is wrapped with a transport key, the payload is clear, or there is no payload). A transport key can be a key-encrypting key or an RSA public-key. All unused values are reserved and undefined.
001	01	Reserved, binary zero.
002	02	Length in bytes of the overall token structure: 46 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead* + *uad* + ((*pl* + 7) / 8) **Key token** **Minimum token length (Release 4.1)** **Skeleton** 46 + (2 * 2) + (2 * 2) + 0 + 0 + 0 + 0 = 54 **Clear V0 payload** 46 + (2 * 2) + (2 * 2) + 0 + 0 + 0 + ((80 + 7) / 8) = 64 **Encrypted V0 payload** 46 + (2 * 2) + (2 * 2) + 0 + 0 + 0 + ((448 + 7) / 8) = 110 **Key token** **Minimum token length (Release 4.2 or later)** **Skeleton** 46 + (2 * 2) + (2 * 3) + 0 + 0 + 0 + 0 = 56 **Clear V0 payload** 46 + (2 * 2) + (2 * 3) + 0 + 0 + 0 + ((80 + 7) / 8) = 66 **Encrypted V0 payload** 46 + (2 * 2) + (2 * 3) + 0 + 0 + 0 + ((448 + 7) / 8) = 112 **Key token** **Maximum token length (Release 4.1)** **External*** 46 + (2 * 2) + (2 * 2) + 64 + 0 + 255 + ((4096 + 7) / 8) = 885 **Input** 46 + (2 * 2) + (2 * 2) + 64 + 0 + 255 + ((2432 + 7) / 8) = 677 **Key token** **Maximum token length (Release 4.2 or later)** **External*** 46 + (2 * 2) + (2 * 3) + 64 + 0 + 255 + ((4096 + 7) / 8) = 887 **Internal** 46 + (2 * 2) + (2 * 3) + 64 + 0 + 255 + ((2432 + 7) / 8) = 679 *This assumes a PKOAEP2 key-wrapping method using a 4096-bit RSA transport key.

Table 237. HMAC MAC variable-length symmetric key-token, version X'05' (CCA 4.1.0 or later) (continued)

Offset (bytes)	Length (bytes)	Description
004	01	Token version number (identifies the format of this key token): **Value** **Meaning** **X'05'** Version 5 format of the key token (variable-length symmetric key-token)
005	03	Reserved, binary zero.
End of header		
Wrapping information section (all data related to wrapping the key)		
008	01	Key material state: **Value** **Meaning** **X'00'** No key is present. This is called a skeleton key-token. The key token is external or internal. **X'01'** Key is clear. The key token is external or internal. **X'02'** Key is wrapped with a transport key. When the encrypted section key-wrapping method is AESKW (value at offset 26 is X'02'), the transport key is an AES key-encrypting key. When it is PKOAEP2 (value at offset 26 is X'03'), the transport key is an RSA public-key. The key token is external. **X'03'** Key is wrapped with the AES master-key. The encrypted section key-wrapping method is AESKW. The key token is internal. All unused values are reserved and undefined.
009	01	Key verification pattern (KVP) type: **Value** **Meaning** **X'00'** No KVP (no key present, key is clear, or key is wrapped with an RSA public-key). The key token is external or internal. **X'01'** AESMK (8 leftmost bytes of SHA-256 hash: X'01 ‖ clear AES MK). The key token is internal. **X'02'** KEK (8 leftmost bytes of SHA-256 hash: X'01 ‖ clear KEK). The key token is external. All unused values are reserved and undefined.
010	16	KVP (value depends on value of key material state, that is, the value at offset 8): **Value at offset 8** **Value of KVP** **X' 00 '** The key-material state is no key present. The field should be filled with binary zeros. The key token is external or internal. **X'01'** The key-material state is key is clear. The field should be filled with binary zeros. The key token is external or internal. **X'02'** The key material state is the key is wrapped with a transport key. The value of the KVP depends on the value of the encrypted section key-wrapping method: • When the key-wrapping method is AESKW (value at offset 26 is X'02'), the field contains the KVP of the key-encrypting key used to wrap the key. The 8-byte KEK KVP is left-aligned in the field and padded on the right low-order bytes with binary zeros. • When the key-wrapping method is PKOAEP2 (value at offset 26 is X'03'), the value should be filled with binary zeros. The encoded message, which contains the key, is wrapped with an RSA public-key. • The key token is external. **X'03'** The key-material state is the key is wrapped with the AES master-key. The field contains the MKVP of the AES master-key used to wrap the key. The 8-byte MKVP is left-aligned in the field and padded on the right low-order bytes with binary zeros. The key token is internal.

Table 237. HMAC MAC variable-length symmetric key-token, version X'05' (CCA 4.1.0 or later) (continued)

Offset (bytes)	Length (bytes)	Description
026	01	Encrypted section key-wrapping method (how data in the encrypted section is protected): **Value** **Meaning** **X'00'** No key-wrapping method (no key present or key is clear). The key token is external or internal. **X'02'** AESKW (ANS X9.102). The key token is external with a key wrapped by an AES key-encrypting key, or the key token is internal with a key wrapped by the AES master-key. **X'03'** PKOAEP2. Message *M*, which contains the key, is encoded using the RSAES-OAEP scheme of the RSA PKCS #1 v2.1 standard. The encoded message (*EM*) is produced using the given hash algorithm by encoding message *M* using the Bellare and Rogaway Optimal Asymmetric Encryption Padding (OAEP) method for encoding messages. For PKAOEP2, *M* is defined as follows: *M* = [32 bytes: *hAD*] ‖ [2 bytes: bit length of the clear key] ‖ [clear key] where *hAD* is the message digest of the associated data, and is calculated using the SHA-256 algorithm on the data starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). *EM* is wrapped with an RSA public-key. The key token is external. All unused values are reserved and undefined.

Table 237. HMAC MAC variable-length symmetric key-token, version X'05' (CCA 4.1.0 or later) (continued)

Offset (bytes)	Length (bytes)	Description
027	01	Hash algorithm used for wrapping key or encoding message. Meaning depends on whether the encrypted section key-wrapping method (value at offset 26) is no key-wrapping method, AESKW, or PKOAEP2: No key-wrapping method (value at offset 26 is X'00') Hash algorithm used for wrapping key when encrypted section key-wrapping method is no key-wrapping method: **Value** **Meaning** **X'00'** No hash (no key present or key is clear). All unused values are reserved and undefined. The key token is external or internal. AESKW key-wrapping method (value at offset 26 is X'02') Hash algorithm used for wrapping key when encrypted section key-wrapping method is AESKW. The value indicates the algorithm used to calculate the message digest of the associated data. The message digest is included in the wrapped payload and is calculated starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). **Value** **Meaning** **X'02'** SHA-256 All unused values are reserved and undefined. The key token is external or internal. PKOAEP2 key-wrapping method (value at offset 26 is X'03') Hash algorithm used for encoding message when encrypted section key-wrapping method is PKOAEP2. The value indicates the given hash algorithm used for encoding message *M* using the RSAES-OAEP scheme of the RSA PKCS #1 v2.1 standard. **Value** **Meaning** **X'01'** SHA-1 **X'02'** SHA-256 **X'04'** SHA-384 **X'08'** SHA-512 All unused values are reserved and undefined. The key token is external.

Table 237. HMAC MAC variable-length symmetric key-token, version X'05' (CCA 4.1.0 or later) (continued)

Offset (bytes)	Length (bytes)	Description
028	01	Payload format version (identifies format of the payload). Release 4.4 or later, otherwise undefined. **Value** **Meaning** **X'00'** V0 payload (V0PYLD). The payload format depends on the encrypted section key-wrapping method (value at offset 26): **Value at offset 26** **Meaning** **X'00'** There is no key-wrapping method. When no key is present, there is no payload. When the key is clear, the payload is unformatted. The key token is external or internal. **X'02'** The key-wrapping method is AESKW and the payload is variable length. The payload is formatted with the minimum size possible to contain the key material. The payload length varies for a given algorithm and key type. The key length can be inferred by the size of the payload. The key token is external or internal. **X'03'** The key-wrapping method is PKOAEP2 and the payload length is equal to the modulus size in bits of the RSA transport key used to wrap the encoded message. The key token is external. When the external key is exported, the internal target key will have the same V0 payload format. All unused values are reserved and undefined.
029	01	Reserved, binary zero.
End of wrapping information section		
Clear key, AESKW, or PKOAEP2 components: (1) associated data section and (2) optional clear key payload, wrapped AESKW formatted payload, or wrapped PKOAEP2 encoded payload (no payload if no key present)		
Associated data section		
030	01	Associated data section version: **Value** **Meaning** **X'01'** Version 1 format of associated data
031	01	Reserved, binary zero.
032	02	Length in bytes of all the associated data for the key token: 24 - 343 (Release 4.1); 26 - 345 (Release 4.2 or later).
034	01	Length in bytes of the optional key label (*kl*): 0 or 64.
035	01	Length in bytes of the optional IBM extended associated data (*iead*): 0.
036	01	Length in bytes of the optional user-definable associated data (*uad*): 0 - 255.
037	01	Reserved, binary zero.
038	02	Length in *bits* of the clear or wrapped payload (*pl*): 0, 80 - 4096. • For no key-wrapping method (no key present or key is clear), *pl* is the length in bits of the key. For no key present, *pl* is 0. For key is clear, *pl* can be 128, 192, or 256. • For PKOAEP2 encoded payloads, *pl* is the length in bits of the modulus size of the RSA key used to wrap the payload. This can be 512 - 4096. • For an AESKW formatted payload, *pl* is based on the key size of the algorithm type and the payload format version: HMAC algorithm (value at offset 41 is X'03') An HMAC key can have a length of 80 - 2048 bits. An HMAC key in an AESKW formatted payload is always wrapped with a V0 payload.
040	01	Reserved, binary zero.

Table 237. HMAC MAC variable-length symmetric key-token, version X'05' (CCA 4.1.0 or later) (continued)

Offset (bytes)	Length (bytes)	Description
041	01	Algorithm type (algorithm for which the key can be used): **Value Meaning** **X'03'** HMAC All unused values are reserved and undefined.
042	02	Key type (general class of the key): **Value Meaning** **X'0002'** MAC All unused values are reserved and undefined.
044	01	Key usage fields count (*kuf*): 2. Key-usage field information defines restrictions on the use of the key. Refer to Figure 10 on page 241. Each key-usage field is 2 bytes in length. The value in this field indicates how many 2-byte key usage fields follow.
045	01	**Key-usage field 1, high-order byte** (MAC operation): **Value Meaning** **B'11xx xxxx'** Key can be used for generate; key can be used for verify (GENERATE). **B'10xx xxxx'** Undefined or not used. **B'01xx xxxx'** Key cannot be used for generate; key can be used for verify (VERIFY). **B'00xx xxxx'** Undefined or not used. All unused bits are reserved and must be zero.
046	01	**Key-usage field 1, low-order byte** (user-defined extension control)
047	01	**Key-usage field 2, high-order byte** (hash method): **Value Meaning** **B'1xxx xxxx'** SHA-1 hash method is allowed for the key (SHA-1). **B'0xxx xxxx'** SHA-1 hash method is not allowed for the key. **B'x1xx xxxx'** SHA-224 hash method is allowed for the key (SHA-224). **B'x0xx xxxx'** SHA-224 hash method is not allowed for the key. **B'xx1x xxxx'** SHA-256 hash method is allowed for the key (SHA-256). **B'xx0x xxxx'** SHA-256 hash method is not allowed for the key. **B'xxx1 xxxx'** SHA-384 hash method is allowed for the key (SHA-384). **B'xxx0 xxxx'** SHA-384 hash method is not allowed for the key. **B'xxxx 1xxx'** SHA-512 hash method is allowed for the key (SHA-512). **B'xxxx 0xxx'** SHA-512 hash method is not allowed for the key. All unused bits are reserved and must be zero.

Table 237. HMAC MAC variable-length symmetric key-token, version X'05' (CCA 4.1.0 or later) (continued)

Offset (bytes)	Length (bytes)	Description
048	01	**Key-usage field 2, low-order byte** (reserved). All bits are reserved and must be zero.
049	01	Key management fields count (*kmf*): 2 for Release 4.1; 3 for Release 4.1 or later. Key-management field information describes how the data is to be managed or helps with management of the key material. Each key-management field is 2 bytes in length. The value in this field indicates how many 2-byte key management fields follow.
050	01	Key-management field 1, high-order byte (symmetric-key export control).
051	01	Key-management field 1, low-order byte (export control by algorithm).
052	01	Key-management field 2, high-order byte (key completeness).
053	01	Key-management field 2, low-order byte (security history).
054, for *kuf* > 2	01	Key-management field 3, high-order byte (pedigree original). Release 4.2 or later.
055, for *kuf* > 2	01	Key-management field 3, low-order byte (pedigree current). Release 4.2 or later.
050 + (2 * *kmf*)	*kl*	Optional key label.
050 + (2 * *kmf*) + *kl*	*iead*	Optional IBM extended associated data (unused).
*050 + (2 * kmf) + kl + iead*	*uad*	Optional user-defined associated data.
End of associated data section		
Optional clear key payload, wrapped AESKW formatted payload, or wrapped PKOAEP2 encoded payload (no payload if no key present)		

Table 237. HMAC MAC variable-length symmetric key-token, version X'05' (CCA 4.1.0 or later) (continued)

Offset (bytes)	Length (bytes)	Description		
050+ *(2 * kmf) + kl + iead + uad*	(*pl* + 7) / 8	Contents of payload (*pl* is in *bits*) depending on the encrypted section key-wrapping method (value at offset 26):		
		Value at offset 26	**Encrypted section key-wrapping method**	**Meaning**
		X'00'	No key-wrapping method. Only applies when key is clear, that is, when key material state (value at offset 8) is X'01'.	Only the key material will be in the payload. The key token is external or internal.
		X'02'	AESKW	An encrypted payload which the Segment 2 code creates by wrapping the unencrypted AESKW formatted payload. The payload is made up of the integrity check value, pad length, length of hash options and hash, hash options, hash of the associated data, key material, and padding. The key token is internal.
		X'03'	PKOAEP2	An encrypted PKOAEP2 encoded payload created using the RSAES-OAEP scheme of the PKCS #1 v2.1 standard. The message *M* is encoded for a given hash algorithm using the Bellare and Rogaway Optimal Asymmetric Encryption Padding (OAEP) method for encoding messages. For PKAOEP2, *M* is defined as follows: *M* = [32 bytes: *hAD*] ‖ [2 bytes: bit length of the clear key] ‖ [clear key] where *hAD* is the message digest of the associated data, and is calculated using the SHA-256 algorithm starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). The encoded message is wrapped with an RSA public-key according to the standard. The key token is external.
End of optional clear key payload, wrapped AESKW formatted payload, or wrapped PKOAEP2 encoded payload				
End of clear key, AESKW, or PKOAEP2 components				
Note: All numbers are in big endian format.				

AES EXPORTER and IMPORTER variable-length symmetric key token

View a table showing the format of the AES EXPORTER and IMPORTER variable-length symmetric key-token.

Table 238. AES EXPORTER and IMPORTER variable-length symmetric key-token, version X'05'.

Offset (bytes)	Length (bytes)	Description
Header		

Table 238. AES EXPORTER and IMPORTER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
000	01	Token identifier: **Value** **Meaning** **X'01'** Internal key-token (encrypted key is wrapped with the master key or there is no payload). **X'02'** External key-token (encrypted payload is wrapped with a transport key or there is no payload). A transport key can be a key-encrypting key or an RSA public-key. All unused values are reserved and undefined.
001	01	Reserved, binary zero.
002	02	Length in bytes of the overall token structure: 46 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead* + *uad* + ((*pl* + 7) / 8) **Key token** **Minimum token length** **Skeleton** 46 + (2 * 4) + (2 * 3) + 0 + 0 + 0 + 0 = 60 **Encrypted V0 payload** 46 + (2 * 4) + (2 * 3) + 0 + 0 + 0 + ((512 + 7) / 8) = 124 **Encrypted V1 payload** 46 + (2 * 4) + (2 * 3) + 0 + 0 + 0 + ((640 + 7) / 8) = 140 **Key token** **Maximum token length** **External*** 46 + (2 * 4) + (2 * 3) + 64 + 0 + 255 + ((4096 + 7) / 8) = 891 **Internal** 46 + (2 * 4) + (2 * 3) + 64 + 0 + 255 + ((640 + 7) / 8) = 459 *This assumes a PKOAEP2 key-wrapping method using a 4096-bit RSA transport key.
004	01	Token version number (identifies the format of this key token): **Value** **Meaning** **X'05'** Version 5 format of the key token (variable-length symmetric key-token)
005	03	Reserved, binary zero.
End of header		
Wrapping information section (all data related to wrapping the key)		
008	01	Key material state: **Value** **Meaning** **X'00'** No key is present. This is called a skeleton key-token. The key token is external or internal. **X'02'** Key is wrapped with a transport key. When the encrypted section key-wrapping method is AESKW (value at offset 26 is X'02'), the transport key is an AES key-encrypting key. When it is PKOAEP2 (value at offset 26 is X'03'), the transport key is an RSA public-key. The key token is external. **X'03'** Key is wrapped with the AES master-key. The encrypted section key-wrapping method is AESKW. The key token is internal. All unused values are reserved and undefined.

Table 238. AES EXPORTER and IMPORTER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
009	01	Key verification pattern (KVP) type: **Value** **Meaning** **X'00'** No KVP (no key present or key is wrapped with an RSA public-key). The key token is external or internal. **X'01'** AESMK (8 leftmost bytes of SHA-256 hash: X'01 ‖ clear AES MK). The key token is internal. **X'02'** KEK (8 leftmost bytes of SHA-256 hash: X'01 ‖ clear KEK). The key token is external. All unused values are reserved and undefined.
010	16	KVP (value depends on value of key material state, that is, the value at offset 8): **Value at offset 8** **Value of KVP** **X'00'** The key-material state is no key present. The field should be filled with binary zeros. The key token is external or internal. **X'02'** The key material state is the key is wrapped with a transport key. The value of the KVP depends on the value of the encrypted section key-wrapping method: • When the key-wrapping method is AESKW (value at offset 26 is X'02'), the field contains the KVP of the key-encrypting key used to wrap the key. The 8-byte KEK KVP is left-aligned in the field and padded on the right low-order bytes with binary zeros. • When the key-wrapping method is PKOAEP2 (value at offset 26 is X'03'), the value should be filled with binary zeros. The encoded message, which contains the key, is wrapped with an RSA public-key. The key token is external. **X'03'** The key-material state is the key is wrapped with the AES master-key. The field contains the MKVP of the AES master-key used to wrap the key. The 8-byte MKVP is left-aligned in the field and padded on the right low-order bytes with binary zeros. The key token is internal.
026	01	Encrypted section key-wrapping method (how data in the encrypted section is protected): **Value** **Meaning** **X'00'** No key-wrapping method (no key present). The key token is external or internal. **X'02'** AESKW (ANS X9.102). The key token is external with a key wrapped by an AES key-encrypting key, or the key token is internal with a key wrapped by the AES master-key. **X'03'** PKOAEP2. Message *M*, which contains the key, is encoded using the RSAES-OAEP scheme of the RSA PKCS #1 v2.1 standard. The encoded message (*EM*) is produced using the given hash algorithm by encoding message *M* using the Bellare and Rogaway Optimal Asymmetric Encryption Padding (OAEP) method for encoding messages. For PKAOEP2, *M* is defined as follows: *M* = [32 bytes: *hAD*] ‖ [2 bytes: bit length of the clear key] ‖ [clear key] where *hAD* is the message digest of the associated data, and is calculated using the SHA-256 algorithm on the data starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). *EM* is wrapped with an RSA public-key. The key token is external. All unused values are reserved and undefined.

Table 238. AES EXPORTER and IMPORTER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
027	01	Hash algorithm used for wrapping key or encoding message. Meaning depends on whether the encrypted section key-wrapping method (value at offset 26) is no key-wrapping method, AESKW, or PKOAEP2: No key-wrapping method (value at offset 26 is X'00') Hash algorithm used for wrapping key when encrypted section key-wrapping method is no key-wrapping method: **Value** **Meaning** **X'00'** No hash (no key present) All unused values are reserved and undefined. The key token is external or internal. AESKW key-wrapping method (value at offset 26 is X'02') Hash algorithm used for wrapping key when encrypted section key-wrapping method is AESKW. The value indicates the algorithm used to calculate the message digest of the associated data. The message digest is included in the wrapped payload and is calculated starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). **Value** **Meaning** **X'02'** SHA-256 All unused values are reserved and undefined. The key token is external or internal. PKOAEP2 key-wrapping method (value at offset 26 is X'03') Hash algorithm used for encoding message when encrypted section key-wrapping method is PKOAEP2. The value indicates the given hash algorithm used for encoding message *M* using the RSAES-OAEP scheme of the RSA PKCS #1 v2.1 standard. **Value** **Meaning** **X'01'** SHA-1 **X'02'** SHA-256 **X'04'** SHA-384 **X'08'** SHA-512 All unused values are reserved and undefined. The key token is external.

Table 238. AES EXPORTER and IMPORTER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
028	01	Payload format version (identifies format of the payload). Release 4.4 or later, otherwise undefined. **Value** **Meaning** **X'00'** V0 payload (V0PYLD). The payload format depends on the encrypted section key-wrapping method (value at offset 26): **Value at offset 26** **Meaning** **X'00'** There is no key-wrapping method. When no key is present, there is no payload. The key token is external or internal. **X'02'** The key-wrapping method is AESKW and the payload is variable length. The payload is formatted with the minimum size possible to contain the key material. The payload length varies for a given algorithm and key type. The key length can be inferred by the size of the payload. The key token is external or internal. **X'03'** The key-wrapping method is PKOAEP2 and the payload length is equal to the modulus size in bits of the RSA transport key used to wrap the encoded message. The key token is external. When the external key is exported, the internal target key will have the same V0 payload format. **X'01'** V1 payload (Release 4.4 or later). The payload format depends on the encrypted section key-wrapping method (value at offset 26): **Value at offset 26** **Meaning** **X'00'** There is no key-wrapping method. When no key is present, there is no payload. The key token is external or internal. **X'02'** The key-wrapping method is AESKW and the payload is fixed length based on the maximum possible key size of the algorithm for the key. The key is padded with random data to the size of the largest key for that algorithm. This helps to deter attacks on keys known to be weaker. The key length cannot be inferred by the size of the payload. The key token is external or internal. **X'03'** The key-wrapping method is PKOAEP2 and the payload length is equal to the modulus size in bits of the RSA transport key used to wrap the encoded message. The key token is external. When the external key is exported, the internal target key will have the same V1 payload format. All unused values are reserved and undefined.
029	01	Reserved, binary zero.
End of wrapping information section		
AESKW or PKOAEP2 components: (1) associated data section and (2) optional wrapped AESKW formatted payload or wrapped PKOAEP2 encoded payload (no payload if no key present)		
Associated data section		
030	01	Associated data section version: **Value** **Meaning** **X'01'** Version 1 format of associated data
031	01	Reserved, binary zero.
032	02	Length in bytes of all the associated data for the key token: 30 - 349.
034	01	Length in bytes of the optional key label (*kl*): 0 or 64.
035	01	Length in bytes of the optional IBM extended associated data (*iead*): 0.
036	01	Length in bytes of the optional user-definable associated data (*uad*): 0 - 255.

Table 238. AES EXPORTER and IMPORTER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
037	01	Reserved, binary zero.
038	02	Length in *bits* of the wrapped payload (*pl*): 0, 512 - 4096. • For no key-wrapping method (no key present), *pl* is 0. • For PKOAEP2 encoded payloads, *pl* is the length in bits of the modulus size of the RSA key used to wrap the payload. This can be 512 - 4096. • For an AESKW formatted payload, *pl* is based on the key size of the algorithm type and the payload format version: • AES algorithm (value at offset 41 is X'02') • An AES key can have a length of 16, 24, or 32 bytes (128, 192, or 256 bits). The following table shows the payload length for a given AES key size and payload format: **AES key size** / **Bit length of V0 payload (value at offset 28 is X'00')** / **Bit length of V1 payload (value at offset 28 is X'01')** 16 bytes (128 bits): 512 / 640 24 bytes (192 bits): 576 / 640 32 bytes (256 bits): 640 / 640
040	01	Reserved, binary zero.
041	01	Algorithm type (algorithm for which the key can be used): **Value** **Meaning** **X'02'** AES All unused values are reserved and undefined.
042	02	Key type (general class of the key): **Value** **Meaning** **X'0003'** EXPORTER **X'0004'** IMPORTER All unused values are reserved and undefined.
044	01	Key usage fields count (*kuf*): 4. Key-usage field information defines restrictions on the use of the key. For key type EXPORTER, see AES EXPORTER Key Token Build2 keywords (Figure 7 on page 230). For key type IMPORTER, see AES IMPORTER Key Token Build2 keywords (Figure 8 on page 234). Each key-usage field is 2 bytes in length. The value in this field indicates how many 2-byte key usage fields follow.

Table 238. AES EXPORTER and IMPORTER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
045 (1 of 2)	01	Key-usage field 1, high-order byte (KEK control). The meaning is determined by the key type (value at offset 42). The key type can be EXPORTER or IMPORTER. **EXPORTER (value at offset 42 is X'0003')** **Value Meaning** **B'1xxx xxxx'** Key can be used to export a key (EXPORT). **B'0xxx xxxx'** Key cannot be used to export a key. **B'x1xx xxxx'** Key can be used to translate a key (TRANSLAT). **B'x0xx xxxx'** Key cannot be used to translate a key. **B'xx1x xxxx'** Key can be used by KGN2 for generating an OPEX key pair (GEN-OPEX). **B'xx0x xxxx'** Key cannot be used by KGN2 for generating an OPEX key pair. **B'xxx1 xxxx'** Key can be used by KGN2 for generating an IMEX key pair (GEN-IMEX). **B'xxx0 xxxx'** Key cannot be used by KGN2 for generating an IMEX key pair. **B'xxxx 1xxx'** Key can be used by KGN2 for generating an EXEX key pair (GEN-EXEX). **B'xxxx 0xxx'** Key cannot be used by KGN2 for generating an EXEX key pair. **B'xxxx x1xx'** Key can be used by PKG for generating an ECC public-private key pair (GEN-PUB). **B'xxxx x0xx'** Key cannot be used by PKG for generating an ECC public-private key pair (GEN-PUB). **Note:** At least one defined bit must be B'1'. All unused bits are reserved and must be zero.

Table 238. AES EXPORTER and IMPORTER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
045 (2 of 2)	01	**IMPORTER (value at offset 42 is X'0004')** **Value Meaning** **B'1xxx xxxx'** Key can be used to import a key (IMPORT). **B'0xxx xxxx'** Key cannot be used to import a key. **B'x1xx xxxx'** Key can be used to translate a key (TRANSLAT). **B'x0xx xxxx'** Key cannot be used to translate a key. **B'xx1x xxxx'** Key can be used by KGN2 for generating an OPIM key pair (GEN-OPIM). **B'xx0x xxxx'** Key cannot be used by KGN2 for generating an OPIM key pair. **B'xxx1 xxxx'** Key can be used by KGN2 for generating an IMEX key pair (GEN-IMEX). **B'xxx0 xxxx'** Key cannot be used by KGN2 for generating an IMEX key pair. **B'xxxx 1xxx'** Key can be used by KGN2 for generating an IMIM key pair (GEN-IMIM). **B'xxxx 0xxx'** Key cannot be used by KGN2 for generating an IMIM key pair. **B'xxxx x1xx'** Key can be used by PKG for generating an ECC public-private key pair (GEN-PUB). **B'xxxx x0xx'** Key cannot be used by PKG for generating an ECC public-private key pair (GEN-PUB). **Note:** At least one defined bit must be B'1'. All unused bits are reserved and must be zero.
046	01	**Key-usage field 1, low-order byte** (user-defined extension control).
047	01	**Key-usage field 2, high-order byte** (TR-31 wrap control): **Value Meaning** **B'1xxx xxxx'** Key can wrap or unwrap a TR-31 key (WR-TR31). Defined for future use. **B'0xxx xxxx'** Key cannot wrap or unwrap a TR-31 key. Defined for future use. All unused bits are reserved and must be zero.
048	01	**Key-usage field 2, low-order byte** (raw key wrap control): **Value Meaning** **B'xxxx xxx1'** Key can wrap or unwrap a raw key (KEK-RAW). Defined for future use. **B'0xxx xxxx'** Key cannot wrap or unwrap a raw key. Defined for future use. All unused bits are reserved and must be zero.

Table 238. AES EXPORTER and IMPORTER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
049	01	**Key-usage field 3, high-order byte** (algorithm wrap control): **Value Meaning** **B'1xxx xxxx'** Key can wrap or unwrap DES keys (WR-DES). **B'0xxx xxxx'** Key cannot wrap or unwrap DES keys. **B'x1xx xxxx'** Key can wrap or unwrap AES keys (WR-AES). **B'x0xx xxxx'** Key cannot wrap or unwrap AES keys. **B'xx1x xxxx'** Key can wrap or unwrap HMAC keys (WR-HMAC). **B'xx0x xxxx'** Key cannot wrap or unwrap HMAC keys. **B'xxx1 xxxx'** Key can wrap or unwrap RSA keys (WR-RSA). **B'xxx0 xxxx'** Key cannot wrap or unwrap RSA keys. **B'xxxx 1xxx'** Key can wrap or unwrap ECC keys (WR-ECC). **B'xxxx 0xxx'** Key cannot wrap or unwrap ECC keys. **Note:** At least one defined bit must be B'1'. All unused bits are reserved and must be zero.
050	01	**Key-usage field 3, low-order byte** (reserved). All bits are reserved and must be zero.

Table 238. AES EXPORTER and IMPORTER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
051	01	**Key-usage field 4, high-order byte** (class wrap control). **Value Meaning** **B'1xxx xxxx'** Key can wrap or unwrap data class keys (WR-DATA). **B'0xxx xxxx'** Key cannot wrap or unwrap data class keys. **B'x1xx xxxx'** Key can wrap or unwrap KEK class keys (WR-KEK). **B'x0xx xxxx'** Key cannot wrap or unwrap KEK class keys. **B'xx1x xxxx'** Key can wrap or unwrap PIN class keys (WR-PIN). **B'xx0x xxxx'** Key cannot wrap or unwrap PIN class keys. **B'xxx1 xxxx'** Key can wrap or unwrap derivation class keys (WRDERIVE). **B'xxx0 xxxx'** Key cannot wrap or unwrap derivation class keys. **B'xxxx 1xxx'** Key can wrap or unwrap card class keys (WR-CARD). **B'xxxx 0xxx'** Key cannot wrap or unwrap card class keys. **B'xxxx x1xx'** Key can wrap or unwrap cryptovariable class keys (WR-CVAR). Undefined in releases before Release 4.4. **B'xxxx x0xx'** Key cannot wrap or unwrap cryptovariable class keys. Undefined in releases before Release 4.4. **Note:** At least one defined bit must be B'1'. All unused values are reserved and undefined.
052	01	**Key-usage field 4, low-order byte** (reserved). All bits are reserved and must be zero.
053	01	Key management fields count (*kmf*): 3. Key-management field information describes how the data is to be managed or helps with management of the key material. For key type EXPORTER, see AES EXPORTER Key Token Build2 keywords (Figure 7 on page 230). For key type IMPORTER, see AES IMPORTER Key Token Build2 keywords (Figure 8 on page 234). Each key-management field is 2 bytes in length. The value in this field indicates how many 2-byte key management fields follow.
054	01	Key-management field 1, high-order byte (symmetric-key export control).
055	01	Key-management field 1, low-order byte (export control by algorithm).
056	01	Key-management field 2, high-order byte (key completeness).
057	01	Key-management field 2, low-order byte (security history).
058	01	Key-management field 3, high-order byte (pedigree original).
059	01	Key-management field 3, low-order byte (pedigree current).
060	*kl*	Optional key label.

Table 238. AES EXPORTER and IMPORTER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description		
060 + *kl*	*iead*	Optional IBM extended associated data (unused).		
060 + *kl* + *iead*	*uad*	Optional user-defined associated data.		
End of associated data section				
Optional wrapped AESKW formatted payload or wrapped PKOAEP2 encoded payload (no payload if no key present)				
060 + *kl* + *iead* + *uad*	(*pl* + 7) / 8	Contents of payload (*pl* is in *bits*) depending on the encrypted section key-wrapping method (value at offset 26):		
		Value at offset 26	**Encrypted section key-wrapping method**	**Meaning**
		X'02'	AESKW	An encrypted payload which the Segment 2 code creates by wrapping the unencrypted AESKW formatted payload. The payload is made up of the integrity check value, pad length, length of hash options and hash, hash options, hash of the associated data, key material, and padding. The key token is internal.
		X'03'	PKOAEP2	An encrypted PKOAEP2 encoded payload created using the RSAES-OAEP scheme of the PKCS #1 v2.1 standard. The message *M* is encoded for a given hash algorithm using the Bellare and Rogaway Optimal Asymmetric Encryption Padding (OAEP) method for encoding messages. For PKAOEP2, *M* is defined as follows: *M* = [32 bytes: *hAD*] ‖ [2 bytes: bit length of the clear key] ‖ [clear key] where *hAD* is the message digest of the associated data, and is calculated using the SHA-256 algorithm starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). The encoded message is wrapped with an RSA public-key according to the standard. The key token is external.
End of optional wrapped AESKW formatted payload or wrapped PKAOEP2 encoded payload				
End of AESKW or PKOAEP2 components				
Note: All numbers are in big endian format.				

AES PINPROT, PINCALC, and PINPRW variable-length symmetric key token

View a table showing the format of the PINPROT, PINCALC, and PINPRW variable-length symmetric key-tokens.

Table 239. AES CIPHER variable-length symmetric key-token, version X'05'.

Offset (bytes)	Length (bytes)	Description
Header		

Table 239. AES CIPHER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
000	01	Token identifier: **Value** **Meaning** **X'01'** Internal key-token (encrypted key is wrapped with the master key or there is no payload). **X'02'** External key-token (encrypted payload is wrapped with a transport key or there is no payload). A transport key can be a key-encrypting key or an RSA public-key. All unused values are reserved and undefined.
001	01	Reserved, binary zero.
002	02	Length in bytes of the overall token structure: 46 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead* + *uad* + ((*pl* + 7) / 8) **Key token** **Minimum token length** **Skeleton** 46 + (2 * 3) + (2 * 3) + 0 + 0 + 0 + 0 = 58 **Encrypted V1 payload** 46 + (2 * 3) + (2 * 3) + 0 + 0 + 0 + ((640 + 7) / 8) = 138 **Key token** **Maximum token length** **External*** 46 + (2 * 3) + (2 * 3) + 64 + 0 + 255 + ((4096 + 7) / 8) = 889 **Internal** 46 + (2 * 3) + (2 * 3) + 64 + 0 + 255 + ((640 + 7) / 8) = 457 *This assumes a PKOAEP2 key-wrapping method using a 4096-bit RSA transport key.
004	01	Token version number (identifies the format of this key token): **Value** **Meaning** **X'05'** Version 5 format of the key token (variable-length symmetric key-token)
005	03	Reserved, binary zero.
End of header		
Wrapping information section (all data related to wrapping the key)		
008	01	Key material state: **Value** **Meaning** **X'00'** No key is present. This is called a skeleton key-token. The key token is external or internal. **X'02'** Key is wrapped with a transport key. When the encrypted section key-wrapping method is AESKW (value at offset 26 is X'02'), the transport key is an AES key-encrypting key. When it is PKOAEP2 (value at offset 26 is X'03'), the transport key is an RSA public-key. The key token is external. **X'03'** Key is wrapped with the AES master-key. The encrypted section key-wrapping method is AESKW. The key token is internal. All unused values are reserved and undefined.

Table 239. AES CIPHER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
009	01	Key verification pattern (KVP) type: **Value** **Meaning** **X'00'** No KVP (no key present or key is wrapped with an RSA public-key). The key token is external or internal. **X'01'** AESMK (8 leftmost bytes of SHA-256 hash: X'01 ‖ clear AES MK). The key token is internal. **X'02'** KEK (8 leftmost bytes of SHA-256 hash: X'01 ‖ clear KEK). The key token is external. All unused values are reserved and undefined.
010	16	KVP (value depends on value of key material state, that is, the value at offset 8): **Value at offset 8** **Value of KVP** **X'00'** The key-material state is no key present. The field should be filled with binary zeros. The key token is external or internal. **X'02'** The key material state is the key is wrapped with a transport key. The value of the KVP depends on the value of the encrypted section key-wrapping method: • When the key-wrapping method is AESKW (value at offset 26 is X'02'), the field contains the KVP of the key-encrypting key used to wrap the key. The 8-byte KEK KVP is left-aligned in the field and padded on the right low-order bytes with binary zeros. • When the key-wrapping method is PKOAEP2 (value at offset 26 is X'03'), the value should be filled with binary zeros. The encoded message, which contains the key, is wrapped with an RSA public-key. The key token is external. **X'03'** The key-material state is the key is wrapped with the AES master-key. The field contains the MKVP of the AES master-key used to wrap the key. The 8-byte MKVP is left-aligned in the field and padded on the right low-order bytes with binary zeros. The key token is internal.
026	01	Encrypted section key-wrapping method (how data in the encrypted section is protected): **Value** **Meaning** **X'00'** No key-wrapping method (no key present). The key token is external or internal. **X'02'** AESKW (ANS X9.102). The key token is external with a key wrapped by an AES key-encrypting key, or the key token is internal with a key wrapped by the AES master-key. **X'03'** PKOAEP2. Message *M*, which contains the key, is encoded using the RSAES-OAEP scheme of the RSA PKCS #1 v2.1 standard. The encoded message (*EM*) is produced using the given hash algorithm by encoding message *M* using the Bellare and Rogaway Optimal Asymmetric Encryption Padding (OAEP) method for encoding messages. For PKAOEP2, *M* is defined as follows: *M* = [32 bytes: *hAD*] ‖ [2 bytes: bit length of the clear key] ‖ [clear key] where *hAD* is the message digest of the associated data, and is calculated using the SHA-256 algorithm on the data starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). *EM* is wrapped with an RSA public-key. The key token is external. All unused values are reserved and undefined.

Table 239. AES CIPHER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
027	01	Hash algorithm used for wrapping key or encoding message. Meaning depends on whether the encrypted section key-wrapping method (value at offset 26) is no key-wrapping method, AESKW, or PKOAEP2: No key-wrapping method (value at offset 26 is X'00') Hash algorithm used for wrapping key when encrypted section key-wrapping method is no key-wrapping method: **Value** **Meaning** **X'00'** No hash (no key present) All unused values are reserved and undefined. The key token is external or internal. AESKW key-wrapping method (value at offset 26 is X'02') Hash algorithm used for wrapping key when encrypted section key-wrapping method is AESKW. The value indicates the algorithm used to calculate the message digest of the associated data. The message digest is included in the wrapped payload and is calculated starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). **Value** **Meaning** **X'02'** SHA-256 All unused values are reserved and undefined. The key token is external or internal. PKOAEP2 key-wrapping method (value at offset 26 is X'03') Hash algorithm used for encoding message when encrypted section key-wrapping method is PKOAEP2. The value indicates the given hash algorithm used for encoding message *M* using the RSAES-OAEP scheme of the RSA PKCS #1 v2.1 standard. **Value** **Meaning** **X'01'** SHA-1 **X'02'** SHA-256 **X'04'** SHA-384 **X'08'** SHA-512 All unused values are reserved and undefined. The key token is external.

Table 239. AES CIPHER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
028	01	Payload format version (identifies format of the payload): **Value** **Meaning** **X'01'** V1 payload (V0PYLD). The payload format depends on the encrypted section key-wrapping method (value at offset 26): **Value at offset 26** **Meaning** **X'00'** There is no key-wrapping method. When no key is present, there is no payload. The key token is external or internal. **X'02'** The key-wrapping method is AESKW and the payload is fixed length based on the maximum possible key size of the algorithm for the key. The key is padded with random data to the size of the largest key for that algorithm. This helps to deter attacks on keys known to be weaker. The key length cannot be inferred by the size of the payload. The key token is external or internal. **X'03'** The key-wrapping method is PKOAEP2 and the payload length is equal to the modulus size in bits of the RSA transport key used to wrap the encoded message. The key token is external. When the external key is exported, the internal target key will have the same V1 payload format. All unused values are reserved and undefined.
029	01	Reserved, binary zero.
End of wrapping information section		
AESKW or PKOAEP2 components: (1) associated data section and (2) optional wrapped AESKW formatted payload or wrapped PKOAEP2 encoded payload (no payload if no key present)		
Associated data section		
030	01	Associated data section version: **Value** **Meaning** **X'01'** Version 1 format of associated data
031	01	Reserved, binary zero.
032	02	Length in bytes of all the associated data for the key token: 16 - 347.
034	01	Length in bytes of the optional key label (*kl*): 0 or 64.
035	01	Length in bytes of the optional IBM extended associated data (*iead*): 0.
036	01	Length in bytes of the optional user-definable associated data (*uad*): 0 - 255.
037	01	Reserved, binary zero.

Table 239. AES CIPHER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
038	02	Length in *bits* of the wrapped payload (*pl*): 0, 512 - 4096. • For no key-wrapping method (no key present), *pl* is 0. • For PKOAEP2 encoded payloads, *pl* is the length in bits of the modulus size of the RSA key used to wrap the payload. This can be 512 - 4096. • For an AESKW formatted payload, *pl* is based on the key size of the algorithm type and the payload format version: AES algorithm (value at offset 41 is X'02') – An AES key can have a length of 16, 24, or 32 bytes (128, 192, or 256 bits). The following table shows the payload length for a given AES key size and payload format: **AES key size** / **Bit length of V0 payload (value at offset 28 is X'00')** / **Bit length of V1 payload (value at offset 28 is X'01')** 16 bytes (128 bits) / Not applicable / 640 24 bytes (192 bits) / Not applicable / 640 32 bytes (256 bits) / Not applicable / 640
040	01	Reserved, binary zero.
041	01	Algorithm type (algorithm for which the key can be used): **Value** **Meaning** **X'02'** AES All unused values are reserved and undefined.
042	02	Key type (general class of the key): **Value** **Meaning** **X'0005'** PINPROT **X'0006'** PINCALC **X'0007'** PINPRW All unused values are reserved and undefined.
044	01	Key usage fields count (*kuf*): 3. Key-usage field information defines restrictions on the use of the key. For key type PINPROT, see AES PINPROT Key Token Build2 keywords (Figure 12 on page 246). For key type PINCALC, see AES PINCALC Key Token Build2 keywords (Figure 11 on page 244). For key type PINPRW, see AES PINPRW Key Token Build2 keywords (Figure 13 on page 249). Each key-usage field is 2 bytes in length. The value in this field indicates how many 2-byte key usage fields follow.

Table 239. AES CIPHER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
045	01	Key-usage field 1, high-order byte. The meaning is determined by the key type (value at offset 42). The key type can be PINPROT, PINCALC, or PINPRW: **PINPROT (value at offset 42 is X'0005')** Encryption operation: **Value Meaning** **B'11xx xxxx'** Undefined or not used. **B'10xx xxxx'** Key can be used for encryption; key cannot be used for decryption (ENCRYPT). **B'01xx xxxx'** Key cannot be used for encryption; key can be used for decryption (DECRYPT). **B'00xx xxxx'** Undefined or not used. All unused bits are reserved and must be zero. PINCALC (value at offset 42 is X'0006') MAC operation: **Value Meaning** **B'1xxx xxxx'** Key can only be used for generate (GENONLY). **B'0xxx xxxx'** Undefined or not used. All unused bits are reserved and must be zero. **PINPRW (value at offset 42 is X'0007')** MAC operation: **Value Meaning** **B'11xx xxxx'** Undefined or not used. **B'10xx xxxx'** Key can be used for generate; key cannot be used for verify (GENONLY). **B'01xx xxxx'** Key cannot be used for generate; key can be used for verify (VERIFY). **B'00xx xxxx'** Undefined or not used. All unused bits are reserved and must be zero.
046	01	Key-usage field 1, low-order byte (user-defined extension control).

Table 239. AES CIPHER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
047	01	Key-usage field 2, high-order byte. The meaning is determined by the key type (value at offset 42). The key type can be PINPROT, PINCALC, or PINPRW: **PINPROT (value at offset 42 is X'0005')** Encryption mode: **Value Meaning** **X'00'** Key can be used for Cipher Block Chaining (CBC). All unused values are reserved and undefined. **PINCALC (value at offset 42 is X'0006')** Encryption mode: **Value Meaning** **X'00'** Key can be used for Cipher Block Chaining (CBC). All unused values are reserved and undefined. **PINPRW (value at offset 42 is X'0007')** MAC mode: **Value Meaning** **X'01'** CMAC mode (NIST SP 800-38B) All unused values are reserved and undefined.
048	01	Key-usage field 2, low-order byte: Reserved, binary zero.
049	01	**Key-usage field 3, high-order byte**. The meaning is determined by the field format identifier (value at offset 50). Currently the only field format identifier is DK enabled: DK enabled (value at offset 50 is X'01') Common control by key type, based on key type PINPROT, PINCALC, or PINPRW: **Key type (value at offset 42)** **Value Meaning PINPROT PINCALC PINPRW** **X'01'** PIN OP (DKPINOP) Valid Valid Valid **X'02'** PIN OPP (DKPINOPP) Valid Undefined Undefined **X'03'** PIN AD1 (DKPINAD1) Valid Undefined Undefined All unused values are reserved and undefined.
050	01	Key-usage field 3, low-order byte (field format identifier). Identifies the format of key-usage field 3, high-order byte (value at offset 49): **Value Meaning** **X'01'** DK enabled (set when **DKPINOP**, **DKPINOPP**, or **DKPINAD1** keyword is used) All unused values are reserved and undefined.

Table 239. AES CIPHER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
051	01	Key management fields count (*kmf*): 3. Key-management field information describes how the data is to be managed or helps with management of the key material. For key type PINPROT, see AES PINPROT Key Token Build2 keywords (Figure 12 on page 246). For key type PINCALC, see AES PINCALC Key Token Build2 keywords (Figure 11 on page 244). For key type PINPRW, see AES PINPRW Key Token Build2 keywords (Figure 13 on page 249). Each key-management field is 2 bytes in length. The value in this field indicates how many 2-byte key management fields follow.
052	01	Key-management field 1, high-order byte (symmetric-key export control).
053	01	Key-management field 1, low-order byte (export control by algorithm).
054	01	Key-management field 2, high-order byte (key completeness).
055	01	Key-management field 2, low-order byte (security history).
056	01	Key-management field 3, high-order byte (pedigree original).
057	01	Key-management field 3, low-order byte (pedigree current).
058	*kl*	Optional key label.
058 + *kl*	*iead*	Optional IBM extended associated data (unused).
058 + *kl* + *iead*	*uad*	Optional user-defined associated data.
End of associated data section		
Optional wrapped AESKW formatted payload or wrapped PKOAEP2 encoded payload (no payload if no key present)		

Table 239. AES CIPHER variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description		
058 + *kl* + *iead* + *uad*	(*pl* + 7) / 8	Contents of payload (*pl* is in *bits*) depending on the encrypted section key-wrapping method (value at offset 26):		
		Value at offset 26	**Encrypted section key-wrapping method**	**Meaning**
		X'02'	AESKW	An encrypted payload which the Segment 2 code creates by wrapping the unencrypted AESKW formatted payload. The payload is made up of the integrity check value, pad length, length of hash options and hash, hash options, hash of the associated data, key material, and padding. The key token is internal.
		X'03'	PKOAEP2	An encrypted PKOAEP2 encoded payload created using the RSAES-OAEP scheme of the PKCS #1 v2.1 standard. The message *M* is encoded for a given hash algorithm using the Bellare and Rogaway Optimal Asymmetric Encryption Padding (OAEP) method for encoding messages. For PKAOEP2, *M* is defined as follows: *M* = [32 bytes: *hAD*] ‖ [2 bytes: bit length of the clear key] ‖ [clear key] where *hAD* is the message digest of the associated data, and is calculated using the SHA-256 algorithm starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). The encoded message is wrapped with an RSA public-key according to the standard. The key token is external.
End of optional wrapped AESKW formatted payload or wrapped PKAOEP2 encoded payload				
End of AESKW or PKOAEP2 components				
Note: All numbers are in big endian format.				

AES DESUSECV variable-length symmetric key token

View a table showing the format of the DESUSECV variable-length symmetric key-token.

Table 240. AES DESUSECV variable-length symmetric key-token, version X'05'.

Offset (bytes)	Length (bytes)	Description
Header		
000	01	Token identifier: **Value** **Meaning** **X'02'** External key-token (encrypted payload is wrapped with a transport key or there is no payload). A transport key can be a key-encrypting key or an RSA public-key. All unused values are reserved and undefined.
001	01	Reserved, binary zero.

Table 240. AES DESUSECV variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
002	02	Length in bytes of the overall token structure: 46 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead* + *uad* + ((*pl* + 7) / 8) Key token Minimum and maximum token length External 46 + (2 * 1) + (2 * 1) + 0 + 11 + 0 + ((576 + 7) / 8) = 133
004	01	Token version number (identifies the format of this key token): **Value** **Meaning** **X'05'** Version 5 format of the key token (variable-length symmetric key-token)
005	03	Reserved, binary zero.
End of header		
Wrapping information section (all data related to wrapping the key)		
008	01	Key material state: **Value** **Meaning** **X'02'** Key is wrapped with a transport key. The transport key is an AES key-encrypting key. The key token is external. All unused values are reserved and undefined.
009	01	Key verification pattern (KVP) type: **Value** **Meaning** **X'02'** KEK (8 leftmost bytes of SHA-256 hash: X'01 ‖ clear KEK). The key token is external. All unused values are reserved and undefined.
010	16	KVP: The field contains the KVP of the key-encrypting key used to wrap the key. The 8-byte KEK KVP is left-aligned in the field and padded on the right low-order bytes with binary zeros. The key token is external.
026	01	Encrypted section key-wrapping method (how data in the encrypted section is protected): **Value** **Meaning** **X'02'** AESKW (ANS X9.102). The key token is external with a key wrapped by an AES key-encrypting key. All unused values are reserved and undefined.
027	01	Hash algorithm used for wrapping key. The value indicates the algorithm used to calculate the message digest of the associated data. The message digest is included in the wrapped payload and is calculated starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). **Value** **Meaning** **X'02'** SHA-256 All unused values are reserved and undefined. The key token is external.

Table 240. AES DESUSECV variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
028	01	Payload format version (identifies format of the payload): **Value** **Meaning** **X'01'** V1 payload (V1PYLD). The payload is fixed length based on the maximum possible key size of the algorithm for the key. The key is padded with random data to the size of the largest key for that algorithm. This helps to deter attacks on keys known to be weaker. The key length cannot be inferred by the size of the payload. The key token is external or internal. All unused values are reserved and undefined.
029	01	Reserved, binary zero.
End of wrapping information section		
AESKW components: (1) associated data section and (2) optional wrapped AESKW payload		
Associated data section		
030	01	Associated data section version: **Value** **Meaning** **X'01'** Version 1 format of associated data
031	01	Reserved, binary zero.
032	02	Length in bytes of all the associated data for thekey token: 31.
034	01	Length in bytes of the optional key label (*kl*): 0.
035	01	Length in bytes of the optional IBM extended associated data (*iead*): 11.
036	01	Length in bytes of the optional user-definable associated data (*uad*): 0.
037	01	Reserved, binary zero.
038	02	Length in *bits* of the wrapped payload (*pl*): 576. For an AESKW formatted payload, *pl* is based on the key size of the algorithm type and the payload format version: DES algorithm (value at offset 41 is X'01') A DES key can have a length of 8, 16, or 24 bytes (64, 128, 192 bits). A DES key in an AESKW formatted payload is always wrapped with a V1 payload and has a fixed length payload of 576 bits.
040	01	Reserved, binary zero.
041	01	Algorithm type (algorithm for which the key can be used): **Value** **Meaning** **X'01'** DES All unused values are reserved and undefined.
042	02	Key type (general class of the key): **Value** **Meaning** **X'0008'** DESUSECV All unused values are reserved and undefined.
044	01	Key usage fields count (*kuf*): 1. Key-usage field information defines restrictions on the use of the key. Each key-usage field is 2 bytes in length. The value in this field indicates how many 2-byte key usage fields follow.

Table 240. AES DESUSECV variable-length symmetric key-token, version X'05' (continued).

<table>
<tr><th>Offset (bytes)</th><th>Length (bytes)</th><th>Description</th></tr>
<tr><td>045</td><td>01</td><td>Key-usage field 1, high-order byte (reserved).

All bits are reserved and must be zero.</td></tr>
<tr><td>046</td><td>01</td><td>Key-usage field 1, low-order byte (reserved).

All bits are reserved and must be zero.</td></tr>
<tr><td>047</td><td>01</td><td>Key management fields count (kmf): 1. Key-management field information describes how the data is to be managed or helps with management of the key material.

Each key-management field is 2 bytes in length. The value in this field indicates how many 2-byte key management fields follow.</td></tr>
<tr><td>048</td><td>01</td><td>Key-management field 1, high-order byte (reserved).

All bits are reserved and must be zero.</td></tr>
<tr><td>049</td><td>01</td><td>Key-management field 1, low-order byte (reserved).

All bits are reserved and must be zero.</td></tr>
<tr><td>050</td><td>iead</td><td>IBM extended associated data:
<table>
<tr><th>Offset (bytes)</th><th>Length (bytes)</th><th>Item</th><th>Contents</th></tr>
<tr><td>0</td><td>1</td><td>Structure version identifier</td><td>X'00'</td></tr>
<tr><td>1</td><td>1</td><td>Flag byte 1</td><td>Copy of flag byte 1 taken from offset 6 of the internal fixed-length DES source key token exported by the CSNDSYX verb into this external enciphered key.</td></tr>
<tr><td>2</td><td>1</td><td>Flag byte 2</td><td>Copy of flag byte 2 taken from offset 7 of the internal fixed-length DES source key token exported by the CSNDSYX verb into this external enciphered key.</td></tr>
<tr><td>3</td><td>0</td><td>Masked control vector</td><td>Copy of 8 bytes of control vector taken from offset 32 of the internal fixed-length DES source key token exported by the CSNDSYX verb into this external enciphered key. The key form bits (CV bits 40 - 42) are copied into the wrapped payload before being masked here to zero to conceal the length of the DES key.</td></tr>
</table></td></tr>
<tr><td colspan="3">End of associated data section</td></tr>
<tr><td colspan="3">Optional wrapped AESKW formatted payload</td></tr>
</table>

Table 240. AES DESUSECV variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
061	72	Contents of payload: An encrypted payload which the Segment 2 code creates by wrapping the unencrypted AESKW formatted payload. The payload is made up of the integrity check value, pad length, length of hash options and hash, hash options, hash of the associated data, key material, and padding. The key token is internal. A DES DESUSECV payload contains key material that is formatted. The key material is formatted as follows: **Offset bytes) / Length (bytes) / Item / Contents** 0 / 1 / Key length (kl) / Value Meaning X'08' Single-length key X'10' Double-length key X'18' Triple-length key (System z only) 1 / 1 / Flag byte 1 / Reserved, binary zero. 2 / 1 / Flag byte 2 / Value Meaning B'xxxx xxx1' The key has guaranteed unique halves B'xxxx xxx0' The key does not have guaranteed unique halves. All unused bits are reserved and must be zero. 3 / 8 / Left part of key / Left part of single-length, double-length, or (System z only) triple-length DES key. 11 / 8 / Middle part of key, or random data / Middle part of double-length or (System z only) triple-length DES key, otherwise random data. 19 / 8 / Right part of key. / Right part of triple-length key (System z only), otherwise random data.
End of optional wrapped AESKW formatted payload		
End of AESKW components		
Note: All numbers are in big endian format.		

AES DKYGENKY variable-length symmetric key token

View a table showing the format of the DKYGENKY variable-length symmetric key-token.

Table 241. AES DKYGENKY variable-length symmetric key-token, version X'05'.

Offset (bytes)	Length (bytes)	Description
Header		
000	01	Token identifier: **Value** **Meaning** **X'01'** Internal key-token (encrypted key is wrapped with the master key or there is no payload). **X'02'** External key-token (encrypted payload is wrapped with a transport key or there is no payload). A transport key can be a key-encrypting key or an RSA public-key. All unused values are reserved and undefined.
001	01	Reserved, binary zero.

Table 241. AES DKYGENKY variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
002	02	Length in bytes of the overall token structure: 46 + (2 * *kuf*) + (2 * *kmf*) + *kl* + *iead* + *uad* + ((*pl* + 7) / 8) **Key token** **Minimum token length** **Skeleton** 46 + (2 * 2) + (2 * 3) + 0 + 0 + 0 + 0 = 56 **Encrypted V1 payload** 46 + (2 * 2) + (2 * 3) + 0 + 0 + 0 + ((640 + 7) / 8) = 136 **Key token** **Maximum token length** **External*** 46 + (2 * 6) + (2 * 3) + 64 + 0 + 255 + ((4096 + 7) / 8) = 895 **Internal** 46 + (2 * 6) + (2 * 3) + 64 + 0 + 255 + ((640 + 7) / 8) = 463 *This assumes a PKOAEP2 key-wrapping method using a 4096-bit RSA transport key.
004	01	Token version number (identifies the format of this key token): **Value** **Meaning** **X'05'** Version 5 format of the key token (variable-length symmetric key-token)
005	03	Reserved, binary zero.
End of header		
Wrapping information section (all data related to wrapping the key)		
008	01	Key material state: **Value** **Meaning** **X'00'** No key is present. This is called a skeleton key-token. The key token is external or internal. **X'02'** Key is wrapped with a transport key. When the encrypted section key-wrapping method is AESKW (value at offset 26 is X'02'), the transport key is an AES key-encrypting key. When it is PKOAEP2 (value at offset 26 is X'03'), the transport key is an RSA public-key. The key token is external. **X'03'** Key is wrapped with the AES master-key. The encrypted section key-wrapping method is AESKW. The key token is internal. All unused values are reserved and undefined.
009	01	Key verification pattern (KVP) type: **Value** **Meaning** **X'00'** No KVP (no key present or key is wrapped with an RSA public-key). The key token is external or internal. **X'01'** AESMK (8 leftmost bytes of SHA-256 hash: X'01 ⌂⌂¥ clear AES MK). The key token is internal. **X'02'** KEK (8 leftmost bytes of SHA-256 hash: X'01 ⌂⌂¥ clear KEK). The key token is external. All unused values are reserved and undefined.

Table 241. AES DKYGENKY variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
010	16	KVP (value depends on value of key material state, that is, the value at offset 8): **Value at offset 8** **Value of KVP** **X'00'** The key-material state is no key present. The field should be filled with binary zeros. The key token is external or internal. **X'02'** The key material state is the key is wrapped with a transport key. The value of the KVP depends on the value of the encrypted section key-wrapping method: • When the key-wrapping method is AESKW (value at offset 26 is X'02'), the field contains the KVP of the key-encrypting key used to wrap the key. The 8-byte KEK KVP is left-aligned in the field and padded on the right low-order bytes with binary zeros. • When the key-wrapping method is PKOAEP2 (value at offset 26 is X'03'), the value should be filled with binary zeros. The encoded message, which contains the key, is wrapped with an RSA public-key. The key token is external. **X'03'** The key-material state is the key is wrapped with the AES master-key. The field contains the MKVP of the AES master-key used to wrap the key. The 8-byte MKVP is left-aligned in the field and padded on the right low-order bytes with binary zeros. The key token is internal.
026	01	Encrypted section key-wrapping method (how data in the encrypted section is protected): **Value** **Meaning** **X'00'** No key-wrapping method (no key present or key is clear). The key token is external or internal. **X'02'** AESKW (ANS X9.102). The key token is external with a key wrapped by an AES key-encrypting key, or the key token is internal with a key wrapped by the AES master-key. **X'03'** PKOAEP2. Message *M*, which contains the key, is encoded using the RSAES-OAEP scheme of the RSA PKCS #1 v2.1 standard. The encoded message (*EM*) is produced using the given hash algorithm by encoding message *M* using the Bellare and Rogaway Optimal Asymmetric Encryption Padding (OAEP) method for encoding messages. For PKAOEP2, *M* is defined as follows: *M* = [32 bytes: *hAD*] ⌂⌂¥ [2 bytes: bit length of the clear key] ⌂⌂¥ [clear key] where *hAD* is the message digest of the associated data, and is calculated using the SHA-256 algorithm on the data starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). *EM* is wrapped with an RSA public-key. The key token is external. All unused values are reserved and undefined.

Table 241. AES DKYGENKY variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
027	01	Hash algorithm used for wrapping key or encoding message. Meaning depends on whether the encrypted section key-wrapping method (value at offset 26) is no key-wrapping method, AESKW, or PKOAEP2: No key-wrapping method (value at offset 26 is X'00') Hash algorithm used for wrapping key when encrypted section key-wrapping method is no key-wrapping method: **Value** **Meaning** **X'00'** No hash (no key present) All unused values are reserved and undefined. The key token is external or internal. AESKW key-wrapping method (value at offset 26 is X'02') Hash algorithm used for wrapping key when encrypted section key-wrapping method is AESKW. The value indicates the algorithm used to calculate the message digest of the associated data. The message digest is included in the wrapped payload and is calculated starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). **Value** **Meaning** **X'02'** SHA-256 All unused values are reserved and undefined. The key token is external or internal. PKOAEP2 key-wrapping method (value at offset 26 is X'03') Hash algorithm used for encoding message when encrypted section key-wrapping method is PKOAEP2. The value indicates the given hash algorithm used for encoding message *M* using the RSAES-OAEP scheme of the RSA PKCS #1 v2.1 standard. **Value** **Meaning** **X'01'** SHA-1 **X'02'** SHA-256 **X'04'** SHA-384 **X'08'** SHA-512 All unused values are reserved and undefined. The key token is external.

Table 241. AES DKYGENKY variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
028	01	Payload format version (identifies format of the payload): **Value** **Meaning** **X'01'** V1 payload (V1PYLD). The payload format depends on the encrypted section key-wrapping method (value at offset 26): **Value at offset 26** **Meaning** **X'00'** There is no key-wrapping method. When no key is present, there is no payload. The key token is external or internal. **X'02'** The key-wrapping method is AESKW and the payload is fixed length based on the maximum possible key size of the algorithm for the key. The key is padded with random data to the size of the largest key for that algorithm. This helps to deter attacks on keys known to be weaker. The key length cannot be inferred by the size of the payload. The key token is external or internal. **X'03'** The key-wrapping method is PKOAEP2 and the payload length is equal to the modulus size in bits of the RSA transport key used to wrap the encoded message. The key token is external. When the external key is exported, the internal target key will have the same V1 payload format. All unused values are reserved and undefined.
029	01	Reserved, binary zero.
End of wrapping information section		
AESKW or PKOAEP2 components: (1) associated data section and (2) optional wrapped AESKW payload or wrapped PKOAEP2 payload (no payload if no key present)		
Associated data section		
030	01	Associated data section version: **Value** **Meaning** **X'01'** Version 1 format of associated data
031	01	Reserved, binary zero.
032	02	Length in bytes of all the associated data for the key token: 26 - 353.
034	01	Length in bytes of the optional key label (*kl*): 0 or 64.
035	01	Length in bytes of the optional IBM extended associated data (*iead*): 0.
036	01	Length in bytes of the optional user-definable associated data (*uad*): 0 - 255.
037	01	Reserved, binary zero.

Table 241. AES DKYGENKY variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
038	02	Length in *bits* of the wrapped payload (*pl*): 0, 512 - 4096. • For no key-wrapping method (no key present), *pl* is 0. • For PKOAEP2 encoded payloads, *pl* is the length in bits of the modulus size of the RSA key used to wrap the payload. This can be 512 - 4096. • For an AESKW formatted payload, *pl* is based on the key size of the algorithm type and the payload format version: • AES algorithm (value at offset 41 is X'02') An AES key can have a length of 16, 24, or 32 bytes (128, 192, or 256 bits). The following table shows the payload length for a given AES key size and payload format: **AES key size** / **Bit length of V0 payload (value at offset 28 is X'00')** / **Bit length of V1 payload (value at offset 28 is X'01')** 16 bytes (128 bits) / Not applicable / 640 24 bytes (192 bits) / Not applicable / 640 32 bytes (256 bits) / Not applicable / 640
040	01	Reserved, binary zero.
041	01	Algorithm type (algorithm for which the key can be used): **Value** **Meaning** **X'02'** AES All unused values are reserved and undefined.
042	02	Key type (general class of the key): **Value** **Meaning** **X'0009'** DKYGENKY All unused values are reserved and undefined.
044	01	Key usage fields count (*kuf*): 2, 4 - 6. Key-usage field information defines restrictions on the use of the key. Count is based on type of key to diversify (value at offset 45): **Value at offset 45** / **Type of key to diversify** / **kuf count** **X'00'** / D-ALL / 2 **X'01'** / D-CIPHER / 4 **X'02'** / D-MAC / 4 (not DK enabled) 5 (DK enabled) **X'03'** / D-EXP / 6 **X'04'** / D-IMP / 6 **X'05'** / D-PPROT / 5 **X'06'** / D-PCALC / 5 **X'07'** / D-PPRW / 5 Each key-usage field is 2 bytes in length. The value in this field indicates how many 2-byte key usage fields follow.

Table 241. AES DKYGENKY variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
045	01	Key-usage field 1, high-order byte (type of key to diversify). Defines the type of diversified key that this diversifying key can generate. **Value Meaning** **X'00'** Any key type listed below (D-ALL) **X'01'** CIPHER (D-CIPHER) **X'02'** MAC (D-MAC) **X'03'** EXPORTER (D-EXP) **X'04'** IMPORTER (D-IMP) **X'05'** PINPROT (D-PPROT) **X'06'** PINCALC (D-PCALC) **X'07'** PINPRW (D-PPRW) All unused values are reserved and undefined.
046	01	Key-usage field 1, low-order byte (user-defined extension control).
047	01	Key-usage field 2, high-order byte (related generated key-usage field level of control): **Value Meaning** **B'1xxx xxxx'** The key usage fields of the key to be generated must be equal (KUF-MBE) to the related generated key usage fields that start with key usage field 3 below. **B'0xxx xxxx'** The key usage fields of the key to be generated must be permissible (KUF-MBP) based on the related generated key usage fields that start with key usage field 3 below. A key to be diversified is not permitted to have a higher level of usage than the related key usage fields permit. The key to be diversified is only permitted to have key usage that is less than or equal to the related key usage fields. One exception is the UDX-ONLY setting in the generated key usage fields. The UDX-ONLY setting must *always* be equal to the UDX-ONLY setting in the related key usage fields. Undefined when the value at offset 45 = X'00' (D-ALL). All unused bits are reserved and must be zero.
048	01	Key-usage field 2, low-order byte (key-derivation sequence level): **Value Meaning** **X'00'** Use this diversifying key to generate a Level 0 diversified key (DKYL0). The type of key to diversify (value at offset 45) determines the key type of the generated key. Level 0 is a completed key. **X'01'** Use this diversifying key to generate a Level 1 diversified key (DKYL1). **X'02'** Use this diversifying key to generate a Level 2 diversified key (DKYL2). All unused values are reserved and undefined.

Table 241. AES DKYGENKY variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
049, for *kuf* > 3	02	Optional key-usage field 3 (related generated key usage fields). Controls the key usage field 1 values of the diversified key. Meaning depends on type of key to diversify (value at offset 45): **Value at offset 45** **Meaning** **X'01'** Same as key-usage field 1 of AES CIPHER key. **X'02'** Same as key-usage field 1 of AES MAC key. **X'03'** Same as key-usage field 1 of AES EXPORTER key. **X'04'** Same as key-usage field 1 of AES IMPORTER key. **X'05'** Same as key-usage field 1 of AES PINPROT key . **X'06'** Same as key-usage field 1 of AES PINCALC key. **X'07'** Same as key-usage field 1 of AES PINPRW key. All unused bits are reserved and must be zero.
051, for *kuf* > 3	02	Optional key-usage field 4 (related generated key usage fields). Controls the key usage field 2 values of the diversified key. Meaning depends on type of key to diversify (value at offset 45): **Value at offset 45** **Meaning** **X'01'** Same as key-usage field 2 of AES CIPHER key. **X'02'** Same as key-usage field 2 of AES MAC key. **X'03'** Same as key-usage field 2 of AES EXPORTER key. **X'04'** Same as key-usage field 2 of AES IMPORTER key. **X'05'** Same as key-usage field 2 of AES PINPROT key. **X'06'** Same as key-usage field 2 of AES PINCALC key. **X'07'** Same as key-usage field 2 of AES PINPRW key. All unused bits are reserved and must be zero.
053, for *kuf* > 4	02	Optional key-usage field 5 (related generated key usage fields). Controls the key usage field 3 values of the diversified key. Meaning depends on type of key to diversify (value at offset 45): **Value at offset 45** **Meaning** **X'02'** Same as key-usage field 3 of AES MAC key. **X'03'** Same as key-usage field 3 of AES EXPORTER key. **X'04'** Same as key-usage field 3 of AES IMPORTER key. **X'05'** Same as key-usage field 3 of AES PINPROT key. **X'06'** Same as key-usage field 3 of AES PINCALC key. **X'07'** Same as key-usage field 3 of AES PINPRW key. All unused bits are reserved and must be zero.
055, for *kuf* > 5	02	Optional key-usage field 6 (related generated key usage fields). Controls the key usage field 4 values of the diversified key. Meaning depends on type of key to diversify (value at offset 45): **Value at offset 45** **Meaning** **X'03'** Same as key-usage field 4 of AES EXPORTER key. **X'04'** Same as key-usage field 4 of AES IMPORTER key. All unused bits are reserved and must be zero.

Table 241. AES DKYGENKY variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
045 + (2 * *kuf*)	01	Key management fields count (*kmf*): 3. Key-management field information describes how the data is to be managed or helps with management of the key material. Each key-management field is 2 bytes in length. The value in this field indicates how many 2-byte key management fields follow.
046 + (2 * *kuf*)	01	Key-management field 1, high-order byte (symmetric-key export control).
047 + (2 * *kuf*)	01	Key-management field 1, low-order byte (export control by algorithm).
048 + (2 * *kuf*)	01	Key-management field 2, high-order byte (key completeness).
049 + (2 * *kuf*)	01	Key-management field 2, low-order byte (security history).
050 + (2 * *kuf*)	01	Key-management field 3, high-order byte (pedigree original).
051 + (2 * *kuf*)	01	Key-management field 3, low-order byte (pedigree current).
052 + (2 * *kuf*)	*kl*	Optional key label.
052 + (2 * *kuf*) + *kl*	*iead*	Optional IBM extended associated data (not used).
052 + (2 * *kuf*) + *kl* + *iead*	*uad*	Optional user-defined associated data.
End of associated data section		
Optional wrapped AESKW formatted payload or wrapped PKOAEP2 encoded payload (no payload if no key present)		

Table 241. AES DKYGENKY variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description		
052 + (2 * *kuf*) + *kl* + *iead* + *uad*	(*pl* + 7) / 8	Contents of payload (*pl* is in *bits*) depending on the encrypted section key-wrapping method (value at offset 26):		
		Value at offset 26	**Encrypted section key-wrapping method**	**Meaning**
		X'02'	AESKW	An encrypted payload which the Segment 2 code creates by wrapping the unencrypted AESKW formatted payload. The payload is made up of the integrity check value, pad length, length of hash options and hash, hash options, hash of the associated data, key material, and padding. The key token is internal.
		X'03'	PKOAEP2	An encrypted PKOAEP2 encoded payload created using the RSAES-OAEP scheme of the PKCS #1 v2.1 standard. The message *M* is encoded for a given hash algorithm using the Bellare and Rogaway Optimal Asymmetric Encryption Padding (OAEP) method for encoding messages. For PKAOEP2, *M* is defined as follows: *M* = [32 bytes: *hAD*] ⌂¥ [2 bytes: bit length of the clear key] ⌂¥ [clear key] where *hAD* is the message digest of the associated data, and is calculated using the SHA-256 algorithm starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). The encoded message is wrapped with an RSA public-key according to the standard. The key token is external.
End of optional wrapped AESKW formatted payload or wrapped PKOAEP2 encoded payload				
End of AESKW or PKOAEP2 components				
Note: All numbers are in big endian format.				

AES SECMSG variable-length symmetric key token

View a table showing the format of the SECMSG variable-length symmetric key-token.

Table 242. AES SECMSG variable-length symmetric key-token, version X'05'.

Offset (bytes)	Length (bytes)	Description
Header		

Table 242. AES SECMSG variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
000	01	Token identifier (note that it is intentional that an external key-token is not defined): **Value** **Meaning** **X'01'** Internal key-token (encrypted key is wrapped with the master key, or there is no payload). All unused values are reserved and undefined.
001	01	Reserved, binary zero.
002	02	Length in bytes of the overall token structure: **Key token** **Minimum token length** **Skeleton** 46 + (2 * 2) + (2 * 3) + 0 + 0 + 0 + 0 = 56 **Encrypted V1 payload** 46 + (2 * 2) + (2 * 3) + 0 + 0 + 0 + ((640 + 7) / 8) = 136 **Key token** **Maximum token length** **Internal** 46 + (2 * 2) + (2 * 3) + 64 + 0 + 255 + ((640 + 7) / 8) = 455
004	01	Token version number (identifies the format of this key token): **Value** **Meaning** **X'05'** Version 5 format of the key token (variable-length symmetric key-token)
005	03	Reserved, binary zero.
End of header		
Wrapping information section (all data related to wrapping the key)		
008	01	Key material state: **Value** **Meaning** **X'00'** No key is present. This is called a skeleton key-token. The key token is internal. **X'03'** Key is wrapped with the AES master-key. The encrypted section key-wrapping method is AESKW. The key token is internal. All unused values are reserved and undefined.
009	01	Key verification pattern (KVP) type: **Value** **Meaning** **X'00'** No KVP (no key present or key is wrapped with an RSA public-key). The key token is external or internal. **X'01'** AESMK (8 leftmost bytes of SHA-256 hash: X'01 || clear AES MK). The key token is internal. All unused values are reserved and undefined.

Table 242. AES SECMSG variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
010	16	KVP (value depends on value of key material state, that is, the value at offset 8): **Value at offset 8** **Value of KVP** **X'00'** The key-material state is no key present. The field should be filled with binary zeros. The key token is internal. **X'03'** The key-material state is the key is wrapped with the AES master-key. The field contains the MKVP of the AES master-key used to wrap the key. The 8-byte MKVP is left-aligned in the field and padded on the right low-order bytes with binary zeros. The key token is internal. All unused values are reserved and undefined.
026	01	Encrypted section key-wrapping method (how data in the encrypted section is protected): **Value** **Meaning** **X'00'** No key-wrapping method (no key present). The key token is external or internal. **X'02'** AESKW (ANS X9.102). The key token is internal with a key wrapped by the AES master-key. All unused values are reserved and undefined.
027	01	Hash algorithm used for wrapping key or encoding message. Meaning depends on whether the encrypted section key-wrapping method (value at offset 26) is no key-wrapping method or AESKW: No key-wrapping method (value at offset 26 is X'00') Hash algorithm used for wrapping key when encrypted section key-wrapping method is no key-wrapping method: **Value** **Meaning** **X'00'** No hash (no key present). All unused values are reserved and undefined. The key token is internal. AESKW key-wrapping method (value at offset 26 is X'02') Hash algorithm used for wrapping key when encrypted section key-wrapping method is AESKW. The value indicates the algorithm used to calculate the hash digest of the associated data. The hash digest is included in the wrapped payload and is calculated starting at offset 30 for the length in bytes of all the associated data for the key token (length value at offset 32). **Value** **Meaning** **X'02'** SHA-256 All unused values are reserved and undefined. The key token is internal.

Table 242. AES SECMSG variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
028	01	Payload format version (identifies format of the payload). **Value** **Meaning** **X'01'** V1 payload. The payload format depends on the encrypted section key-wrapping method (value at offset 26): **Value** **Meaning** **X'00'** There is no key-wrapping method. When no key is present, there is no payload. The key token is internal. **X'02'** The key-wrapping method is AESKW and the payload is fixed length based on the maximum possible key size of the algorithm for the key. The key is padded with random data to the size of the largest key for that algorithm. This helps to deter attacks on keys known to be weaker. The key length cannot be inferred by the size of the payload. The key token is internal. All unused values are reserved and undefined.
029	01	Reserved, binary zero.
End of wrapping information section		
AESKW components: (1) associated data section and (2) optional wrapped AESKW formatted payload (no payload if no key present)		
Associated data section		
030	01	Associated data section version: **Value** **Meaning** **X'01'** Version 1 format of associated data.
031	01	Reserved, binary zero.
032	02	Length in bytes of all the associated data for the key token: 26 - 345.
034	01	Length in bytes of the optional key label (*kl*) : 0 or 64 .
035	01	Length in bytes of the optional IBM extended associated data (*iead*) : 0 .
036	01	Length in bytes of the optional user-definable associated data: 0 - 255 (*uad*).
037	01	Reserved, binary zero.

Table 242. AES SECMSG variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
038	02	Length in *bits* of the wrapped payload (*pl*): 0, 640. For no key-wrapping method (no key present), *pl*is 0. For an AESKW formatted payload, *pl*is based on the key size of the algorithm type and the payload format version: • AES algorithm (value at offset 41 is X'02') • An AES key can have a length of 16, 24, or 32 bytes (128, 192, or 256 bits). The payload length for a given AES key size and payload format are as follows: **AES key size** / **Bit length of V0 payload (value at offset 28 is X'00')** / **Bit length of V1 payload (value at offset 28 is X'01')** 16 bytes (128 bits) / Not applicable / 640 24 bytes (192 bits) / Not applicable / 640 32 bytes (256 bits) / Not applicable / 640
040	01	Reserved, binary zero.
041	01	Algorithm type (algorithm for which the key can be used): **Value** **Meaning** **X'02'** AES All unused values are reserved and undefined.
042	02	Key type (general class of the key): **Value** **Meaning** **X'000A'** SECMSG All unused values are reserved and undefined.
044	01	Key usage fields count (*kuf*): 2. Key-usage field information defines restrictions on the use of the key. Each key-usage field is 2 bytes in length. The value in this field indicates how many 2-byte key usage fields follow.
045	01	Key-usage field 1, high-order byte (secure message encryption enablement): **Value** **Meaning** **X'00'** Enable the encryption of PINs in an EMV secure message (SMPIN). All unused values are reserved and undefined.
046	01	Key-usage field 1, low-order byte (user-defined extension control).
047	01	Key-usage field 2, high-order byte (verb restriction): **Value** **Meaning** **X'00'** Any verb can use this key (ANY-USE). **X'01'** Only CSNBDPC can use this key (DPC-ONLY). All unused values are reserved and undefined.

Table 242. AES SECMSG variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
048	01	Key-usage field 2, low-order byte (reserved). All bits are reserved and must be zero.
049	01	Key management fields count (*kmf*): 3. Key-management field information describes how the data is to be managed or helps with management of the key material. Each key-management field is 2 bytes in length. The value in this field indicates how many 2-byte key management fields follow.
050	01	Key-management field 1, high-order byte (symmetric-key export control). Note that an AES SECMSG key cannot be exported because it has no external key-token defined. Symmetric-key export control: **Value Meaning** **B'1xxx xxxx'** Undefined. **B'0xxx xxxx'** Prohibit export using symmetric key (NOEX-SYM). Unauthenticated asymmetric-key export control: **Value Meaning** **B'x1xx xxxx'** Undefined. **B'x0xx xxxx'** Prohibit export using unauthenticated asymmetric key (NOEXUASY). Authenticated asymmetric-key export control: **Value Meaning** **B'xx1x xxxx'** Undefined. **B'xx0x xxxx'** Prohibit export using authenticated asymmetric key (NOEXAASY). Raw-key export control: **Value Meaning** **B'xxx1 xxxx'** Undefined. **B'xxx0 xxxx'** Prohibit export using raw key (NOEX-RAW). Defined for future use. Currently ignored. All unused bits are reserved and must be zero.

Table 242. AES SECMSG variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description
051	01	Key-management field 1, low-order byte (export control by algorithm). Note that an AES SECMSG key can only be internal; it cannot be external. DES-key export control: **Value Meaning** **B'1xxx xxxx'** Prohibit export using DES key (NOEX-DES). **B'0xxx xxxx'** Undefined. AES-key export control: **Value Meaning** **B'x1xx xxxx'** Prohibit export using AES key (NOEX-AES). **B'x0xx xxxx'** Undefined. RSA-key export control: **Value Meaning** **B'xxxx 1xxx'** Prohibit export using RSA key (NOEX-RSA). **B'xxxx 0xxx'** Undefined. All unused bits are reserved and must be zero.
052	01	Key-management field 2, high-order byte (key completeness).
053	01	Key-management field 2, low-order byte (security history).
054	01	Key-management field 3, high-order byte (pedigree original).
055	01	Key-management field 3, low-order byte (pedigree current).
056	01	Optional key label.
056 + *kl*	*iead*	Optional IBM extended associated data (unused).
056 + *kl* + *iead*	*uad*	Optional user-defined associated data.
End of associated data section		
Optional wrapped AESKW formatted payload, or wrapped PKOAEP2 encoded payload (no payload if no key present)		

Table 242. AES SECMSG variable-length symmetric key-token, version X'05' (continued).

Offset (bytes)	Length (bytes)	Description		
056 + *kl* + *iead* + *uad*	(*pl* + 7) / 8	Contents of payload (*pl* is in *bits*) depending on the encrypted section key-wrapping method (value at offset 26):		
		Value at offset 26	**Encrypted section key-wrapping method**	**Meaning**
		X'02'	AESKW	An encrypted payload which the Segment 2 code creates by wrapping the unencrypted AESKW formatted payload. The payload is made up of the integrity check value, pad length, length of hash options and hash, hash options, hash of the associated data, key material, and padding. The key token is internal.
End of optional wrapped AESKW formatted payload				
End of AESKW components				
Note: All numbers are in big endian format.				

TR-31 optional block data

A TR-31 key block can contain an optional block with IBM-defined data.

See *X9 TR-31 2010: Interoperable Secure Key Exchange Block Specification for Symmetric Algorithms* for the definition of a TR-31 key block. As defined by X9 TR-31, a TR-31 key block can contain one or more optional blocks. A TR-31 key block contains at least one optional block when byte number 12 - 13 is a value other than ASCII string "00".

The data of an IBM-defined optional block contains ASCII string *10* (X'3130') in the first two bytes, and contains ASCII string *IBMC* (X'49424D43') beginning at offset 4 of the data. CCA treats an optional block with these characteristics as a proprietary container for a CCA control vector. See Table 243 on page 877. An optional block with different characteristics is ignored by CCA.

If a TR-31 key block contains an optional block as defined by Table 243 on page 877, the data contains a copy of the 8-byte or 16-byte DES control vector that was in the CCA key-token of the key being exported. The copied control vector is in hex-ASCII format.

The control vector is only copied from the CCA key-token when the user of the Key Export to TR31 verb specifies a control vector transport control keyword (**INCL-CV** or **ATTR-CV**):

1. If the optional block contains a control vector as the result of specifying the **INCL-CV** keyword during export, the key usage and mode of use fields indicate the key attributes, and these attributes are verified during export to be compatible with the ones in the included control vector.

2. If the optional block contains a control vector as the result of specifying the **ATTR-CV** keyword during export, the key usage field (byte number 5 - 6 of the TR-31 key block) is set to the proprietary value *10* (X'3130'), and the mode of use field (byte number 8) is set to the proprietary value *1* (X'31'). These proprietary values indicate that the key attributes are specified in the included control vector.

See Chapter 4, "TR-31 symmetric-key management," on page 71 for additional information on how CCA uses an IBM-defined optional block in a TR-31 key block.

Table 243. IBM optional block data in a TR-31 key block

Offset (bytes)	Length (bytes)	Description
00	02	Proprietary ID of TR-31 optional block (alphanumeric-ASCII): **X'3130'** IBM proprietary optional block (ASCII string *10*)
02	02	Length of optional block (hex-ASCII): For TLV valued to *01*: **X'3143'** "1C" for 8-byte (single-length) control vector **X'3243'** "2C" for 16-byte (double-length) control vector
Beginning of optional block data		
04	04	Magic value (alphanumeric-ASCII): **X'49424D43'** A constant value ("IBMC") used to reduce ambiguity and the chance for false interpretation of the proprietary optional blocks of non-IBM vendors. An optional block that uses the same proprietary ID but does not include this magic value will be ignored.
08	02	Tag-length-value (TLV) ID (numeric-ASCII): **X'3031'** IBM CCA control vector (*01*)
10	02	Length of TLV (hex-ASCII) For TLV valued to *01*: **X'3134'** "14" (decimal 20) TLV of 8-byte control vector **X'3234'** *24* (decimal 36) TLV of 16-byte control vector
12	16 or 32	Control vector (hex-ASCII)

Trusted blocks

A *key token* is a data structure that contains information about a key and usually contains a key or keys.

A trusted block is an extension of CCA key tokens using new section identifiers. A trusted block was introduced to CCA beginning with Release 3.25. Trusted blocks are an integral part of a remote key-loading process. See "Remote key loading" on page 46.

In general, a key that is available to an application program or held in key storage is multiply-enciphered by some other key. When a key is enciphered by the CCA node's master key, the key is designated an internal key and is held in an internal key-token structure. Therefore, an *internal key token* or *internal trusted block* is used to hold a key and its related information for use at a specific CCA node.

An *external key token* or *external trusted block* is used to communicate a key between nodes, or to hold a key in a form not enciphered by a CCA master key. DES keys and PKA private-keys contained in an external key-token or external trusted block are multiply-enciphered by a *transport key*. In a CCA-node, a transport key is a double-length DES key encrypting key (KEK).

Trusted blocks contain various items, some of which are optional, and some of which can be present in different forms. Tokens are composed of concatenated sections that, unlike CCA PKA key tokens, occur in no prescribed order.

As with other CCA key-tokens, both internal and external forms are defined:

- An external trusted block contains a randomly generated confounder and a triple-length MAC key enciphered under a DES IMP-PKA transport key. The MAC key is used to calculate an ISO 16609 CBC mode TDES MAC of the trusted block contents. An external trusted block is created by the Trusted Block Create verb. This verb can:
  1. Create an inactive external trusted block
  2. Change an external trusted block from inactive to active
- An internal trusted block contains a confounder and triple-length MAC key enciphered under a variant of the PKA master key. The MAC key is used to calculate a TDES MAC of the trusted block contents. A PKA master-key verification pattern is also included to enable determination that the proper master key is available to process the key. The Remote Key Export verb only operates on trusted blocks that are internal. An internal trusted block must be imported from an external trusted block that is active using the PKA Key Import verb.

Note: Trusted blocks do not contain a private key section.

Trusted block organization

A trusted block is a concatenation of a header followed by an unordered set of sections.

Some elements are required, while others are optional. The data structures of these sections are summarized in Table 244.

Table 244. Trusted block sections and their use

Section	Reference	Usage
Header	Table 245 on page 880	Trusted block token header
X'11'	Table 246 on page 881	Trusted block public key
X'12'	Table 247 on page 882	Trusted block rule
X'13'	Table 254 on page 889	Trusted block name (key label)
X'14'	Table 255 on page 889	Trusted block information
X'15'	Table 259 on page 892	Trusted block application-defined data

Every trusted block starts with a token header. The first byte of the token header determines the key form:

- An external header (first byte X'1E'), created by the Trusted Block Create verb
- An internal header (first byte X'1F'), imported from an active external trusted block by the PKA Key Import verb

Following the token header of a trusted block is an unordered set of sections. A trusted block is formed by concatenating these sections to a trusted block header:

- An optional public-key section (trusted block section identifier X'11')

  The trusted block trusted RSA public key section includes the key itself in addition to a key-usage flag. No multiple sections are allowed.
- An optional rule section (trusted block section identifier X'12')

  A trusted block can have zero or more rule sections.

  1. A trusted block with no rule sections can be used by the PKA Key Token Change and PKA Key Import verbs. A trusted block with no rule sections can also be used by the Digital Signature Verify verb, provided there is an RSA public key section that has its key-usage flag bits set to allow digital signature operations.
  2. At least one rule section is required when the Remote Key Export verb is used to:
     - Generate an RKX key-token
     - Export an RKX key-token
     - Export a CCA DES key-token
     - Encrypt the clear generated or exported key using the provided vendor certificate
  3. If a trusted block has multiple rule sections, each rule section must have a unique 8-character Rule ID.
- An optional name (key label) section (trusted block section identifier X'13')

  The trusted block name section provides a 64-byte variable to identify the trusted block, just as key labels are used to identify other CCA keys. This name, or label, enables a host access-control system such as RACF® to use the name to verify that the application has authority to use the trusted block. No multiple sections are allowed.
- A required information section (trusted block section identifier X'14')

  The trusted block information section contains control and security information related to the trusted block. The information section is required while the others are optional. This section contains the cryptographic information that guarantees its integrity and binds it to the local system. No multiple sections are allowed.
- An optional application-defined data section (trusted block section identifier X'15')

  The trusted block application-defined data section can be used to include application-defined data in the trusted block. The purpose of the data in this section is defined by the application. CCA does not examine or use this data in any way. No multiple sections are allowed.

Trusted block integrity

An enciphered confounder and triple-length MAC key contained within the required information section of the trusted block is used to protect the integrity of the trusted block.

The randomly generated MAC key is used to calculate an ISO 16609 CBC mode TDES MAC of the trusted block contents. Together, the MAC key and MAC value provide a way to verify that the trusted block originated from an authorized source, and binds it to the local system.

An external trusted block has its MAC key enciphered under an IMP-PKA key-encrypting key. An internal trusted block has its MAC key enciphered under a variant of the PKA master key, and the master-key verification pattern is stored in the information section.

Number representation in trusted blocks

The number format in trusted blocks.

- All length fields are in binary
- All binary fields (exponents, lengths, and so forth) are stored with the high-order byte first (left, low-address, z/OS format); thus the least significant bits are to the right and preceded with zero-bits to the width of a field
- In variable-length binary fields that have an associated field-length value, leading bytes that would otherwise contain X'00' can be dropped and the field shortened to contain only the significant bits

Trusted block sections

At the beginning of every trusted block is a trusted block header.

The header contains the following information:

- A token identifier, which specifies if the token contains an external or internal key-token
- A token version number to allow for future changes
- A length in bytes of the trusted block, including the length of the header

The trusted block header is defined in Table 245.

Table 245. Trusted block header format

Offset (bytes)	Length (bytes)	Description
000	001	Token identifier (a flag that indicates token type) **Value** **Description** **X'1E'** External trusted block token **X'1F'** Internal trusted block token
001	001	Token version number (X'00').
002	002	Length of the key-token structure in bytes.
004	004	Reserved, binary zero.

Note: See “Number representation in trusted blocks.”

Following the header, in no particular order, are trusted block sections. There are five different sections defined, each identified by a one-byte section identifier (X'11' - X'15'). Two of the five sections have subsections defined. A subsection is a tag-length-value (TLV) object, identified by a two-byte subsection tag.

Only sections X'12' and X'14' have subsections defined; the other sections do not. A section and its subsections, if any, are one contiguous unit of data. The subsections are concatenated to the related section, but are otherwise in no particular order.

Section X'12' has five subsections defined (X'0001' - X'0005'). Section X'14' has two subsections, (X'0001' and X'0002'). Of all the subsections, only subsection X'0001' of section X'14' is required. Section X'14' is also required.

The trusted block sections and subsections are described in detail in the following topics.

Trusted block section X'11'

Trusted block section X'11' contains the trusted RSA public key in addition to a key-usage flag indicating whether the public key is usable in key-management operations, digital signature operations, or both.

Section X'11' is optional. No multiple sections are allowed. It has no subsections defined.

Table 246. Trusted block trusted RSA public key section (X'11')

Offset (bytes)	Length (bytes)	Description
000	001	Section identifier: **X'11'** Trusted block trusted RSA public key
001	001	Section version number (X'00').
002	002	Section length (16 + *xxx* + *yyy*).
004	002	Reserved, must be binary zero.
006	002	RSA public key exponent field length in bytes, *xxx*.
008	002	RSA public key modulus length in bits.
010	002	RSA publickey modulus field length in bytes, *yyy*.
012	xxx	Public key exponent, *e* (this field length is typically 1, 3, or 64 - 512 bytes). *e* must be odd and $1 \le e < n$. (*e* is frequently valued to 3 or $2^{16}+1$ (=65537), otherwise *e* is of the same order of magnitude as the modulus). **Note:** Although the current product implementation does not generate such a public key, you can import an RSA public key having an exponent valued to two (2). Such a public key (a Rabin key) can correctly validate an ISO 9796-1 digital signature.
012 + *xxx*	*yyy*	RSA public key modulus, *n*. $n=pq$, where *p* and *q* are prime and $2^{512} \le n < 2^{4096}$. The field length is 64 - 512 bytes. **Note:** Keys with a modulus greater than 2048 bits are not supported in releases before Release 3.30.
012 + *xxx* + *yyy*	004	Flags: **Value Description** **X'00000000'** Trusted block public key can be used in digital signature operations only **X'80000000'** Trusted block public key can be used in both digital signature and key management operations **X'C0000000'** Trusted block public key can be used in key management operations only

Trusted block section X'12'

Trusted block section X'12' contains information that defines a rule.

TA trusted block can have zero or more rule sections.

1. A trusted block with no rule sections can be used by the PKA Key Token Change and PKA Key Import verbs. A trusted block with no rule sections can be used by the Digital Signature Verify verb, provided there is an RSA public key section that has its key-usage flag set to allow digital signature operations.
2. At least one rule section is required when the Remote Key Export verb is used to:

- Generate an RKX key-token
- Export an RKX key-token
- Export a CCA DES key-token
- Generate or export a key encrypted by a public key. The public key is contained in a vendor certificate and is the root certification key for the ATM vendor. It is used to verify the digital signature on public-key certificates for specific individual ATMs.

3. If a trusted block has multiple rule sections, each rule section must have a unique 8-character Rule ID.

Section X'12' is the only section that can have multiple sections. Section X'12' is optional.

Note: The overall length of the trusted block cannot exceed its maximum size of 3500 bytes.

Five subsections (TLV objects) are defined.

Table 247. Trusted block rule section (X'12')

Offset (bytes)	Length (bytes)	Description
000	001	Section identifier: **X'12'** Trusted block rule
001	001	Section version number (X'00').
002	002	Section length in bytes (20 + *yyy*).
004	008	Rule ID (in ASCII). An 8-byte character string that uniquely identifies the rule within the trusted block. Valid ASCII characters are: A - Z, a - z, 0 - 9, - (hyphen), and _ (underscore), left-aligned and padded on the right with space characters.
012	004	Flags (undefined flag bits are reserved and must be zero). **Value Description** **X'00000000'** Generate new key **X'00000001'** Export existing key
016	001	Generated key length. Length in bytes of key to be generated when flags value (offset 012) is set to generate a new key; otherwise ignore this value. Valid values are 8, 16, or 24; return an error if not valid.
017	001	Key-check algorithm identifier (all others are reserved and must not be used): **Value Description** **X'00'** Do not compute key-check value. Set the key_check_value_length variable to zero. **X'01'** Encrypt an 8-byte block of binary zeros with the key. See “Encrypt zeros DES-key verification algorithm” on page 930. **X'02'** Compute the MDC-2 hash of the key. See “Modification Detection Code calculation” on page 930.

Table 247. Trusted block rule section (X'12') (continued)

Offset (bytes)	Length (bytes)	Description
018	001	Symmetric encrypted output key format flag (all other values are reserved and must not be used). Return the indicated symmetric key-token using the *sym_encrypted_key_identifier* parameter. **Value** **Description** **X'00'** Return an RKX key-token encrypted under a variant of the MAC key. **Note:** This key format is permitted when the flags value (offset 012) is set to either: 1. Generate a new key 2. Export an existing key **X'01'** Return a CCA DES key-token encrypted under a transport key. **Note:** This key format is not permitted if the flags value (offset 012) is set to generate a new key; it is only permitted when exporting an existing key.
019	001	Asymmetric encrypted output key format flag (all other values are reserved and must not be used). Return the indicated asymmetric key-token in the *asym_encrypted_key* variable. **Value** **Description** **X'00'** Do not return an asymmetric key. Set the *asym_encrypted_key_length* variable to zero. **X'01'** Output in PKCS-1.2 format. **X'02'** Output in RSA-OAEP format.
020	*yyy*	Rule section subsections (tag-length-value objects). A series of zero - five objects in TLV format.

Trusted block section X'12' subsections:

Section X'12' has five rule subsections (tag-length-value objects) defined.

These subsections are summarized in Table 248.

Table 248. Summary of trusted block X'12' subsections

Rule subsection tag	TLV object	Optional or required	Comments
X'0001'	Transport key variant	Optional	Contains variant to be XORed into the cleartext transport key.
X'0002'	Transport key rule reference	Optional; required to use an RKX key-token as a transport key	Contains the rule ID for the rule that must have been used to create the transport key.
X'0003'	Common export key parameters	Optional for key generation; required for key export of an existing key	Contains the export key and source key minimum and maximum lengths, an output key variant length and variant, a CV length, and a CV to be XORed with the cleartext transport key to control usage of the key.
X'0004'	Source key reference	Optional; required if the source key is an RKX key-token	Contains the rule ID for the rule used to create the source key. **Note:** Include all rules that will ever be needed when a trusted block is created. A rule cannot be added to a trusted block after it has been created.

Table 248. Summary of trusted block X'12' subsections (continued)

Rule subsection tag	TLV object	Optional or required	Comments
X'0005'	Export key CCA token parameters	Optional; used for export of CCA DES key tokens only	Contains mask length, mask, and CV template to limit the usage of the exported key. Also contains the template length and template that defines which source key labels are allowed. The key type of a source key input parameter can be "filtered" by using the export key CV limit mask (offset 005) and limit template (offset 005 + *yyy*) in this subsection.

Note: See "Number representation in trusted blocks" on page 880.

Trusted block section X'12' subsection X'0001'

Subsection X'0001' of the trusted block rule section (X'12') is the transport key variant TLV object. This subsection is optional. It contains a variant to be XORed into the cleartext transport key.

This subsection is defined in Table 249.

Table 249. Transport key variant subsection (X'0001') of trusted block rule section (X'12')

Offset (bytes)	Length (bytes)	Description
000	002	Subsection tag: **X'0001'** Transport key variant TLV object
002	002	Subsection length in bytes (8 + *nnn*).
004	001	Subsection version number (X'00').
005	002	Reserved, must be binary zero.
007	001	Length of variant field in bytes (*nnn*). This length must be greater than or equal to the length of the transport key that is identified by the *transport_key_identifier* parameter. If the variant is longer than the key, truncate it on the right to the length of the key prior to use.
008	*nnn*	Transport key variant. XOR this variant into the cleartext transport key, provided: (1) the length of the variant field value (offset 007) is not zero, and (2) the symmetric encrypted output key format flag (offset 018 in section X'12') is X'01'. **Note:** A transport key is not used when the symmetric encrypted output key is in RKX key-token format.

Note: See "Number representation in trusted blocks" on page 880.

Trusted block section X'12' subsection X'0002'

Subsection X'0002' of the trusted block rule section (X'12') is the transport key rule reference TLV object. This subsection is optional. It contains the rule ID for the rule that must have been used to create the transport key. This subsection must be present to use an RKX key-token as a transport key.

This subsection is defined in Table 250 on page 885.

Table 250. Transport key rule reference subsection (X'0002') of trusted block rule section (X'12')

Offset (bytes)	Length (bytes)	Description
000	002	Subsection tag: **X'0002'** Transport key rule reference TLV object
002	002	Subsection length in bytes (14).
004	001	Subsection version number (X'00').
005	001	Reserved, must be binary zero.
006	008	Rule ID. Contains the rule identifier for the rule that must have been used to create the RKX key-token used as the transport key. The Rule ID is an 8-byte string of ASCII characters, left-aligned and padded on the right with space characters. Acceptable characters are A - Z, a - z, 0 - 9, - (X'2D'), and _ (X'5F'). All other characters are reserved for future use.

Note: See "Number representation in trusted blocks" on page 880.

Trusted block section X'12' subsection X'0003'

Subsection X'0003' of the trusted block rule section X'12') is the common export key parameters TLV object. This subsection is optional, but is required for the key export of an existing source key (identified by the *source_key_identifier* parameter) in either RKX key-token format or CCA DES key-token format. For new key generation, this subsection applies the output key variant to the clear text generated key, if such an option is desired. It contains the input source key and output export key minimum and maximum lengths, an output key variant length and variant, a CV length, and a CV to be XORed with the clear text transport key.

This subsection is defined in Table 251.

Table 251. Common export key parameters subsection (X'0003') of trusted block rule section (X'12')

Offset (bytes)	Length (bytes)	Description
000	002	Subsection tag: **X'0003'** Common export key parameters TLV object
002	002	Subsection length in bytes (12 + *xxx* + *yyy*).
004	001	Subsection version number (X'00').
005	002	Reserved, must be binary zero.
007	001	Flags (must be set to binary zero).
008	001	Export key minimum length in bytes. Length must be 0, 8, 16, or 24. Also applies to the source key. Not applicable for key generation.
009	001	Export key maximum length in bytes (*yyy*). Length must be 0, 8, 16, or 24. Also applies to the source key. Not applicable for key generation.
010	001	Output key variant length in bytes (*xxx*). Valid values are 0 or 8 - 255. If greater than 0, the length must be at least as long as the longest key ever to be exported using this rule. If the variant is longer than the key, truncate it on the right to the length of the key prior to use. **Note:** The output key variant (offset 011) is not used if this length is zero.

Table 251. Common export key parameters subsection (X'0003') of trusted block rule section (X'12') (continued)

Offset (bytes)	Length (bytes)	Description
011	*xxx*	Output key variant. The variant can be any value. XOR this variant into the cleartext value of the output key.
011 + *xxx*	001	CV length in bytes (*yyy*). • If the length is not 0, 8, or 16, return an error. • If the length is 0, and if the source key is a CCA DES key-token, preserve the CV in the symmetric encrypted output if the output is to be in the form of a CCA DES key-token. • If a nonzero length is less than the length of the key identified by the *source_key_identifier* parameter, return an error. • If the length is 16, and if the CV (offset 012 + *xxx*) is valued to 16 bytes of X'00' (ignoring the key-part bit), then: 1. Ignore all CV bit definitions 2. If CCA DES key-token format, set the flag byte of the symmetric encrypted output key to indicate a CV value is present. 3. If the source key is eight bytes in length, do not replicate the key to 16 bytes
012 + *xxx*	*yyy*	CV. (See "Control vector table" on page 897.) Place this CV into the output exported key-token, provided that the symmetric encrypted output key format selected (offset 018 in rule section) is CCA DES key-token. • If the symmetric encrypted output key format flag (offset 018 in section X'12') indicates return an RKX key-token (X'00'), then ignore this CV. Otherwise, XOR this CV into the cleartext transport key. • XOR the CV of the source key into the cleartext transport key if the CV length (offset 011 + *xxx*) is set to 0. If a transport key to encrypt a source key has equal left and right key halves, return an error. Replicate the key halves of the key identified by the *source_key_identifier* parameter whenever all of these conditions are met: 1. The Key Generate - SINGLE-R command (offset X'00DB') is enabled in the active role 2. The CV length (offset 011 + *xxx*) is 16, and both CV halves are nonzero 3. The *source_key_identifier* parameter (contained in either a CCA DES key-token or RKX key-token) identifies an 8-byte key 4. The key-form bits (40 - 42) of this CV do not indicate a single-length key (are not set to zero) 5. Key-form bit 40 of this CV does not indicate the key is to have guaranteed unique halves (is not set to B'1'). See "Key Form Bits, *fff*" on page 903. **Note:** A transport key is not used when the symmetric encrypted output key is in RKX key-token format.

Note: See "Number representation in trusted blocks" on page 880.

Trusted block section X'12' subsection X'0004'

Subsection X'0004' of the trusted block rule section (X'12') is the source key rule reference TLV object. This subsection is optional, but is required if using an RKX key-token as a source key (identified by *source_key_identifier* parameter). It contains the rule ID for the rule used to create the export key. If this subsection is not present, an RKX key-token format source key will not be accepted for use.

This subsection is defined in Table 252 on page 887.

Table 252. Source key rule reference subsection (X'0004') of trusted block rule section (X'12')

Offset (bytes)	Length (bytes)	Description
000	002	Subsection tag: **X'0004'** Source key rule reference TLV object
002	002	Subsection length in bytes (14).
004	001	Subsection version number (X'00').
005	001	Reserved, must be binary zero.
006	008	Rule ID. Rule identifier for the rule that must have been used to create the source key. The Rule ID is an 8-byte string of ASCII characters, left-aligned and padded on the right with space characters. Acceptable characters are A - Z, a - z, 0 - 9, - (X'2D'), and _ (X'5F'). All other characters are reserved for future use.

Note: See "Number representation in trusted blocks" on page 880.

Trusted block section X'12' subsection X'0005'

Subsection X'0005' of the trusted block rule section (X'12') is the export key CCA token parameters TLV object. This subsection is optional. It contains a mask length, mask, and template for the export key CV limit. It also contains the template length and template for the source key label. When using a CCA DES key-token as a source key input parameter, its key type can be "filtered" by using the export key CV limit mask (offset 005) and limit template (offset 005+*yyy*) in this subsection.

This subsection is defined in Table 253.

Table 253. Export key CCA token parameters subsection (X'0005') of trusted block rule section (X'12')

Offset (bytes)	Length (bytes)	Description
000	002	Subsection tag: **X'0005'** Export key CCA token parameters TLV object
002	002	Subsection length in bytes (8 + *yyy* + *yyy* + *zzz*).
004	001	Subsection version number (X'00').
005	002	Reserved, must be binary zero.
007	001	Flags (must be set to binary zero).
008	001	Export key CV limit mask length in bytes (*yyy*). Do not use CV limits if this CV limit mask length (*yyy*) is zero. Use CV limits if *yyy* is nonzero, in which case *yyy*: • Must be 8 or 16 • Must not be less than the export key minimum length (offset 008 in subsection X'0003') • Must be equal in length to the actual source key length of the key **Example**: An export key minimum length of 16 and an export key CV limit mask length of 8 returns an error.

Table 253. Export key CCA token parameters subsection (X'0005') of trusted block rule section (X'12') (continued)

Offset (bytes)	Length (bytes)	Description
009	*yyy*	Export key CV limit mask (does not exist if *yyy*=0). See "Control-vector-base bit maps" on page 899 Indicates which CV bits to check against the source key CV limit template (offset 009 + *yyy*). **Examples**: A mask of X'FF' means check all bits in a byte. A mask of X'FE' ignores the parity bit in a byte.
009 + *yyy*	*yyy*	Export key CV limit template (does not exist if *yyy* = 0). Specifies the required values for those CV bits that are checked based on the export key CV limit mask (offset 009). (See "Control-vector-base bit maps" on page 899.) The export key CV limit mask and template have the same length, *yyy*. This is because these two variables work together to restrict the acceptable CVs for CCA DES key tokens to be exported. The checks work as follows: 1. If the length of the key to be exported is less than *yyy*, return an error 2. Logical AND the CV for the key to be exported with the export key CV limit mask 3. Compare the result to the export key CV limit template 4. Return an error if the comparison is not equal **Examples**: An export key CV limit mask of X'FF' for CV byte 1 (key type) along with an export key CV limit template of X'3F' (key type CVARENC) for byte 1 filters out all key types except CVARENC keys. **Note:** Using the mask and template to permit multiple key types is possible, but cannot consistently be achieved with one rule section. For example, setting bit 10 to B'1' in the mask and the template permits PIN processing keys and cryptographic variable encrypting keys, and only those keys. However, a mask to permit PIN-processing keys and key-encrypting keys, and only those keys, is not possible. In this case, multiple rule sections are required, one to permit PIN-processing keys and the other to permit key-encrypting keys.
009 + *yyy* + *yyy*	001	Source key label template length in bytes (*zzz*). Valid values are 0 and 64. Return an error if the length is 64 and a source key label is not provided.
010 + *yyy* + *yyy*	*zzz*	Source key label template (does not exist if *zzz* = 0). If a key label is identified by the *source_key_identifier* parameter, verify that the key label name matches this template. If the comparison fails, return an error. The source key label template must conform to the following rules: • The key label template must be 64 bytes in length • The first character cannot be in the range X'00' - X'1F', nor can it be X'FF' • The first character cannot be numeric (X'30' - X'39') • A key label name is terminated by a space character (X'20') on the right and must be padded on the right with space characters • The only special characters permitted are #, $, @, and * (X'23', X'24', X'40', and X'2A') • The wildcard X'2A' (*) is permitted only as the first character, the last character, or the only character in the template • Only alphanumeric characters (a - z, A - Z, 0 - 9), the four special characters (X'23', X'24', X'40', and X'2A'), and the space character (X'20') are allowed

Note: See "Number representation in trusted blocks" on page 880.

Trusted block section X'13'

Trusted block section X'13' contains the name (key label).

Trusted block section X'13'

The trusted block name section provides a 64-byte variable to identify the trusted block, just as key labels are used to identify other CCA keys. This name, or label, enables a host access-control system such as RACF to use the name to verify that the application has authority to use the trusted block.

Section X'13' is optional. No multiple sections are allowed. It has no subsections defined.

Table 254. Trusted block key label (name) section (X'13')

Offset (bytes)	Length (bytes)	Description
000	001	Section identifier: **X'13'** Trusted block name (key label)
001	001	Section version number (X'00').
002	002	Section length in bytes (68).
004	064	Name (key label).

Trusted block section X'14'

Trusted block section X'14' contains control and security information related to the trusted block.

This information section is separate from the public key and other sections because this section is required while the others are optional. This section contains the cryptographic information that guarantees its integrity and binds it to the local system.

Section X'14' is required. No multiple sections are allowed. Two subsections are defined.

Table 255. Trusted block information section (X'14')

Offset (bytes)	Length (bytes)	Description
000	001	Section identifier: **X'14'** Trusted block information
001	001	Section version number (X'00').
002	002	Section length in bytes (10+*xxx*).
004	002	Reserved, binary zero.
006	004	Flags: **Value Description** **X'00000000'** Trusted block is in the inactive state **X'00000001'** Trusted block is in the active state
010	*xxx*	Information section subsections (tag-length-value objects). One or two objects in TLV format.

Trusted block section X'14' subsections:

Section X'14' has two information subsections (tag-length-value objects) defined.

These subsections are summarized in Table 256. See also “Number representation in trusted blocks” on page 880.

Table 256. Summary of trusted block information subsections

Rule subsection tag	TLV object	Optional or required	Comments
X'0001'	Protection information	Required	Contains the encrypted 8-byte confounder and triple-length (24-byte) MAC key, the ISO-16609 TDES CBC MAC value, and the MKVP of the PKA master key (computed using MDC4).
X'0002'	Activation and expiration dates	Optional	Contains flags indicating whether or not the coprocessor is to validate dates, and contains the activation and expiration dates that are considered valid for the trusted block.

Trusted block section X'14' subsection X'0001'

Subsection X'0001' of the trusted block information section (X'14') is the protection information TLV object. This subsection is required. It contains the encrypted 8-byte confounder and triple-length (24-byte) MAC key, the ISO-16609 TDES CBC MAC value, and the MKVP of the PKA master key (computed using MDC4).

This subsection is defined in Table 257.

Table 257. Protection information subsection (X'0001') of trusted block information section (X'14')

Offset (bytes)	Length (bytes)	Description
000	002	Subsection tag: **X'0001'** Trusted block information TLV object
002	002	Subsection length in bytes (62).
004	001	Subsection version number (X'00'.
005	001	Reserved, must be binary zero.
006	032	Encrypted MAC key. Contains the encrypted 8-byte confounder and triple-length (24-byte) MAC key in the following format: **Offset Description** **00 - 07** Confounder **08 - 15** Left key **16 - 23** Middle key **24 - 31** Right key
038	008	MAC. Contains the ISO-16609 TDES CBC Message Authentication Code value.
046	016	MKVP. Contains the PKA master-key verification pattern, computed using MDC4, when the trusted block is in internal form, otherwise contains binary zero.

Trusted block section X'14' subsection X'0002'

Subsection X'0002' of the trusted block information section (X'14') is the activation and expiration dates TLV object. This subsection is optional. It contains flags indicating whether or not the coprocessor is to validate dates, and contains the activation and expiration dates that are considered valid for the trusted block.

This subsection is defined in Table 258.

Table 258. Activation and expiration dates subsection (X'0002') of trusted block information section (X'14')

Offset (bytes)	Length (bytes)	Description
000	002	Subsection tag: **X'0002'** Activation and expiration dates TLV object
002	002	Subsection length in bytes (16).
004	001	Subsection version number (X'00').
005	001	Reserved, must be binary zero.
006	002	Flags: **Value** **Description** **X'0000'** The coprocessor does not check dates. **X'0001'** The coprocessor checks dates. Compare the activation date (offset 008) and the expiration date (offset 012) to the coprocessor's internal real-time clock. Return an error if the coprocessor date is before the activation date or after the expiration date.
008	004	Activation date. Contains the first date that the trusted block can be used for generating or exporting keys. Format of the date is YYMDD, where: **YY** Big-endian year (return an error if greater than 9999) **MM** Month (return an error if any value other than X'01' - X'0C') **DD** Day of month (return an error if any value other than X'01' - X'1F'. Day must be valid for given month and year, including leap years). Return an error if the activation date is after the expiration date or is not valid.
012	004	Expiration date. Contains the last date that the trusted block can be used. Same format as activation date (offset 008). Return an error if date is not valid.

Trusted block section X'15'

Trusted block section X'15' contains application-defined data.

The trusted block application-defined data section can be used to include application-defined data in the trusted block. The purpose of the data in this section is defined by the application; it is neither examined nor used by CCA in any way.

Section X'15' is optional. No multiple sections are allowed. It has no subsections defined.

Table 259. Trusted block application-defined data section (X'15')

Offset (bytes)	Length (bytes)	Description
000	001	Section identifier: **X'15'** Application-defined data
001	001	Section version number (X'00').
002	002	Section length (6 + *xxx*)
004	002	Application data length The value of *xxx* must be between 0 and *N*, where *N* does not cause the overall length of the trusted block to exceed its maximum size of 3500 bytes.
006	*xxx*	Application-defined data Could be used to hold a public-key certificate for the trusted public key.

Chapter 18. Key forms and types used in the Key Generate verb

The Key Generate verb is the most complex of all the CCA verbs. You can generate different key forms and key types with the Key Generate verb.

Generating an operational key

There are different methods that you can use to generate an operational key.

Choose one of the following methods:

- For **operational keys,** call the Key Generate (CSNBKGN) verb. Table 43 on page 179 and Table 44 on page 179 show the key type and key form combinations for a single key and for a key pair.
- For **data-encrypting keys,** call the Random Number Generate (CSNBRNG) verb and specify the *form* parameter as ODD. Then pass the generated value to the Clear Key Import (CSNBCKI) verb or the Multiple Clear Key Import (CSNBCKM) verb. The DATA key type is now in operational form.

You cannot generate a PIN verification (PINVER) key in operational form because the originator of the PIN generation (PINGEN) key generates the PINVER key in exportable form, which is sent to you to be imported.

Generating an importable key

To generate an importable key form, call the Key Generate (CSNBKGN) verb.

If you want a DATA, MAC, PINGEN, DATAM, or DATAC key type in importable form, obtain it directly by generating a single key. If you want any other key type in importable form, request a key pair where either the first or second key type is importable (IM). Discard the generated key form that you do not need.

Generating an exportable key

To generate an exportable key form, call the Key Generate (CSNBKGN) verb.

If you want a DATA, MAC, PINGEN, DATAM, or DATAC key type in exportable form, obtain it directly by generating a single key. If you want any other key type in exportable form, request a key pair where either the first or second key type is exportable (EX). Discard the generated key form that you do not need.

Examples of single-length keys in one form only

An example of single-length keys.

```
Key   Key
Form   1

OP   DATA  Encipher or Decipher data. Use Data Key Export or Key Export
           to send encrypted key to another cryptograpic partner. Then
           communicate the ciphertext.
OP   MAC   MAC Generate. Because no MACVER key exists, there is no
           secure communication of the MAC with another cryptographic
           partner.
```

```
IM   DATA  Key Import, and then Encipher or Decipher. Then Key Export
           to communicate ciphertext and key with another cryptographic
           partner.
EX   DATA  You can send this key to a cryptographic partner, but you
           can do nothing with it directly. Use it for the key
           distribution service. The partner could then use Key Import
           to get it in operational form, and use it as in OP DATA
           above.
```

Examples of OPIM single-length, double-length, and triple-length keys in two forms

The first two letters of the key form indicate the form that key type 1 parameter is in, and the second two letters indicate the form that key type 2 parameter is in.

```
Key    Type Type
Form    1    2

OPIM   DATA DATA  Use the OP form in Encipher. Use Key Export with the
                  OP form to communicate ciphertext and key with
                  another cryptographic partner. Use Key Import at a
                  later time to use Encipher or Decipher with the same
                  key again.
OPIM   MAC  MAC   Single-length MAC Generate key. Use the OP form in
                  MAC Generate. You have no corresponding verb MACVER key,
                  but you can call the MAC Verify verb with
                  the MAC key directly. Use the Key Import verb
                  and then compute the MAC again using the MAC Verify
                  verb, which compares the MAC
                  it generates with the MAC supplied with the message
                  and issues a return code indicating whether they
                  compare.
```

Examples of OPEX single-length, double-length, and triple-length keys in two forms

Examples of OPEX single-length, double-length, and triple-length keys in two forms.

```
Key    Type  Type
Form   1      2

OPEX   DATA  DATA    Use the OP form in Encipher. Send the EX form and
                     the ciphertext to another cryptographic partner.
OPEX   MAC   MAC     Single-length MAC generation key. Use the OP form in
                     both MAC Generate and MAC Verify. Send the
                     EX form to a cryptographic partner to be used in the
                     MAC Generate or MAC Verify verbs.
OPEX   MAC   MACVER  Single-length MAC generation and MAC verification
                     keys. Use the OP form in MAC Generate. Send the EX
                     form to a cryptographic partner where it will be put
                     into Key Import, and then MAC Verify, with the
                     message and MAC that you have also transmitted.
OPEX   PINGEN PINVER Use the OP form in Clear PIN Generate. Send the
                     EX form to a cryptographic partner where it is put
                     into Key Import, and then Encrypted PIN Verify,
                     along with an IPINENC key.
OPEX   IMPORTER EXPORTER
                     Use the OP form in Key Import or Key Generate.
                     Send the EX form to a cryptographic partner where
                     it is used in Key Export, Data Key Export, or Key Generate,
                     or put in the CCA key storage file.
OPEX   EXPORTER IMPORTER
                     Use the OP form in Key Export, Data Key Export,
```

```
                    or Key Generate. Send the EX form to a cryptographic
                    partner where it is put into the CCA Key storage file
                    or used in Key Import or Key Generate.
```

When you and your partner have the OPEX IMPORTER EXPORTER, OPEX EXPORTER IMPORTER pairs of keys in "Examples of OPEX single-length, double-length, and triple-length keys in two forms" on page 894 installed, you can start key and data exchange.

Examples of IMEX single-length and double-length keys in two forms

Examples of IMEX single-length and double-length keys in two forms.

```
Key    Type    Type
Form   1        2

IMEX   DATA    DATA      Use the Key Import verb to import
                         IM form and use the OP form in Encipher. Send
                         the EX form to a cryptographic partner.
IMEX  MAC      MACVER    Use the Key Import verb to import
                         the IM form and use the OP form in MAC Generate.
                         Send the EX form to a cryptographic
                         partner who can verify the MAC.
IMEX IMPORTER  EXPORTER  Use the Key Import verb to import
                         the IM form and send the EX form to a
                         cryptographic partner. This establishes a new
                         IMPORTER/EXPORTER key between you and your
                         partner.
IMEX  PINGEN   PINVER    Use the Key Import verb to import
                         the IM form and send the EX form to a
                         cryptographic partner. This establishes a new
                         PINGEN/PINVER key between you and your partner.
```

Examples of EXEX single-length and double-length keys in two forms

Examples of IMEX single-length and double-length keys in two forms.

For the keys shown in the following list, you are providing key distribution services for other nodes in your network, or other cryptographic partners. Neither key type can be used in your installation.

```
Key    Type    Type
Form    1       2

EXEX  DATA     DATA     Send the first EX form to a cryptographic
EXEX  MAC      MACVER   partner with the corresponding IMPORTER and
EXEX  IMPORTER EXPORTER send the second EX form to another
EXEC  OPINENC  IPINENC  cryptographic partner with the corresponding
                        IMPORTER. This exchange establishes a key
                        between two partners.
```

Chapter 19. Control vectors and changing control vectors with the Control Vector Translate verb

A control vector table shows the default value of the control vector associated with each type of key.

This information unit also describes how to change control vectors with the Control Vector Translate verb

Control vector table

The control vector values that CCA uses to XOR key halves depend on the type of key.

Note: The control vectors descriptions here build on the descriptions used for earlier IBM products supporting CCA, each in turn: 4765, 4764, 4758, and TSS.

The master key enciphers all keys operational on your system. A transport key enciphers keys distributed off your system. Before a master key or transport key enciphers a key, CCA XORs both halves of the master key or transport key with a control vector. The same control vector is XORed to the left and right half of a master key or transport key.

Also, if you are entering a key part, CCA XORs each half of the key part with a control vector before placing the key part into the key storage file.

Each type of CCA key (except the master key) has either one or two unique control vectors associated with it. The master key or transport key CCA XORs with the control vector depending on the type of key the master key or transport key is enciphering. For double-length keys, a unique control vector exists for each half of a specific key type. For example, there is a control vector for the left half of an input PIN-encrypting key, and a control vector for the right half of an input PIN-encrypting key.

If you are entering a cleartext key part, CCA XORs the key part with the unique control vector(s) associated with the key type. CCA also enciphers the key part with two master key variants for a key part. One master key variant enciphers the left half of the key part and another master key variant enciphers the right half of the key part. CCA creates the master key variants for a key part by XORing the master key with the control vectors for key parts. These procedures protect key separation.

Table 260 on page 898 displays the default value of the control vector associated with each type of key. Some key types do not have a default control vector. For keys that are double-length, CCA enciphers using a unique control vector on each half.

Table 260. Default control vector values

Key Type	Control Vector Value (Hex) Value for Single-length Key or Left Half of Double-length Key	Control Vector Value (Hex) Value for Right Half of Double-length Key
AES	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
AESTOKEN	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
CIPHERXI	00 0C 50 00 03 C0 00 00	00 0C 50 00 03 A0 00 00
CIPHERXO	00 0C 60 00 03 C0 00 00	00 0C 60 00 03 A0 00 00
CIPHERXL	00 0C 71 00 03 C0 00 00	00 0C 71 00 03 A0 00 00
CIPHER	00 03 71 00 03 00 00 00	
CIPHER (double length)	00 03 71 00 03 41 00 00	00 03 71 00 03 21 00 00
CVARDEC	00 3F 42 00 03 00 00 00	
CVARENC	00 3F 48 00 03 00 00 00	
CVARPINE	00 3F 41 00 03 00 00 00	
CVARXCVL	00 3F 44 00 03 00 00 00	
CVARXCVR	00 3F 47 00 03 00 00 00	
DATA (external)	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
DATA (internal)	00 00 7D 00 03 41 00 00	00 00 7D 00 03 21 00 00
DATA	00 00 00 00 00 00 00 00	
DATAC	00 00 71 00 03 41 00 00	00 00 71 00 03 21 00 00
DATAM generation key (external)	00 00 4D 00 03 41 00 00	00 00 4D 00 03 21 00 00
DATAM key (internal)	00 05 4D 00 03 00 00 00	00 05 4D 00 03 00 00 00
DATAMV MAC verification key (external)	00 00 44 00 03 41 00 00	00 00 44 00 03 21 00 00
DATAMV MAC verification key (internal)	00 05 44 00 03 00 00 00	00 05 44 00 03 00 00 00
DATAXLAT	00 06 71 00 03 00 00 00	
DECIPHER	00 03 50 00 03 00 00 00	
DECIPHER (double-length)	00 03 50 00 03 41 00 00	00 03 50 00 03 21 00 00
DKYGENKY	00 71 44 00 00 03 41 00	00 71 44 00 03 21 00 00
DKYL0	This control vector has the DKYL0 set by default.	
DKYL1	00 72 44 00 00 03 41 00	00 71 44 00 03 21 00 00
DKYL2	00 74 44 00 00 03 41 00	00 71 44 00 03 21 00 00
DKYL3	00 77 44 00 00 03 41 00	00 71 44 00 03 21 00 00
DKYL4	00 78 44 00 00 03 41 00	00 71 44 00 03 21 00 00
DKYL5	00 7B 44 00 00 03 41 00	00 71 44 00 03 21 00 00
DKYL6	00 7D 44 00 00 03 41 00	00 71 44 00 03 21 00 00
DKYL7	00 7E 44 00 00 03 41 00	00 71 44 00 03 21 00 00
ENCIPHER	00 03 60 00 03 00 00 00	
ENCIPHER (double-length)	00 03 60 00 03 41 00 00	00 03 60 00 03 21 00 00
EXPORTER	00 41 7D 00 03 41 00 00	00 41 7D 00 03 21 00 00
IKEYXLAT	00 42 42 00 03 41 00 00	00 42 42 00 03 21 00 00

Table 260. Default control vector values (continued)

Key Type	Control Vector Value (Hex) Value for Single-length Key or Left Half of Double-length Key	Control Vector Value (Hex) Value for Right Half of Double-length Key
IMP-PKA	00 42 05 00 03 41 00 00	00 42 05 00 03 21 00 00
IMPORTER	00 42 7D 00 03 41 00 00	00 42 7D 00 03 21 00 00
IPINENC	00 21 5F 00 03 41 00 00	00 21 5F 00 03 21 00 00
MAC	00 05 4D 00 03 00 00 00	
MAC (double-length)	00 05 4D 00 03 41 00 00	00 05 4D 00 03 21 00 00
MACVER	00 05 44 00 03 00 00 00	
MACVER (double-length)	00 05 44 00 03 41 00 00	00 05 44 00 03 21 00 00
OKEYXLAT	00 41 42 00 03 41 00 00	00 41 42 00 03 21 00 00
OPINENC	00 24 77 00 03 41 00 00	00 24 77 00 03 21 00 00
PINGEN	00 22 7E 00 03 41 00 00	00 22 7E 00 03 21 00 00
PINVER	00 22 42 00 03 41 00 00	00 22 42 00 03 21 00 00
SECMSG with SMPIN set	00 0A 50 00 03 41 00 00	00 0A 50 00 03 21 00 00
SECMSG with SMKEY set	00 0A 60 00 03 41 00 00	00 0A 60 00 03 21 00 00

Note: The external control vectors for DATAC, DATAM MAC generation, and DATAMV MAC verification keys are also referred to as data compatibility control vectors.

Control-vector-base bit maps

Details of the control vector base bit maps.

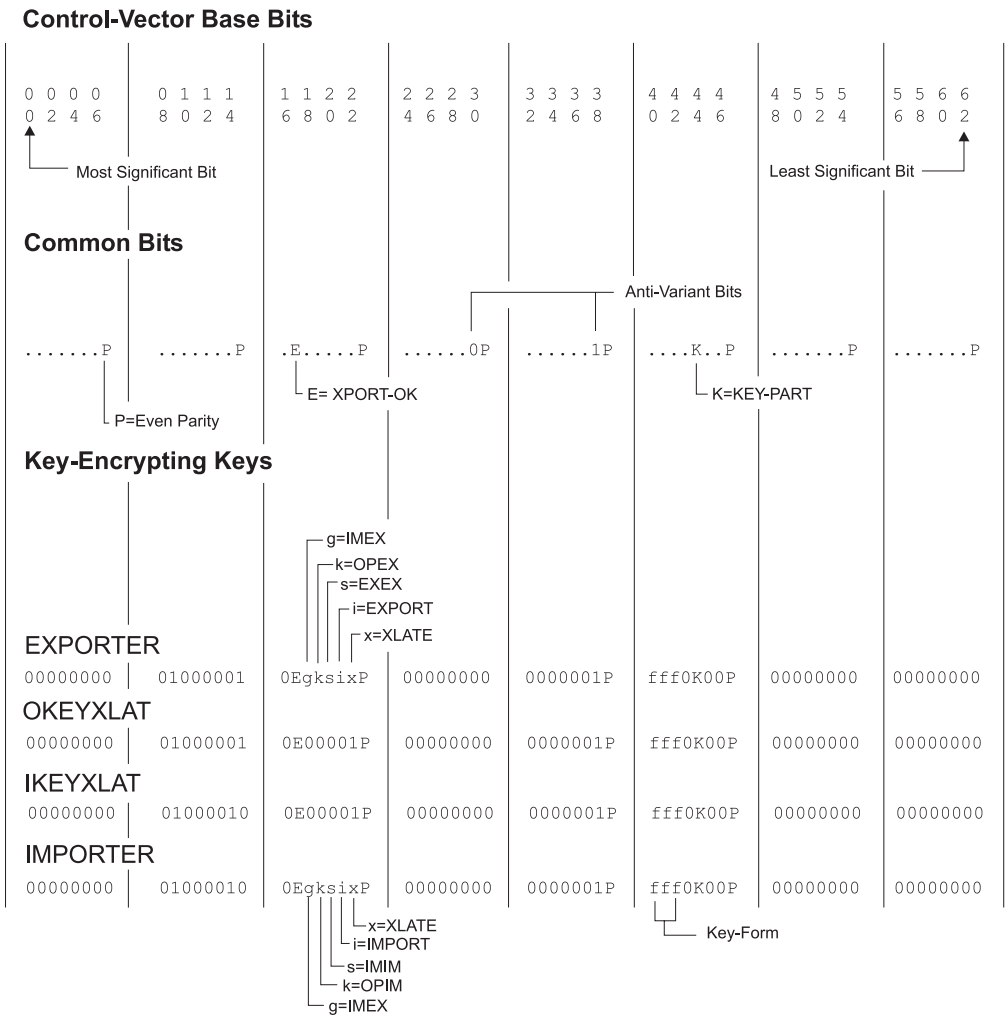


Figure 15. Control vector base bit map (common bits and key-encrypting keys)

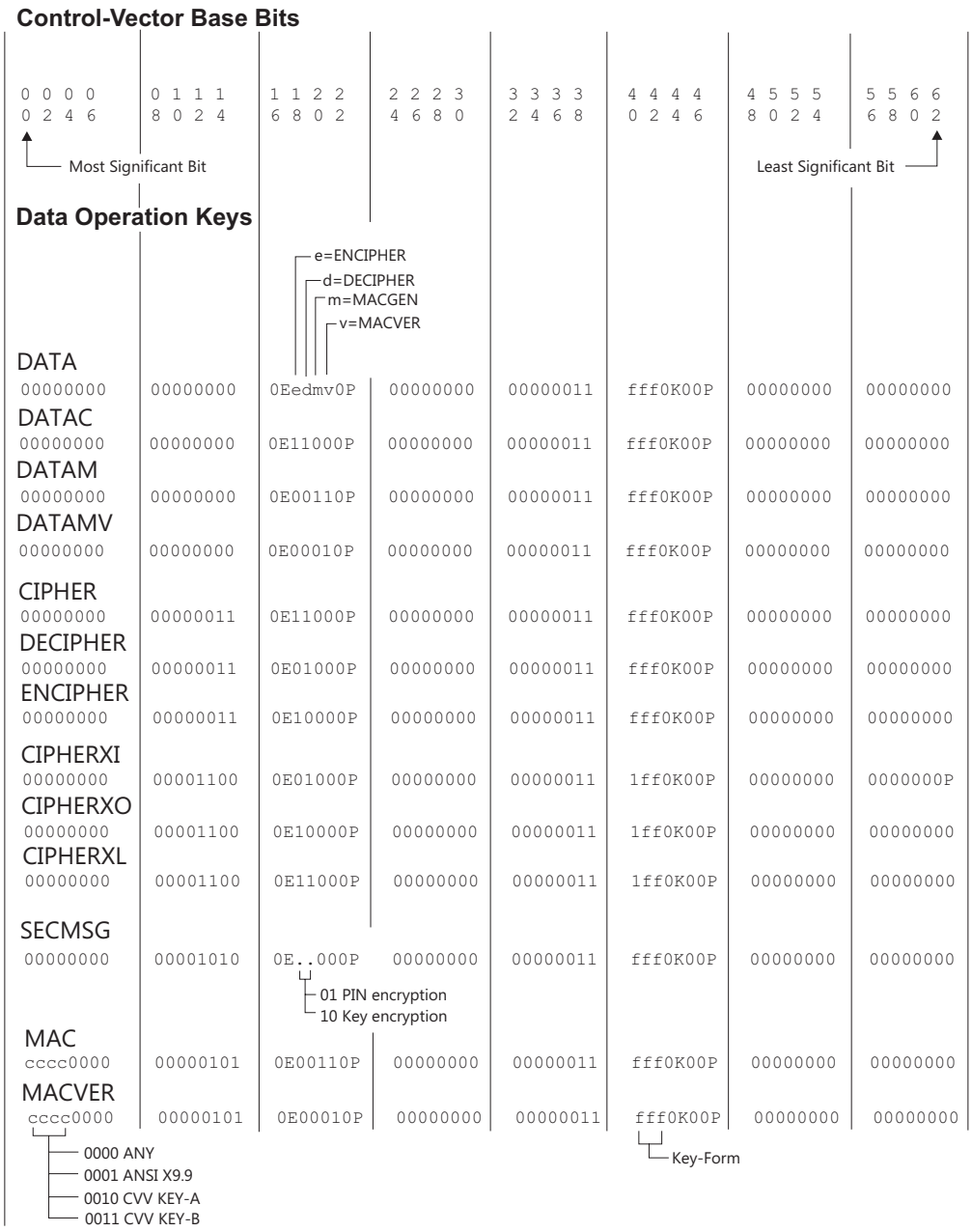


Figure 16. Control vector base bit map (data operation keys)

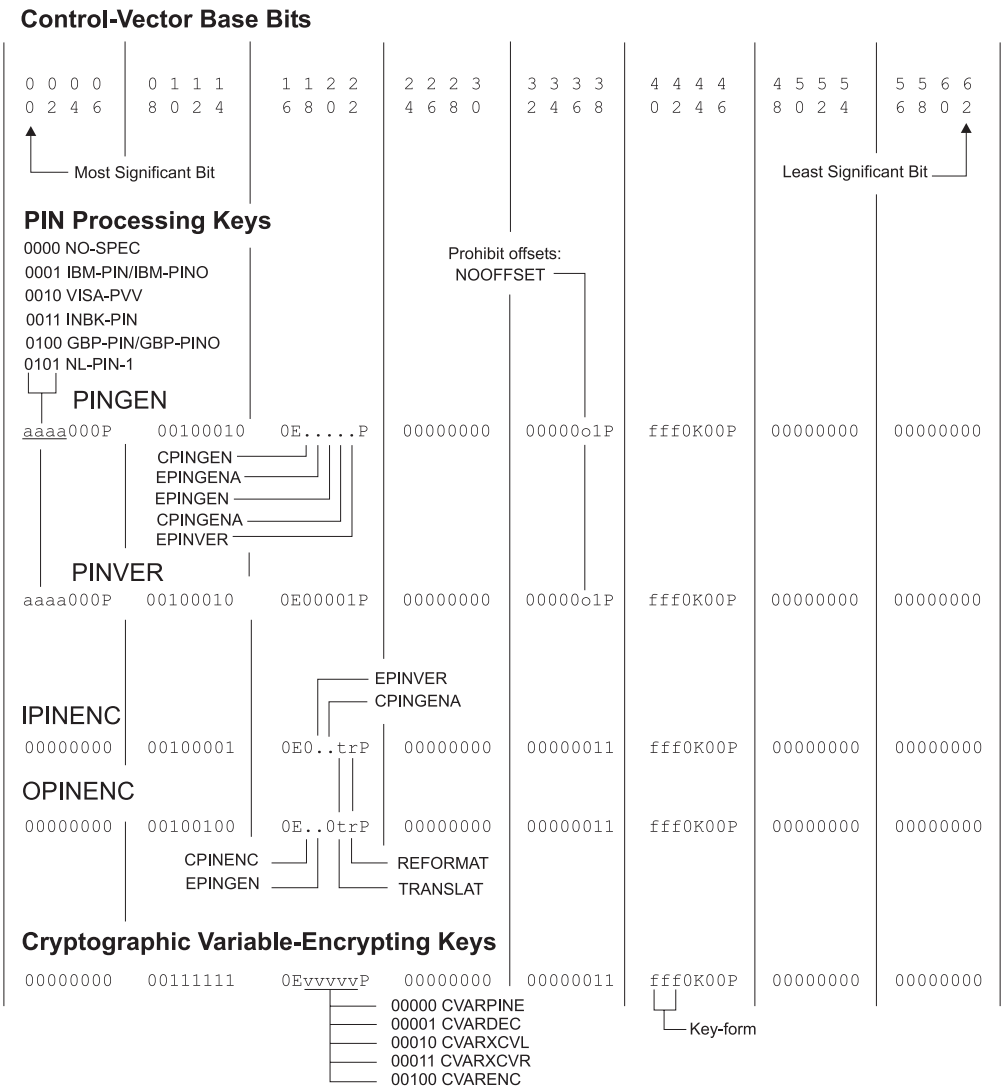


Figure 17. Control vector base bit map (PIN processing keys and cryptographic variable-encrypting keys)

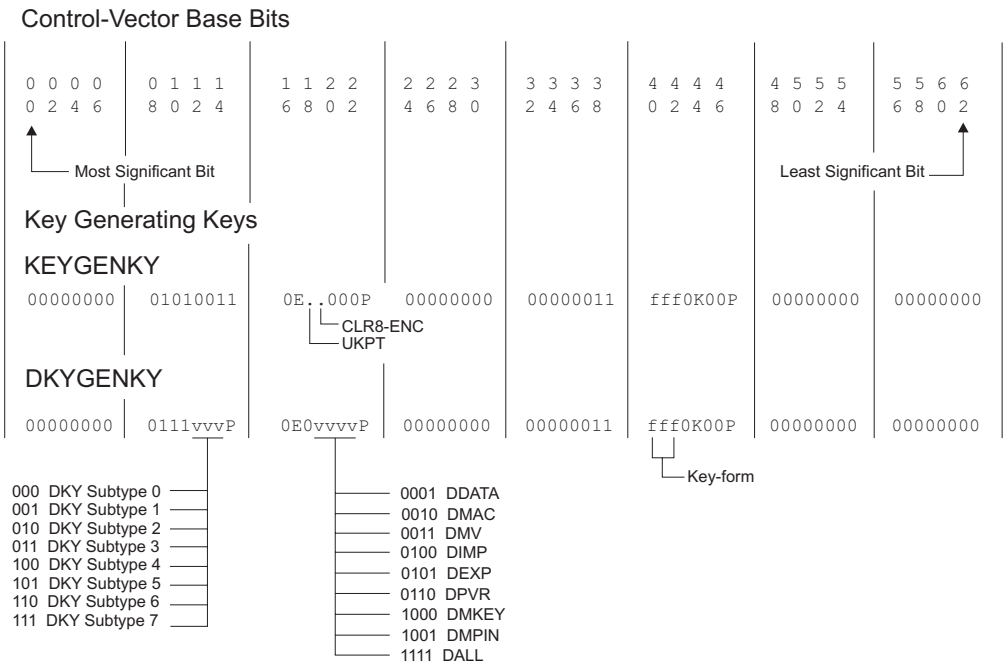


Figure 18. Control vector base bit map (key generating keys)

Key Form Bits, *fff*

The key form bits, 40-42, and for a double-length key, bits 104-106, are designated *fff* in the preceding illustration.

These bits can have the following values:

Value	Description
000	Single length key
010	Double length key, left half
001	Double length key, right half

The following values could exist in some CCA implementations:

Value	Description
110	Double-length key, left half, halves guaranteed unique
101	Double-length key, right half, halves guaranteed unique

Specifying a control-vector-base value

You can determine the value of a control vector by working through a series of questions.

Procedure

Work through this series of questions:

1. Begin with a field of 64 bits (eight bytes) set to B'0'. The most significant bit is referred to as bit 0. Define the key type and subtype (bits 8 - 14) as follows:

- The main key type bits (bits 8 - 11). Set bits 8 - 11 to one of the following values:

Table 261. Main key type bits

Bits 8 - 11	Main Key Type
0000	Data operation keys
0010	PIN keys
0011	Cryptographic variable-encrypting keys
0100	Key-encrypting keys
0101	Key-generating keys
0111	Diversified key-generating keys

- The key subtype bits (bits 12 - 14). Set bits 12 - 14 to one of the following values:

  Note: For Diversified Key Generating Keys, the subtype field specifies the hierarchical level of the DKYGENKY. If the subtype is nonzero, the DKYGENKY can generate only another DKYGENKY key with the hierarchy level decremented by one. If the subtype is zero, the DKYGENKY can generate only the final diversified key (a non-DKYGENKY key) with the key type specified by the usage bits.

Table 262. Key subtype bits

Bits 12 - 14	Key Subtype
Data Operation Keys	
000	Compatibility key (DATA)
001	Confidentiality key (CIPHER, DECIPHER, or ENCIPHER)
010	MAC key (MAC or MACVER)
101	Secure messaging keys
Key-Encrypting Keys	
000	Transport-sending keys (EXPORTER and OKEYXLAT)
001	Transport-receiving keys (IMPORTER and IKEYXLAT)
PIN Keys	
001	PIN-generating key (PINGEN, PINVER)
000	Inbound PIN-block decrypting key (IPINENC)
010	Outbound PIN-block encrypting key (OPINENC)
Cryptographic Variable-Encrypting Keys	
111	Cryptographic variable-encrypting key (CVAR....)
Diversified Key Generating Keys	
000	DKY Subtype 0
001	DKY Subtype 1
010	DKY Subtype 2
011	DKY Subtype 3
100	DKY Subtype 4
101	DKY Subtype 5
110	DKY Subtype 6

Table 262. Key subtype bits (continued)

Bits 12 - 14	Key Subtype
111	DKY Subtype 7

2. For key-encrypting keys, set the following bits:
   - The key-generating usage bits (gks, bits 18 - 20). Set the gks bits to B'111' to indicate the Key Generate verb can use the associated key-encrypting key to encipher generated keys when the Key Generate verb is generating various key-pair key-form combinations (see the Key-Encrypting Keys section of Figure 15 on page 900). Without any of the gks bits set to B'1', the Key Generate verb cannot use the associated key-encrypting key. The Key Token Build verb can set the gks bits to B'1' when you supply the **OPIM, IMEX, IMIM, OPEX**, and **EXEX** keywords.
   - The IMPORT and EXPORT bit and the XLATE bit (ix, bits 21 and 22). If the 'i' bit is set to B'1', the associated key-encrypting key can be used in the Data Key Import, Key Import, Data Key Export, and Key Export verbs. If the 'x' bit is set to B'1', the associated key-encrypting key can be used in the Key Translate and Key Translate2 verbs.
   - The key-form bits (fff, bits 40 - 42). The key-form bits indicate how the key was generated and how the control vector participates in multiple-enciphering. To indicate the parts can be the same value, set these bits to B'010'. For information about the value of the key-form bits in the right half of a control vector, see Step 8 on page 906.
3. For MAC and MACVER keys, set the following bits:
   - The MAC control bits (bits 20 and 21). For a MAC-generate key, set bits 20 and 21 to B'11'. For a MAC-verify key, set bits 20 and 21 to B'01'.
   - The key-form bits (fff, bits 40 - 42). For a single-length key, set the bits to B'000'. For a double-length key, set the bits to B'010'.
4. For PINGEN and PINVER keys, set the following bits:
   - The PIN calculation method bits (aaaa, bits 0 - 3). Set these bits to one of the following values:

Table 263. Calculation method keyword bits

Bits 0 - 3	Calculation Method Keyword	Description
0000	NO-SPEC	A key with this control vector can be used with any PIN calculation method.
0001	IBM-PIN or IBM-PINO	A key with this control vector can be used only with the IBM PIN or PIN Offset calculation method.
0010	VISA-PVV	A key with this control vector can be used only with the VISA-PVV calculation method.
0100	GBP-PIN or GBP-PINO	A key with this control vector can be used only with the German Banking Pool PIN or PIN Offset calculation method.
0011	INBK-PIN	A key with this control vector can be used only with the Interbank PIN calculation method.

- The prohibit-offset bit (o, bit 37) to restrict operations to the PIN value. If set to B'1', this bit prevents operation with the IBM 3624 PIN Offset calculation method and the IBM German Bank Pool PIN Offset calculation method.

5. For PINGEN, IPINENC, and OPINENC keys, set bits 18 - 22 to indicate whether the key can be used with the following verbs:

Table 264. INGEN, IPINENC, and OPINENC key bits

Service Allowed	Bit Name	Bit
Clear PIN Generate	CPINGEN	18
Encrypted PIN Generate Alternate	EPINGENA**	19
Encrypted PIN Generate	EPINGEN	20 for PINGEN 19 for OPINENC
Clear PIN Generate Alternate	CPINGENA	21 for PINGEN 20 for IPINENC
Encrypted PIN Verify	EPINVER	19
Clear PIN Encrypt	CPINENC	18
** EPINGENA is no longer supported, although the bit retains this definition for compatibility There is no Encrypted Pin Generate Alternate verb.		

6. For the IPINENC (inbound) and OPINENC (outbound) PIN-block ciphering keys, do the following:
   - Set the TRANSLAT bit (t, bit 21) to B'1' to permit the key to be used in the PIN Translate verb. The Control Vector Generate verb can set the TRANSLAT bit to B'1' when you supply the **TRANSLAT** keyword.
   - Set the REFORMAT bit (r, bit 22) to B'1' to permit the key to be used in the PIN Translate verb. The Control Vector Generate verb can set the REFORMAT bit and the TRANSLAT bit to B'1' when you supply the **REFORMAT** keyword.
7. For the cryptographic variable-encrypting keys (bits 18 - 22), set the variable-type bits (bits 18 - 22) to one of the following values:

Table 265. Generic key type bits

Bits 18 - 22	Generic Key Type	Description
00000	CVARPINE	Used in the Encrypted PIN Generate Alternate verb to encrypt a clear PIN.
00010	CVARXCVL	Used in the Control Vector Translate verb to decrypt the left mask array.
00011	CVARXCVR	Used in the Control Vector Translate verb to decrypt the right mask array.

8. For key-generating keys, set the following bits:
   - For KEYGENKY, set bit 18 for UKPT usage and bit 19 for CLR8-ENC usage.
   - For DKYGENKY, bits 12–14 will specify the hierarchical level of the DKYGENKY key. If the subtype CV bits are nonzero, the DKYGENKY can generate only another DKYGENKY key with the hierarchical level

decremented by one. If the subtype CV bits are zero, the DKYGENKY can generate only the final diversified key (a non-DKYGENKY key) with the key type specified by usage bits.

To specify the subtype values of the DKYGENKY, keywords DKYL0, DKYL1, DKYL2, DKYL3, DKYL4, DKYL5, DKYL6, and DKYL7 will be used.

- For DKYGENKY, bit 18 is reserved and must be zero.
- Usage bits 18-22 for the DKYGENKY key type are defined as follows. They will be encoded as the final key type that the DKYGENKY key generates.

Table 266. DKYGENKY key type bits

Bits 19 - 22	Keyword	Usage
0001	DDATA	DATA, DATAC, single or double length
0010	DMAC	MAC, DATAM
0011	DMV	MACVER, DATAMV
0100	DIMP	IMPORTER, IKEYXLAT
0101	DEXP	EXPORTER, OKEYXLAT
0110	DPVR	PINVER
1000	DMKEY	Secure message key for encrypting keys
1001	DMPIN	Secure message key for encrypting PINs
1111	DALL	All key types can be generated except **DKYGENKY** and **KEYGENKY** keys. Usage of the **DALL** keyword is controlled by a separate access control point.

9. For secure messaging keys, set the following bits:
   - Set bit 18 to B'1' if the key will be used in the secure messaging for PINs service. Set bit 19 to B'1' if the key will be used in the secure messaging for keys service.
10. For all keys, set the following bits:
    - The export bit (E, bit 17). If set to B'0', the export bit prevents a key from being exported. By setting this bit to B'0', you can prevent the receiver of a key from exporting or translating the key for use in another cryptographic subsystem. After this bit is set to B'0', it cannot be set to B'1' by any service other than Control Vector Translate. The Prohibit Export verb can reset the export bit.
    - The key-part bit (K, bit 44). Set the key-part bit to B'1' in a control vector associated with a key part. When the final key part is combined with previously accumulated key parts, the key-part bit in the control vector for the final key part is set to B'0'. The Control Vector Generate verb can set the key-part bit to B'1' when you supply the **KEY-PART** keyword.
    - The anti-variant bits (bit 30 and bit 38). Set bit 30 to B'0' and bit 38 to B'1'. Many cryptographic systems have implemented a system of variants where a 7-bit value is XORed with each 7-bit group of a key-encrypting key before enciphering the target key. By setting bits 30 and 38 to opposite values, control vectors do not produce patterns that can occur in variant-based systems.

- Control vector bits 64 - 127. If bits 40 - 42 are B'000' (single-length key), set bits 64 - 127 to B'0'. Otherwise, copy bits 0 - 63 into bits 64 - 127 and set bits 105 and 106 to B'01'.
- Set the parity bits (low-order bit of each byte, bits 7, 15, ..., 127). These bits contain the parity bits (P) of the control vector. Set the parity bit of each byte so the number of zero-value bits in the byte is an even number.
- For secure messaging keys, usage bit 18 on will enable the encryption of keys in a secure message and usage bit 19 on will enable the encryption of PINs in a secure message.

Changing control vectors with the Control Vector Translate verb

What you need to do when you use the Control Vector Translate verb.

About this task

Do the following when using the verb:

- Provide the control information for testing the control vectors of the source, target, and key-encrypting keys to ensure that only sanctioned changes can be performed
- Select the key-half processing mode.

Providing the control information for testing the control vectors

To minimize your security exposure, the Control Vector Translate verb requires control information (*mask array* information) to limit the range of allowable control vector changes.

To ensure that this verb is used only for authorized purposes, the source-key control vector, target-key control vector, and key-encrypting key (KEK) control vector must pass specific tests. The tests on the control vectors are performed within the secured cryptographic engine.

The tests consist of evaluating four logic expressions, the results of which must be a string of binary zeros. The expressions operate bitwise on information that is contained in the mask arrays and in the portions of the control vectors associated with the key or key-half that is being processed. If any of the expression evaluations do not result in all zero bits, the verb is ended with a *control vector violation* return and reason code (8/39). See Figure 19 on page 910. Only the 56-bit positions that are associated with a key value are evaluated. The low-order bit that is associated with key parity in each key byte is not evaluated.

Mask array preparation

A mask array consists of seven 8-byte elements: A_1, B_1, A_2, B_2, A_3, B_3, and B_4.

You choose the values of the array elements such that each of the following four expressions evaluates to a string of binary zeros. (See Figure 19 on page 910.) Set the **A** bits to the value you require for the corresponding control vector bits. In expressions 1 on page 909 through 3 on page 909, set the **B** bits to select the control vector bits to be evaluated. In expression 4 on page 909, set the **B** bits to select the source and target control vector bits to be evaluated. Also, use the following control vector information:

$\mathbf{C_1}$ is the control vector associated with the left half of the KEK.

C_2 is the control vector associated with the source key or selected source-key half/halves.

C_3 is the control vector associated with the target key or selected target-key half/halves.

1. (C_1 XOR A_1) logical-AND B_1

   This expression tests whether the KEK used to encipher the key meets your criteria for the desired translation.

2. (C_2 XOR A_2) logical-AND B_2

   This expression tests whether the control vector associated with the source key meets your criteria for the desired translation.

3. (C_3 XOR A_3) logical-AND B_3

   This expression tests whether the control vector associated with the target key meets your criteria for the desired translation.

4. (C_2 XOR C_3) logical-AND B_4

   This expression tests whether the control vectors associated with the source key and the target key meet your criteria for the desired translation.

Encipher two copies of the mask array, each under a different cryptographic-variable key (key type CVARENC). Use two different keys so the enciphered-array copies are unique values. When using the Control Vector Translate verb, the *mask_array_left* parameter and the *mask_array_right* parameter identify the enciphered mask arrays. The *array_key_left* parameter and the *array_key_right* parameter identify the internal keys for deciphering the mask arrays. The *array_key_left* parameter must have a key type of CVARXCVL and the *array_key_right* parameter must have a key type of CVARXCVR. The cryptographic process deciphers the arrays and compares the results; for the service to continue, the deciphered arrays must be equal. If the results are not equal, the service returns the return and reason code for data that is not valid (8/385).

Use the Key Generate verb to create the key pairs CVARENC-CVARXCVL and CVARENC-CVARXCVR. Each key in the key pair must be generated for a different node. The CVARENC keys are generated for, or imported into, the node where the mask array will be enciphered. After enciphering the mask array, you should destroy the enciphering key. The CVARXCVL and CVARXCVR keys are generated for, or imported into, the node where the Control Vector Translate verb will be performed.

If using the **BOTH** keyword to process both halves of a double-length key, remember that bits 41, 42, 104, and 105 are different in the left and right halves of the CCA control vector and must be ignored in your mask-array tests (that is, make the corresponding **B_2** and/or **B_3** bits equal to zero).

When the control vectors pass the masking tests, the verb does the following:

- Deciphers the source key. In the decipher process, the service uses a key that is formed by the XOR of the KEK and the control vector in the key token variable the *source_key_token* parameter identifies.
- Enciphers the deciphered source key. In the encipher process, the verb uses a key that is formed by the XOR of the KEK and the control vector in the key token variable the *target_key_token* parameter identifies.
- Places the enciphered key in the key field in the key token variable the *target_key_token* parameter identifies.

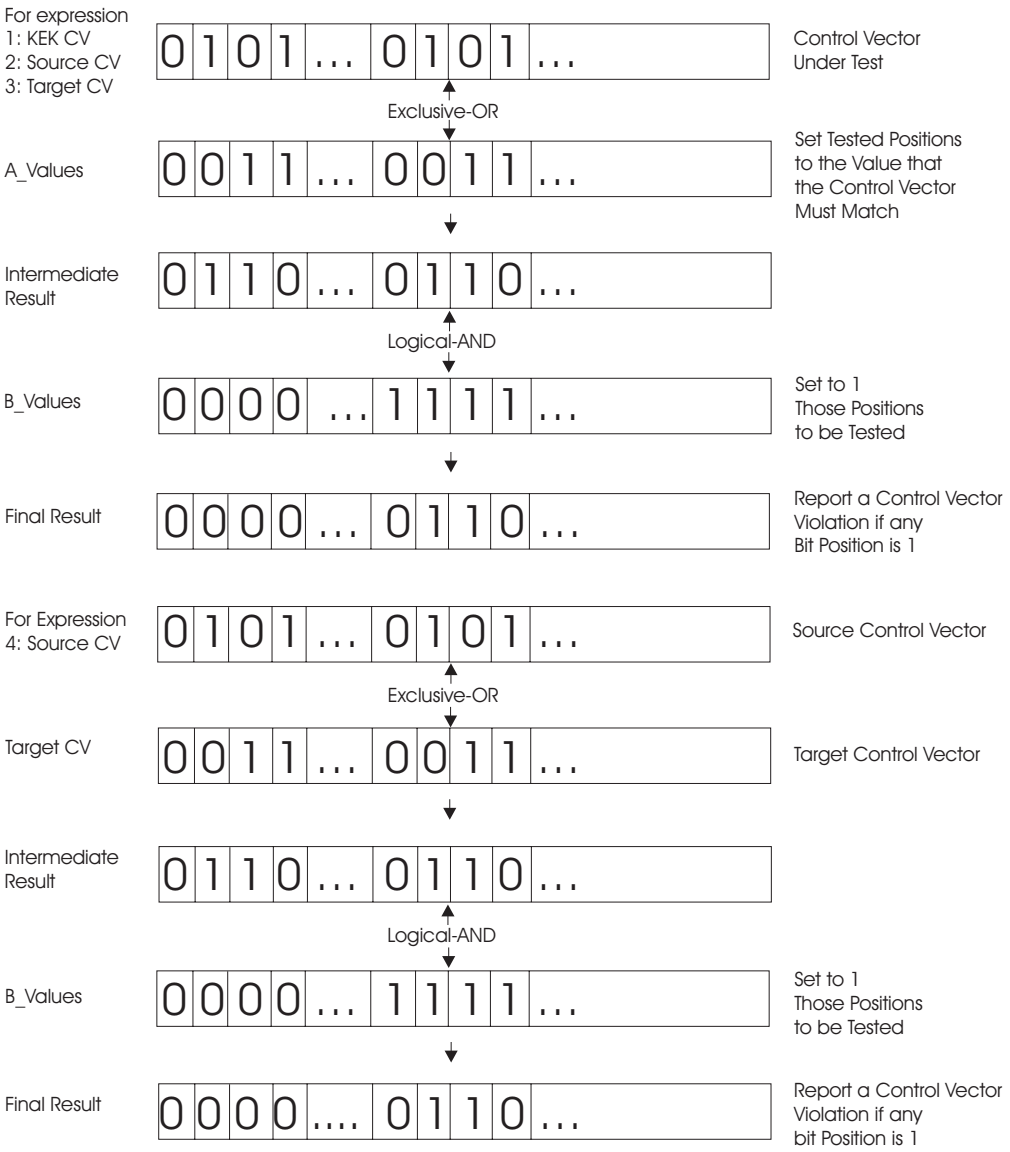


Figure 19. Control Vector Translate verb mask_array processing

Selecting the key-half processing mode

Use the Control Vector Translate verb to change a control vector associated with a key.

rule_array keywords determine which key halves are processed in the call, as shown in Figure 20 on page 911.

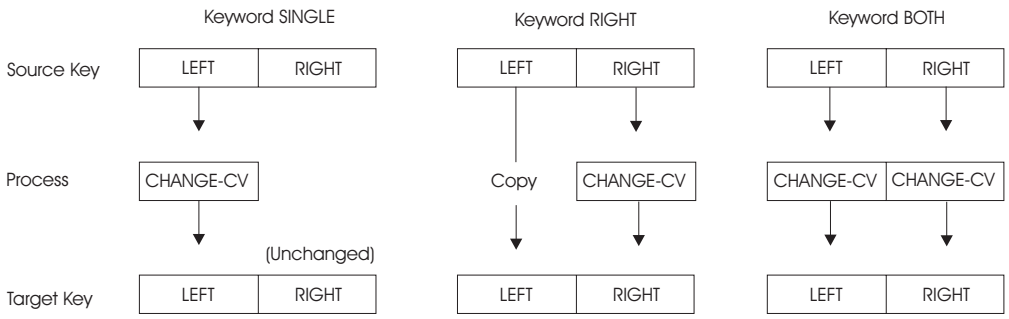


Figure 20. Control Vector Translate verb

Keyword
Description

SINGLE
This keyword causes the control vector of the left half of the source key to be changed. The updated key half is placed into the left half of the target key in the target key token. The right half of the target key is unchanged.

The **SINGLE** keyword is useful when processing a single-length key or when first processing the left half of a double-length key (to be followed by processing the right half).

RIGHT
This keyword causes the control vector of the right half of the source key to be changed. The updated key half is placed into the right half of the target key of the target key token. The left half of the source key is copied unchanged into the left half of the target key in the target key token.

BOTH This keyword causes the control vector of both halves of the source key to be changed. The updated key is placed into the target key in the target key token.

A single set of control information must permit the control vector changes applied to each key half. Normally, control vector bit positions 41, 42, 105, and 106 are different for each key half. Therefore, set bits 41 and 42 to B'00' in mask array elements B_1, B_2, and B_3.

You can verify that the source and target key tokens have control vectors with matching bits in bit positions 40-42 and 104-106, the "form field" bits. Ensure bits 40-42 of mask array B_4 are set to B'111'.

LEFT This keyword enables you to supply a single-length key and obtain a double-length key. The source key token must contain:

- The KEK-enciphered single-length key
- The control vector for the single-length key (often this is a null value)
- A control vector, stored in the source token where the right-half control vector is normally stored, used in decrypting the single-length source key when the key is being processed for the target right half of the key.

The verb first processes the source and target tokens as with the **SINGLE** keyword. Then the source token is processed using the single-length enciphered key and the source token right-half control vector to obtain the actual key value. The key value is then enciphered using the KEK and the control vector in the target token for the right-half of the key.

This approach is frequently of use when you must obtain a double-length CCA key from a system that supports only a single-length key, for example when processing PIN keys or key-encrypting keys received from non-CCA systems.

To prevent the verb from ensuring each key byte has odd parity, you can specify the **NOADJUST** keyword. If you do not specify the **NOADJUST** keyword, or if you specify the **ADJUST** keyword, the verb ensures each byte of the target key has odd parity.

When the target key-token CV is null

When you use any of the **LEFT**, **BOTH**, or **RIGHT** keywords, and when the control vector in the target key token is null (all B'0'), bit 3 in byte 59 will be set to B'1' to indicate this is a double-length DATA key.

Control vector translate example

As an example, consider the case of receiving a single-length PIN-block encrypting key from a non-CCA system.

Often such a key will be encrypted by an unmodified transport key (no control vector or variant is used). In a CCA system, an inbound PIN encrypting key is double-length.

First use the Key Token Build verb to insert the single-length key value into the left-half key-space in a key token. Specify **USE-CV** as a key type and a control vector value set to 16 bytes of X'00'. Also specify **EXTERNAL**, **KEY**, and **CV** keywords in the *rule_array*. This key token will be the source key key-token.

Second, the target key token can also be created using the Key Token Build verb. Specify a key type of **IPINENC** and the **NO-EXPORT** *rule_array* keyword.

Then call the Control Vector Translate verb and specify a *rule_array* keyword of **LEFT**. The mask arrays can be constructed as follows:

- A_1 is set to the value of the KEK's control vector, most likely the value of an IMPORTER key, perhaps with the NO-EXPORT bit set. B_1 is set to eight bytes of X'FF' so all bits of the KEK's control vector will be tested.
- A_2 is set to eight bytes of X'00', the (null) value of the source key control vector. B_2 is set to eight bytes of X'FF' so all bits of the source-key “control vector” will be tested.
- A_3 is set to the value of the target key's left-half control vector. B_3 is set to X'FFFF FFFF FF9F FFFF'. This will cause all bits of the control vector to be tested except for the two (“fff”) bits used to distinguish between the left-half and right-half target-key control vector.
- B_4 is set to eight bytes of X'00' so no comparison is made between the source and target control vectors.

Chapter 20. PIN formats and algorithms

Personal identification number (PIN) notation, PIN block formats, PIN extraction rules, and PIN algorithms.

For PIN calculation procedures, see *IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface Reference*.

PIN notation

The content of PIN blocks are assigned a letter that represents the content.

The following notations describe the contents of PIN blocks:

P = A 4-bit decimal digit that is one digit of the PIN value.

C = A 4-bit hexadecimal control value. The valid values are X'0', X'1', and X'2'.

L = A 4-bit hexadecimal value that specifies the number of PIN digits. This value is in the range 4 - 12.

F = A 4-bit field delimiter of value X'F'.

f = A 4-bit delimiter filler that is either P or F, depending on the length of the PIN.

D = A 4-bit decimal padding value. All pad digits in the PIN block have the same value.

X = A 4-bit hexadecimal padding value. All pad digits in the PIN block have the same value.

x = A 4-bit hexadecimal filler that is either P or X, depending on the length of the PIN.

R = A 4-bit hexadecimal random digit. The sequence of R digits can each take a different value.

r = A 4-bit random filler that is either P or R, depending on the length of the PIN.

Z = A 4-bit hexadecimal zero (X'0').

z = A 4-bit zero filler that is either P or Z, depending on the length of the PIN.

S = A 4-bit hexadecimal digit that constitutes one digit of a sequence number.

A = A 4-bit decimal digit that constitutes one digit of a user-specified constant.

PIN block formats

All PIN block formats have a code assigned to them.

ANSI X9.8

This format is also named ISO format 0, VISA format 1, VISA format 4, and ECI format 1.

```
   P1 = CLPPPPfffffffffFF

   P2 = ZZZZAAAAAAAAAAAA

   PIN Block = P1 XOR P2

where C = X'0'
      L = X'4' to X'C'
```

Programming Note: The rightmost 12 digits (excluding the check digit) in P2 are the rightmost 12 digits of the account number for all formats except VISA format 4. For VISA format 4, the rightmost 12 digits (excluding the check digit) in P2 are the leftmost 12 digits of the account number.

ISO Format 1

Example code for ISO Format 1.

This format is also named ECI format 4.

```
   PIN Block = CLPPPPrrrrrrrrRR

where C = X'1'
      L = X'4' to X'C'
```

ISO Format 2

Example code for ISO Format 2.

```
   PIN Block = CLPPPPfffffffffFF

where C = X'2'
      L = X'4' to X'C'
```

ISO Format 3

The formats of the intermediate PIN-block, the PAN block, and the ISO-3 PIN-block.

An ISO-3 PIN-block format is equivalent to the ANSI X9.8, VISA-1, and ECI-1 PIN-block formats in length. A PIN that is longer than 12 digits is truncated on the right.

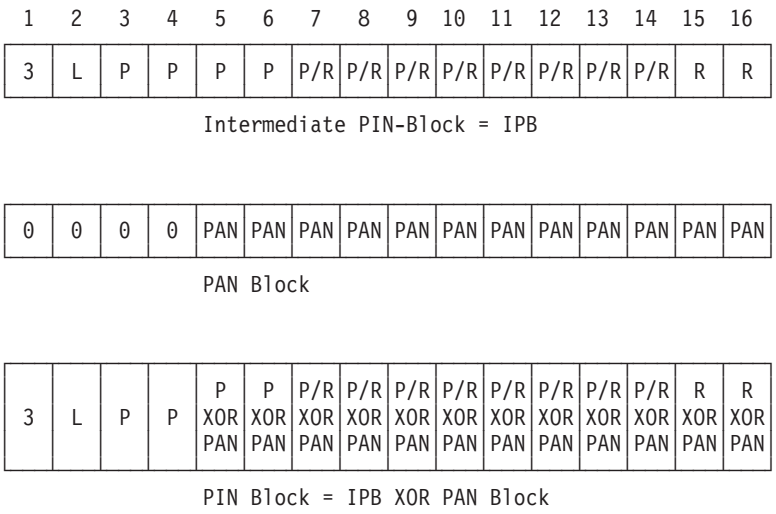


Figure 21. ISO-3 PIN-block format

where:

3 Is the value X'3' for ISO-3.

L Is the length of the PIN, which is a 4-bit value from X'4' - X'C'.

P Is a PIN digit, which is a 4-bit value from X'0' - X'9'. The values of the PIN digits are independent.

P/R Is a PIN digit or pad value. A PIN digit has a 4-bit value from X'0' - X'9'. A pad value has a random 4-bit value of X'A' - X'F'. The number of pad values in the intermediate PIN block (IPB) is from 2 - 10.

R Is the random value X'A' - X'F' for the pad value.

PAN Is twelve 4-bit digits that represent one of the following:

- The rightmost 12 digits of the primary account-number (excluding the check digit) if the format of the PIN block is ISO-3, ANSI X9.8, VISA-1, or ECI-1.
- The leftmost 12 digits of the primary account-number (excluding the check digit) if the format of the PIN block is VISA-4.

Each PAN digit has a value from X'0' - X'9'.

The PIN block is the result of XORing the 64-bit IPB with the 64-bit PAN block.

Example:

```
L = 6, PIN = 123456, Personal Account Number = 111222333444555
36123456AFBECDDC : IPB
0000222333444555 : PAN block for ISO-3 (ANSI X9.8, VISA-1, ECI-1) format
361216759CFA8889 : PIN block for ISO-3 (ANSI X9.8, VISA-1, ECI-1) format
```

Visa Format 2

Example code for Visa Format 2.

```
PIN Block = LPPPPzzDDDDDDDDD

where L = X'4' to X'6'
```

Visa Format 3

Example code for Visa Format 3.

This format specifies that the PIN length can be in the range of 4 - 12 digits. The PIN starts from the leftmost digit and ends by the delimiter ('F'), and the remaining digits are padding digits.

An example of a 6-digit PIN:

```
PIN Block = PPPPPPFXXXXXXXXX
```

IBM 3624 Format

This format requires the program to specify the delimiter, X, for determining the PIN length.

```
PIN Block = PPPPxxxxxxxxXXXX
```

IBM 3621 Format

This format requires the program to specify the delimiter, X, for determining the PIN length.

```
PIN Block = SSSSPPPPxxxxxxxx
```

ECI Format 2

This format defines the PIN to be 4 digits.

```
PIN Block = PPPPRRRRRRRRRRRR
```

ECI Format 3

Example code for ECI Format 3.

```
PIN Block = LPPPPzzRRRRRRRRR

where L = X'4' to X'6'
```

PIN extraction rules

There are PIN extraction rules for the Encrypted PIN Verify and Encrypted PIN Translate verbs.

Encrypted PIN Verify verb

This verb extracts the customer-entered PIN from the input PIN block.

It extracts the PIN according to the following rules:

- If the input PIN block format is ANSI X9.8, ISO format 0, VISA format 1, VISA format 4, ECI format 1, ISO format 1, ISO format 2, ISO format 3, VISA format 2, IBM Encrypting PINPAD format, or ECI format 3, the verb extracts the PIN according to the length specified in the PIN block.
- If the input PIN block format is VISA format 3, the specified delimiter (padding) determines the PIN length. The search starts at the leftmost digit in the PIN block. If the input PIN block format is 3624, the specification of a PIN extraction method for the 3624 is supported through *rule_array* keywords. If no PIN extraction method is specified in the *rule_array*, the specified delimiter (padding) determines the PIN length.
- If the input PIN block format is 3621, the specification of a PIN extraction method for the 3621 is supported through *rule_array* keywords. If no PIN extraction method is specified in the *rule_array*, the specified delimiter (padding) determines the PIN length.
- If the input PIN block format is ECI format 2, the PIN is the leftmost 4 digits.

For the VISA algorithm, if the extracted PIN length is less than 4, the verb sets a reason code that indicates verification failed. If the length is greater than or equal to 4, the verb uses the leftmost 4 digits as the referenced PIN.

For the IBM German Banking Pool algorithm, if the extracted PIN length is not 4, the verb sets a reason code that indicates verification failed.

For the IBM 3624 algorithm, if the extracted PIN length is less than the PIN check length, the verb sets a reason code that indicates verification failed.

Clear PIN Generate Alternate verb

This verb extracts the customer-entered PIN from the input PIN block.

It extracts the PIN from the input PIN block according to the following rules:

- This verb supports the specification of a PIN extraction method for the 3624 and 3621 PIN block formats through the use of the *rule_array* keyword. The *rule_array* points to an array of one or two 8-byte elements. The first element in the *rule_array* specifies the PIN calculation method. The second element in the *rule_array* (if specified) indicates the PIN extraction method. Refer to the "Clear PIN Generate Alternate (CSNBCPA)" on page 468 for an explanation of PIN extraction method keywords.

Encrypted PIN Translate verb

This verb extracts the customer-entered PIN from the input PIN block.

It extracts the PIN from the input PIN block according to the following rules:

- If the input PIN block format is ANSI X9.8, ISO format 0, VISA format 1, VISA format 4, ECI format 1, ISO format 1, ISO format 2, ISO format 3, VISA format 2, IBM Encrypting PINPAD format, or ECI format 3 and, if the specified PIN length is less than 4, the verb sets a reason code to reject the operation. If the specified PIN length is greater than 12, the operation proceeds to normal completion with unpredictable contents in the output PIN block. Otherwise, the verb extracts the PIN according to the specified length.
- If the input PIN block format is VISA format 3, the specified delimiter (padding) determines the PIN length. The search starts at the leftmost digit in the PIN block. If the input PIN block format is 3624, the specification of a PIN extraction method for the 3624 is supported through *rule_array* keywords. If no PIN extraction method is specified in the *rule_array*, the specified delimiter (padding) determines the PIN length.
- If the input PIN block format is 3621, the specification of a PIN extraction method for the 3621 is supported through *rule_array* keywords. If no PIN extraction method is specified in the *rule_array*, the specified delimiter (padding) determines the PIN length.
- If the input block format is ECI format 2, the PIN is always the leftmost 4 digits.

If the maximum PIN length allowed by the output PIN block is shorter than the extracted PIN, only the leftmost digits of the extracted PIN that form the allowable maximum length are placed in the output PIN block. The PIN length field in the output PIN block, it if exists, specifies the allowable maximum length.

IBM PIN algorithms

There are several PIN algorithms, including the IBM PIN generation algorithms, IBM PIN offset generation algorithm, and IBM PIN verification algorithms.

3624 PIN generation algorithm

This algorithm generates an n-digit PIN based on account-related data or person-related data, namely the validation data. The assigned PIN length parameter specifies the length of the generated PIN.

The algorithm requires the following input parameters:

- A 64-bit validation data
- A 64-bit decimalization table
- A 4-bit assigned PIN length
- A 128-bit PIN-generation key

The service uses the PIN generation key to encipher the validation data. Each digit of the enciphered validation data is replaced by the digit in the decimalization table whose displacement from the leftmost digit of the table is the same as the value of the digit of the enciphered validation data. The result is an intermediate PIN. The leftmost n digits of the intermediate PIN are the generated PIN, where n is specified by the assigned PIN length.

Figure 22 illustrates the 3624 PIN generation algorithm.

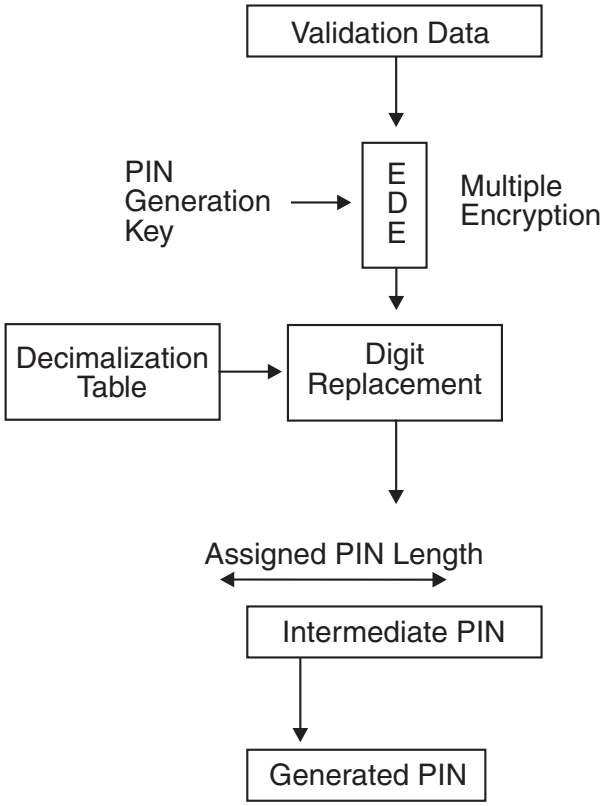


Figure 22. 3624 PIN generation algorithm

German Banking Pool PIN Generation algorithm

This algorithm generates a 4-digit PIN based on account-related data or person-related data, namely the validation data.

The algorithm requires the following input parameters:

- A 64-bit validation data
- A 64-bit decimalization table
- A 128-bit PIN-generation key

The validation data is enciphered using the PIN generation key. Each digit of the enciphered validation data is replaced by the digit in the decimalization table whose displacement from the leftmost digit of the table is the same as the value of the digit of enciphered validation data. The result is an intermediate PIN. The rightmost 4 digits of the leftmost 6 digits of the intermediate PIN are extracted. The leftmost digit of the extracted 4 digits is checked for zero. If the digit is zero, the digit is changed to one; otherwise, the digit remains unchanged. The resulting four digits is the generated PIN.

Figure 23 illustrates the German Banking Pool (GBP) PIN generation algorithm.

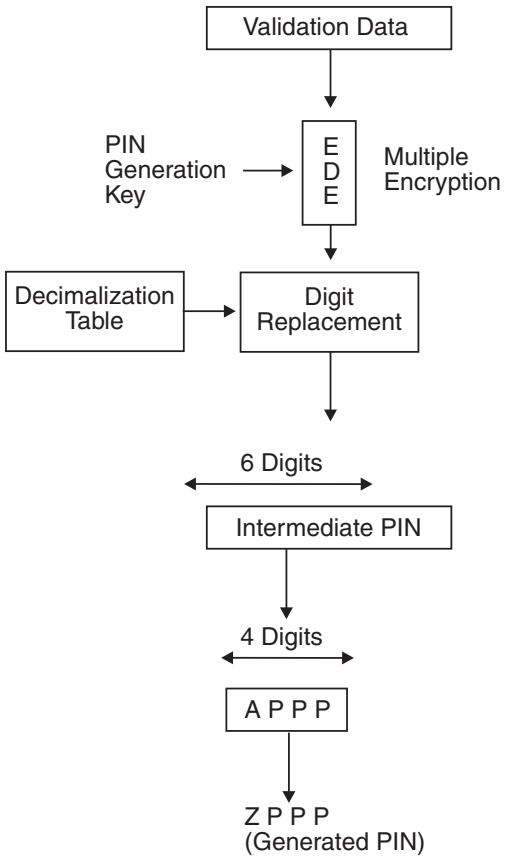


Figure 23. GBP PIN generation algorithm

PIN offset generation algorithm

To allow the customer to select his own PIN, a PIN offset is used by the IBM 3624 and GBP PIN generation algorithms to relate the customer-selected PIN to the generated PIN.

The PIN offset generation algorithm requires two parameters in addition to those used in the 3624 PIN generation algorithm. They are a customer-selected PIN and a 4-bit PIN check length. The length of the customer-selected PIN is equal to the assigned-PIN length, n.

The "3624 PIN generation algorithm" on page 918 is performed. The offset data value is the result of subtracting (modulo 10) the leftmost n digits of the intermediate PIN from the customer-selected PIN. The modulo 10 subtraction ignores borrows. The rightmost m digits of the offset data form the PIN offset, where m is specified by the PIN check length. Note that n cannot be less than m. To generate a PIN offset for a GBP PIN, m is set to 4 and n is set to 6.

Figure 24 illustrates the PIN offset generation algorithm.

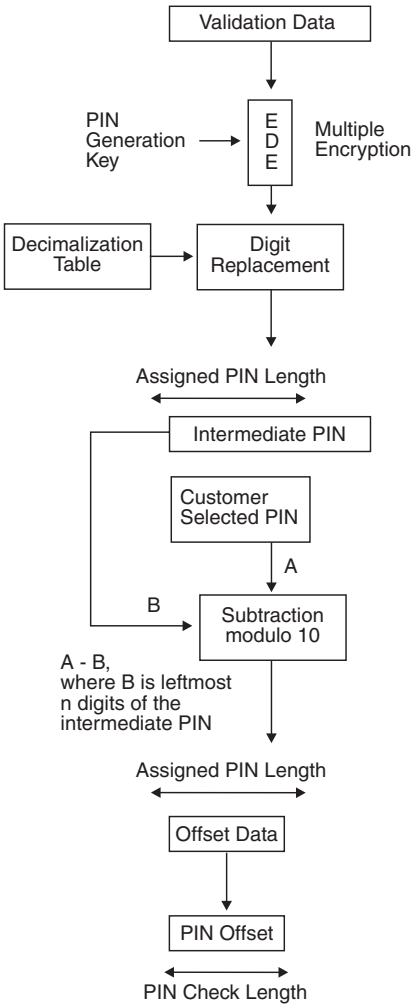


Figure 24. PIN-Offset generation algorithm

3624 PIN verification algorithm

This algorithm generates an intermediate PIN based on the specified validation data. A part of the intermediate PIN is adjusted by adding an offset data. A part of the result is compared with the corresponding part of the customer-entered PIN.

The algorithm requires the following input parameters:

- A 64-bit validation data
- A 64-bit decimalization table
- A 128-bit PIN-verification key
- A 4-bit PIN check length
- An offset data
- A customer-entered PIN

The rightmost m digits of the offset data form the PIN offset, where m is the PIN check length.

1. The validation data is enciphered using the PIN verification key. Each digit of the enciphered validation data is replaced by the digit in the decimalization table whose displacement from the leftmost digit of the table is the same as the value of the digit of enciphered validation data.
2. The leftmost n digits of the result is added (modulo 10) to the offset data value, where n is the length of the customer-entered PIN. The modulo 10 addition ignores carries.
3. The rightmost m digits of the result of the addition operation form the PIN check number. The PIN check number is compared with the rightmost m digits of the customer-entered PIN. If they match, PIN verification is successful; otherwise, verification is unsuccessful.

When a nonzero PIN offset is used, the length of the customer-entered PIN is equal to the assigned PIN length.

Figure 25 on page 922 illustrates the PIN verification algorithm.

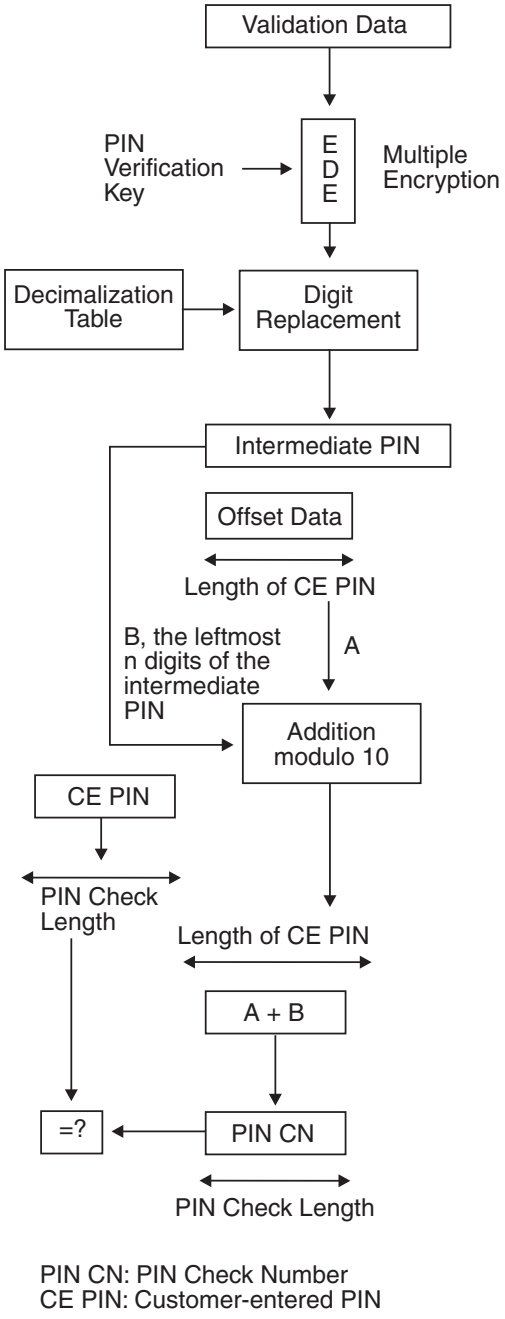


Figure 25. PIN verification algorithm

German Banking Pool PIN Verification algorithm

This algorithm generates an intermediate PIN based on the specified validation data.

A part of the intermediate PIN is adjusted by adding an offset data. A part of the result is extracted. The extracted value might or might not be modified before it compares with the customer-entered PIN.

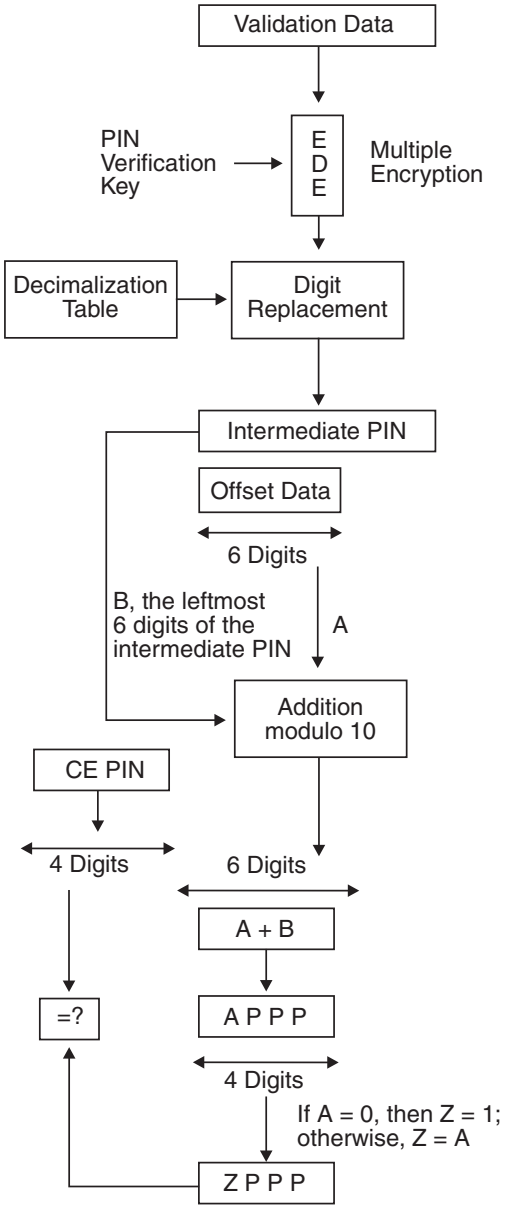
The algorithm requires the following input parameters:

- A 64-bit validation data
- A 64-bit decimalization table
- A 128-bit PIN verification key
- An offset data
- A customer-entered PIN

The rightmost 4 digits of the offset data form the PIN offset.

1. The validation data is enciphered using the PIN verification key. Each digit of the enciphered validation data is replaced by the digit in the decimalization table whose displacement from the leftmost digit of the table is the same as the value of the digit of enciphered validation data.
2. The leftmost 6 digits of the result is added (modulo 10) to the offset data. The modulo 10 addition ignores carries.
3. The rightmost 4 digits of the result of the addition (modulo 10) are extracted.
4. The leftmost digit of the extracted value is checked for zero. If the digit is zero, the digit is set to one; otherwise, the digit remains unchanged. The resulting four digits are compared with the customer-entered PIN. If they match, PIN verification is successful; otherwise, verification is unsuccessful.

Figure 26 on page 924 illustrates the GBP PIN verification algorithm.



CE PIN: Customer-entered PIN

Figure 26. GBP PIN verification algorithm

VISA PIN algorithms

The VISA PIN verification algorithm performs a multiple encipherment of a value, called the transformed security parameter (TSP), and a extraction of a 4-digit PIN verification value (PVV) from the ciphertext.

The calculated PVV is compared with the referenced PVV and stored on the plastic card or data base. If they match, verification is successful.

PVV Generation algorithm

The algorithm generates a 4-digit PIN verification value (PVV) based on the transformed security parameter (TSP).

The algorithm requires the following input parameters:

- A 64-bit TSP
- A 128-bit PVV generation key

1. A multiple encipherment of the TSP using the double-length PVV generation key is performed.
2. The ciphertext is scanned from left to right. Decimal digits are selected during the scan until four decimal digits are found. Each selected digit is placed from left to right according to the order of selection. If four decimal digits are found, those digits are the PVV.
3. If, at the end of the first scan, less than four decimal digits have been selected, a second scan is performed from left to right. During the second scan, all decimal digits are skipped and only non-decimal digits can be processed. Non-decimal digits are converted to decimal digits by subtracting 10. The process proceeds until four digits of PVV are found.

Figure 27 illustrates the PVV generation algorithm.

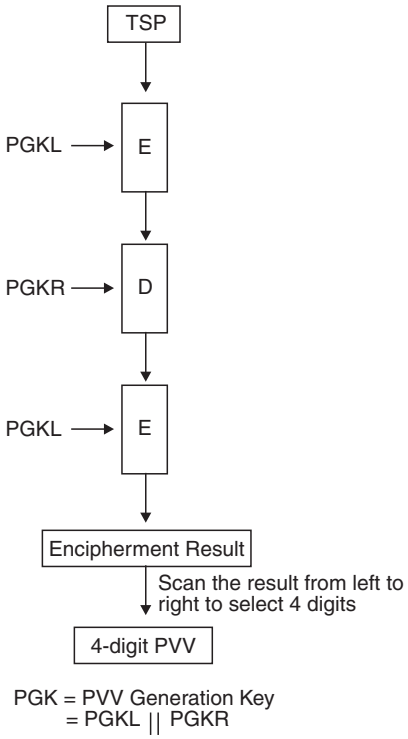


Figure 27. PVV generation algorithm

Programming Note: For VISA PVV algorithms, the leftmost 11 digits of the TSP are the personal account number (PAN), the leftmost 12th digit is a key table index to select the PVV generation key, and the rightmost 4 digits are the PIN. The key table index should have a value in the range 1 - 6.

PVV Verification algorithm

The PVV verification algorithm requires specific parameters.

The algorithm requires the following input parameters:

- A 64-bit TSP
- A 16-bit referenced PVV
- A 128-bit PVV verification key

A PVV is generated using the PVV generation algorithm, except a PVV verification key rather than a PVV generation key is used. The generated PVV is compared with the referenced PVV. If they match, verification is successful.

Interbank PIN Generation algorithm

A description of the Interbank PIN calculation method.

The Interbank PIN calculation method consists of the following steps:

1. Let X denote the *transaction_security* parameter element converted to an array of 16 4-bit numeric values. This parameter consists of (in the following sequence) the 11 rightmost digits of the customer PAN (excluding the check digit), a constant of 6, a 1-digit key indicator, and a 3-digit validation field.
2. Encrypt X with the double-length PINGEN (or PINVER) key to get 16 hexadecimal digits (64 bits).
3. Perform decimalization on the result of the previous step by scanning the 16 hexadecimal digits from left to right, skipping any digit greater than X'9' until 4 decimal digits (for example, digits that have values from X'0' - X'9') are found.

   If all digits are scanned but 4 decimal digits are not found, repeat the scanning process, skipping all digits that are X'9' or less and selecting the digits that are greater than X'9'. Subtract 10 (X'A') from each digit selected in this scan.

   If the 4 digits that were found are all zeros, replace the 4 digits with 0100.
4. Concatenate and use the resulting digits for the Interbank PIN. The 4-digit PIN consists of the decimal digits in the sequence in which they are found.

Chapter 21. Cryptographic algorithms and processes

CCA algorithms and processes include key verification algorithms, data-encryption processes, as well as key format and encryption processes.

These processing details are described for these aspects:

Cryptographic key-verification techniques

The key-verification implementations described in this document employ several mechanisms for assuring the integrity and value of the key.

Information is presented about these topics:

Master-key verification algorithms

The CEX*C implementations employ triple-length DES and PKA master keys (three DES keys) that are internally represented in 24 bytes (168 bits).

Beginning with Release 4.1.0, the CEX*C employs an APKA master key represented in 32 bytes (256 bits). Verification patterns on the contents of the new, current, and old master-key registers can be generated and verified when the selected register is not in the empty state. For the AES master key, the SHA-256 verification method is used.

The CEX*C employ several verification pattern generation methods.

SHA-1 based master-key verification method

A SHA-1 hash algorithm is calculated on the quantity X'01' prepended to the 24-byte register contents.

The resulting 20-byte hash value is used in the following ways:

- The Key Test and Key Test2 verb2 uses the first eight bytes of the 20-byte hash value as the random_number variable, and uses the second eight bytes as the verification_pattern.
- A SHA-1 based master-key verification pattern stored in a two-byte or an eight-byte master-key verification pattern field in a key token consists of the first two or the first eight bytes of the calculated SHA-1 value, respectively.

z/OS-based master-key verification method

When the first and third portions of the symmetric master key have the same value, the master key is effectively a double-length DES key.

In this case, the master-key verification pattern (MKVP) is based on this algorithm:

- C = X'4545454545454545'
- IR = $MK_{first\text{-}part}$ XOR $e_C(MK_{first\text{-}part})$
- MKVP = $MK_{second\text{-}part}$ XOR $e_{IR}(MK_{second\text{-}part})$

where:

- $e_x(Y)$ is the DES encoding of Y using x as a key
- XOR means bitwise exclusive OR

Version X'00' internal CCA DES key tokens use this eight-byte master-key verification pattern.

SHA-256 based master-key verification method

A SHA-256 hash algorithm is calculated on the quantity X'01' prep-ended to the 24-byte register contents.

For AES, there will be verification patterns for both the AES master key and for AES operational keys that are used to encipher or decipher data. The verification pattern on the master key is called the MKVP. The verification pattern on operational keys is referred to as a key-verification pattern (KVP).

Both the MKVP and KVP for AES will use the same algorithm. Both will be computed with the following process.

1. Compute the SHA-256 hash of the string formed by prepending the byte X'01' to the cleartext key value.
2. Take the leftmost eight bytes of the hash as the verification pattern.

This value is truncated to eight bytes because this is the length allocated for the verification in several CCA structures and APIs. For example, the AES key token has eight bytes for the MKVP, and the Key Test and Key Test2 verbs have an eight-byte parameter for the verification pattern.

Asymmetric master key MDC-based verification method

The verification pattern for the asymmetric master keys is based on hashing the value of the master-key using the MDC-4 hashing algorithm.

The master key is not parity adjusted.

The RSA private key sections X'06' and X'08' use this 16-byte master-key version number.

Key-token verification patterns

These verification pattern techniques are used in the several types of CCA key tokens.

The techniques are:

- AES and ECC key tokens: leftmost 8 bytes of SHA-256 hash of the string formed by pre-pending X'01' to the cleartext key value.
- DES key tokens:

  - Triple-length master key, key token version X'00': leftmost 8 bytes of SHA-1 hash
  - Triple-length master key, key token version X'03': leftmost 2 bytes of SHA-1 hash
  - Double-length master key, key token version X'00': leftmost 8 bytes of z/OS hash
  - Double-length master key, key token version X'03': leftmost 2 bytes of SHA-1 hash
- RSA key tokens:
  - Private-key section types X'06' and X'08': 16-byte MDC-4 value
  - Private-key section types X'02' and X'05': leftmost 2 bytes of SHA-1 hash
- Trusted blocks: 16-byte MDC-4 value

CCA DES-key verification algorithm

The cryptographic engines provide a method for verifying the value of a DES cryptographic key or key part without revealing information about the value of the key or key part.

The CCA verification method first creates a random number. A one-way cryptographic function combines the random number with the key or key part. The verification method returns the result of this one-way cryptographic function (the *verification pattern*) and the random number.

Note: A one-way cryptographic function is a function in which it is easy to compute the output from a given input, but it is not computationally feasible to compute the input given an output.

For information about how you can use an application program to invoke this verification method, see "Key Test (CSNBKYT)" on page 199.

The CCA DES key verification algorithm does the following:

1. Sets *KKR'* = *KKR* XOR *RN*
2. Sets *K1* = X'4545454545454545'
3. Sets *X1* = DES encoding of *KKL* using key *K1*
4. Sets *K2* = *X1* XOR *KKL*
5. Sets *X2* = DES encoding of *KKR'* using key *K2*
6. Sets *VP* = *X2* XOR *KKR'*

where:

RN Is the random number generated or provided

KKL Is the value of the single-length key, or is the left half of the double-length key

KKR Is XL8'00' if the key is a single-length key, or is the value of the right half of the double-length key

VP Is the verification pattern

Encrypt zeros AES-key verification algorithm

The cryptographic engine provides a method for verifying the value of an AES cryptographic key or key part without revealing information about the value of the key or key part.

In this method, the AES key data encryption algorithm encodes a 128-bit value that is all zero bits. The leftmost 32 bits of the result are compared to the trial input value or returned from the Key Test2 verb in an 8-byte variable that is padded with bits valued to zero.

Encrypt zeros DES-key verification algorithm

The cryptographic engine provides a method for verifying the value of a DES cryptographic key or key part without revealing information about the value of the key or key part.

In this method the single-length or double-length key DEA encodes a 64-bit value that is all zero bits. The leftmost 32 bits of the result are compared to the trial input value or returned from the Key Test and Key Test2 verbs.

For a single-length key, the key DEA encodes an 8-byte, all-zero-bits value.

For a double-length key, the key DEA triple-encodes an 8-byte, all-zero-bits value. The left half (high-order half) key encodes the zero-bit value, this result is DEA decoded by the right key half, and that result is DEA encoded by the left key half.

SHAVP1 algorithm

This algorithm is used by the Key Test2 callable service to generate and verify the verification pattern.

```
VP = Trunc128( SHA256( KA || KT || KL || K ))
```

where:

VP Is the 128-bit verification pattern

TruncN(*x*)
Is truncation of the string *x* to the left most **N** bits

SHA256(*x*)
is the SHA-256 hash of the string *x*

KA Is the one-byte CCA variable-length key token constant for the algorithm of key (HMAC X'03')

KT Is the two-byte CCA variable-length key token constant for the type of key (MAC X'0002')

KL Is the two-byte bit length of the clear key value

K Is the clear key value left-aligned and padded on the right with binary zeros to byte boundary

|| Is string concatenation

Modification Detection Code calculation

The Modification Detection Code (MDC) calculation method defines a one-way cryptographic function.

A one-way cryptographic function is a function in which it is easy to compute the input into output (a digest) but very difficult to compute the output into input. MDC uses DES encryption only and a default key of X'5252 5252 5252 5252 2525 2525 2525 2525'.

The MDC Generate verb supports four versions of the MDC calculation method that you specify by using one of the keywords shown in Table 267. All versions use the MDC-1 calculation.

Table 267. Versions of the MDC calculation method

Keyword	Version of the MDC calculation
MDC-2, **PADMDC-2**	Specifies two encipherments for each 8-byte input data block. These versions use the MDC-2 calculation procedure described in Table 268.
MDC-4, **PADMDC-4**	Specifies four encipherments for each 8-byte input data block. These versions use the MDC-4 calculation procedure described in Table 268.

When the keywords **PADMDC-2** and **PADMDC-4** are used, the supplied text is *always* padded as follows:

- If the total supplied text is less than 16 bytes in length, pad bytes are appended to make the text length equal to 16 bytes. A length of zero is allowed.
- If the total supplied text is a minimum of 16 bytes in length, pad bytes are appended to make the text length equal to the next-higher multiple of eight bytes. One or more pad bytes are always added.
- All appended pad bytes, other than the last pad byte, are set to X'FF'.
- The last pad byte is set to a binary value equal to the count of all appended pad bytes (X'01' - X'10').

Use the resulting pad text in the Table 268. The MDC Generate verb uses these MDC calculation methods. See “MDC Generate (CSNBMDG)” on page 393 for more information.

Table 268. MDC calculation procedures

Calculation	Procedure

MDC-1	`MDC-1(KD1, KD2, IN1, IN2, OUT1, OUT2);`
	` Set KD1mod := set KD1 bit 1 to B'1' and bit 2 to B'0' (bits 0-7)`
	` Set KD2mod := set KD2 bit 1 to B'0' and bit 2 to B'1' (bits 0-7)`
	` Set F1 := IN1 XOR eKD1mod(IN1)`
	` Set F2 := IN2 XOR eKD2mod(IN2)`
	` Set OUT1 := (bits 0..31 of F1) || (bits 32..63 of F2)`
	` Set OUT2 := (bits 0..31 of F2) || (bits 32..63 of F1)`
	`End procedure`

MDC-2	`MDC-2(n, text, KEY1, KEY2, MDC);`
	` For i := 1, 2, ..., n do`
	`  Call MDC-1(KEY1, KEY2, T8<i>, T8<i>, OUT1, OUT2)`
	`  Set KEY1 := OUT1`
	`  Set KEY2 := OUT2`
	` End do`
	` Set output MDC := (KEY1 || KEY2)`
	`End procedure`

Table 268. MDC calculation procedures (continued)

Calculation	Procedure

MDC-4	`MDC-4(n, text, KEY1, KEY2, MDC);`
	`  For i := 1, 2, ..., n do`
	`  Call MDC-1(KEY1, KEY2, T8<i>, T8<i>, OUT1, OUT2)`
	`  Set KEY1int := OUT1`
	`  Set KEY2int := OUT2`
	`  Call MDC-1(KEY1int, KEY2int, KEY2, KEY1, OUT1, OUT2)`
	`  Set KEY1 := OUT1`
	`  Set KEY2 := OUT2`
	`  End do`
	`  Set output MDC := (KEY1 || KEY2)`
	`End procedure`

Notation: **eK(X)** DES encryption of plaintext X using key K **||** Concatenation operation **XOR** Exclusive-OR operation **:=** Assignment operation **T8<1>** First 8-byte block of text **T8<2>** Second 8-byte block of text **KD1, KD2** 64-bit quantities **IN1, IN2** 64-bit quantities **OUT1, OUT2** 64-bit quantities **n** Number of 8-byte blocks	

Ciphering methods

The Data Encryption Standard (DES) algorithm defines operations on 8-byte data strings.

The DES algorithm is used in many different processes within CCA:

- Encrypting and decrypting general data
- Triple-encrypting and triple-decrypting PIN blocks
- Triple-encrypting and triple-decrypting CCA DES keys
- Triple-encrypting and triple-decrypting RSA private keys with several processes
- Deriving keys, hashing data, generating CVV values, and so forth

The Encipher and Decipher verbs describe how you can request encryption or decryption of application data. See the following topic: “General data-encryption processes” on page 933 for a description of the two standardized processes you can use.

In CCA, PIN blocks are encrypted with double-length keys. The PIN block is encrypted with the left-half key, for which the result is decrypted with the right-half key and this result is encrypted with the left-half key.

See “Triple-DES ciphering algorithms” on page 936 and “Ciphering methods,” which describe how CCA DES keys are enciphered.

General data-encryption processes

Although the fundamental concepts of enciphering and deciphering data are simple, different methods exist to process data strings that are not a multiple of eight bytes in length.

Two widely used methods for enciphering general data are defined in these ANSI standards:

- ANSI X3.106 cipher block chaining (CBC)
- ANSI X9.23

These methods also differ in how they define the initial chaining value (ICV).

This section describes how the Encipher and Decipher verbs implement these methods.

Single-DES and Triple-DES encryption algorithms for general data

Using the CEX*C, you can use the triple-DES algorithm in addition to the classical single-DES algorithm.

In the subsequent descriptions of the CBC method and ANSI X9.23 method, the actions of the Encipher and Decipher verbs encompass both single-DES and triple-DES algorithms. The triple-DES processes are depicted in Figure 28 where "left key" and "right key" refer to the two halves of a double-length DES key.

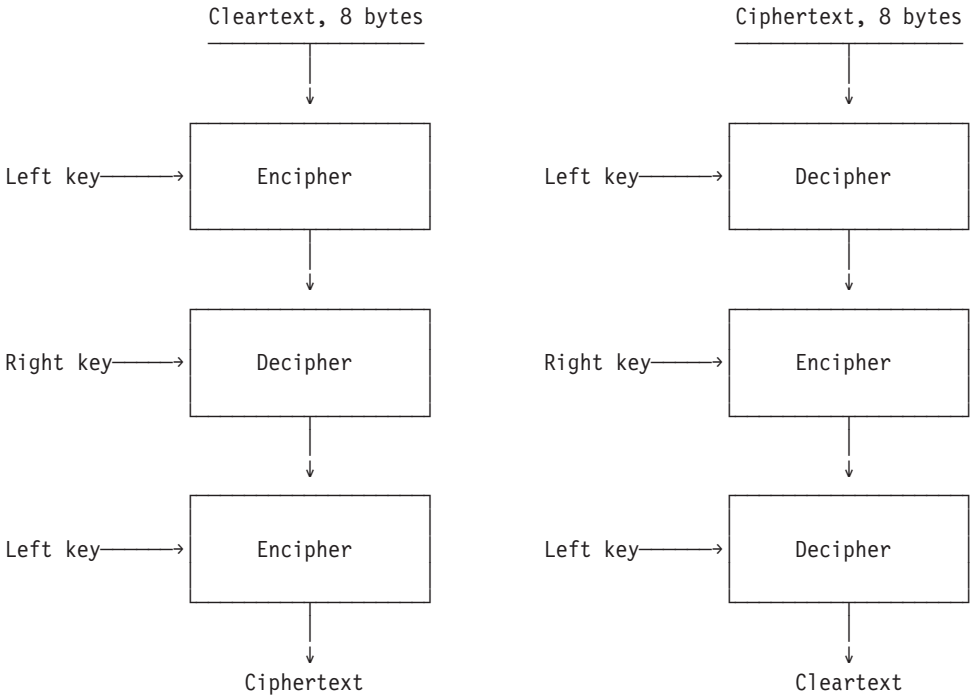


Figure 28. Triple-DES data encryption and decryption

ANSI X3.106 Cipher Block Chaining (CBC) method

ANSI standard X3.106 defines four modes of operation for ciphering, and one of these modes, Cipher Block Chaining (CBC), defines the basic method for ciphering multiple 8-byte data strings.

Figure 29 and Figure 30 on page 935 show CBC using the Encipher and Decipher verbs. A plaintext data string that must be a multiple of eight bytes is processed as a series of 8-byte blocks. The ciphered result from processing an 8-byte block is XORed with the next block of 8 input bytes. The last 8-byte ciphered result is defined as an output chaining value (OCV). The security server stores the OCV in bytes 0 - 7 of the *chaining_vector* variable.

An ICV is XORed with the first block of eight bytes. When you call the Encipher or Decipher verb, specify the **INITIAL** or **CONTINUE** keywords. If you specify the **INITIAL** keyword, the default, the initialization vector from the verb parameter is XORed with the first eight bytes of data. If you specify the **CONTINUE** keyword, the OCV identified by the *chaining_vector* parameter is XORed with the first eight bytes of data.

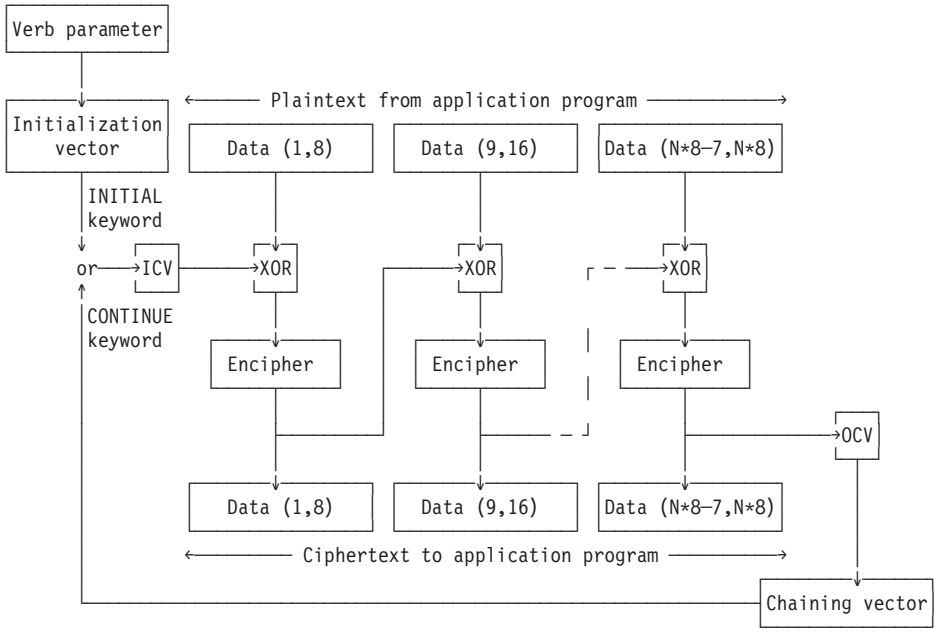


Figure 29. Enciphering using the ANSI X3.106 CBC method

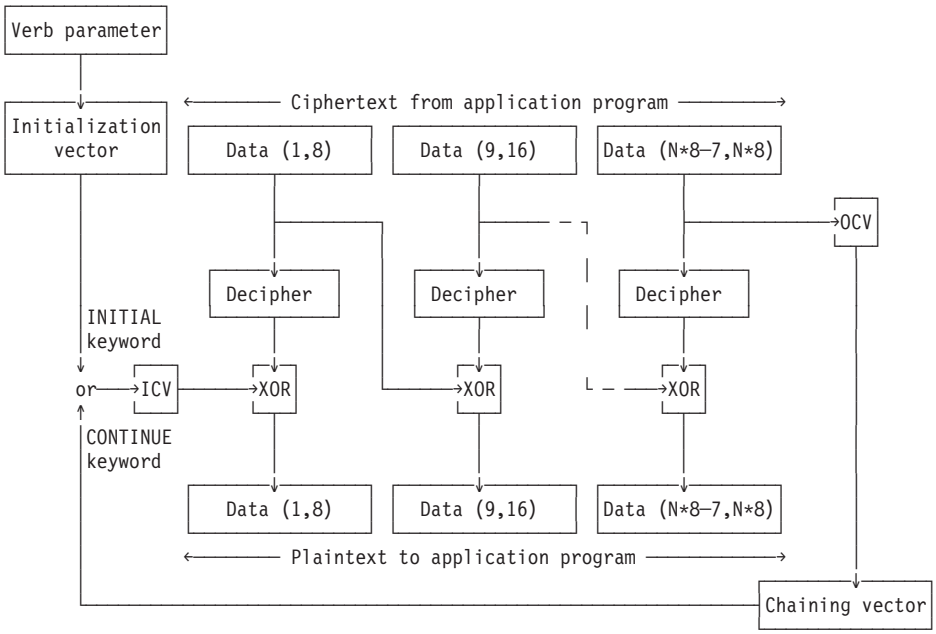


Figure 30. Deciphering using the CBC method

ANSI X9.23 cipher block chaining

ANSI X9.23 defines an enhancement to the basic cipher block chaining (CBC) mode of ANSI X3.106 so that the system can process data with a length that is not an exact multiple of eight bytes.

The ANSI X9.23 method always appends from 1 - 8 bytes to the plaintext before encipherment. The last appended byte is the count of the added bytes and is in the range of X'01' - X'08'. The standard defines that any other added bytes, or pad characters, be random.

When the coprocessor enciphers the plaintext, the resulting ciphertext is always 1 - 8 bytes longer than the plaintext. See Figure 31 on page 936. This is true even if the length of the plaintext is a multiple of eight bytes. When the coprocessor deciphers the ciphertext, it uses the last byte of the deciphered data as the number of bytes to remove from the end (pad bytes, if any, and count byte). The result is the original plaintext. See Figure 32 on page 936.

The output chaining vector can be used as feedback with this method in the same way as with the X3.106 method.

The ANSI X9.23 method requires the caller to supply an initialization vector, and it does not allow specification of a pad character.

Note: The ANSI X9.23 standard has been withdrawn, but the X9.23 padding method is retained in CCA for compatibility with applications that rely on this method.

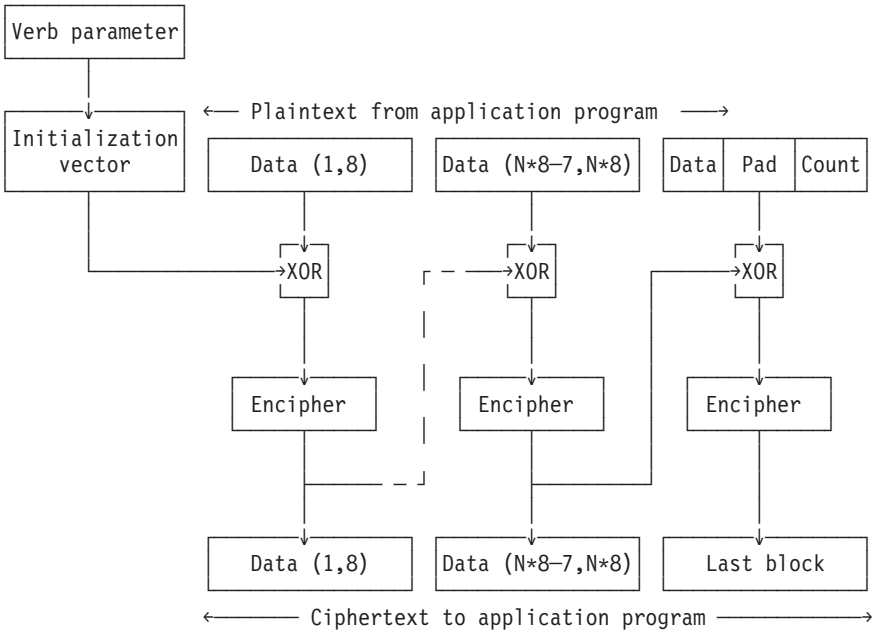


Figure 31. Enciphering using the ANSI X9.23 method

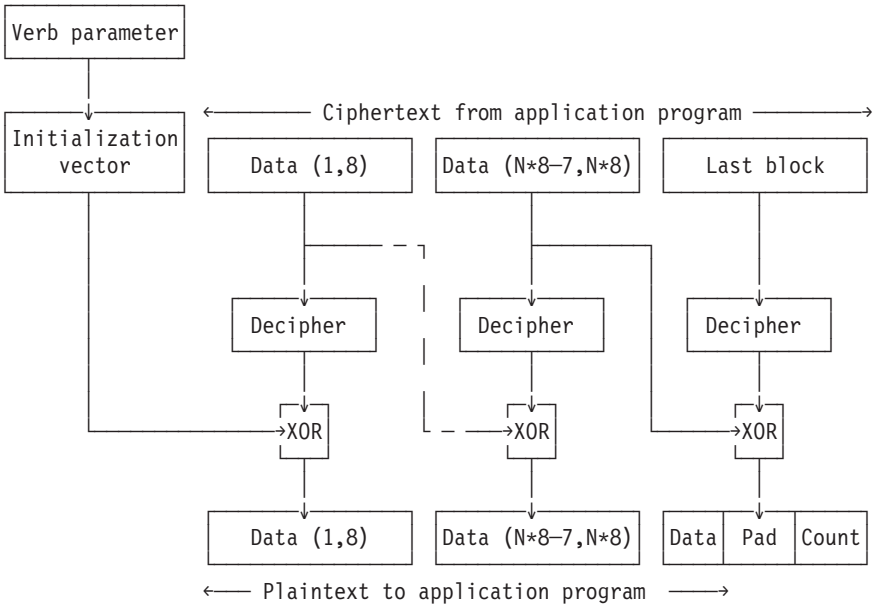


Figure 32. Deciphering using the ANSI X9.23 method

Triple-DES ciphering algorithms

A triple-DES (TDES) algorithm is used to encrypt keys, PIN blocks, and general data.

Several techniques are employed:

TDES ECB
DES keys, when triple encrypted under a double-length DES key, are ciphered using an e-d-e scheme without feedback.

TDES CBC

Encryption of general data, and RSA section type X'08' CRT-format private keys and OPK keys, employs the scheme depicted in Figure 33 and Figure 34 on page 938. This is often referred to as “outer CBC mode.”

This CCA supports double-length DES keys for triple-DES data encryption using the Encipher and Decipher verbs. The triple-length asymmetric master key is used to CBC encrypt CRT-format OPK keys.

EDE*x* / DEDx

CCA employs EDE*x* processes for encrypting several of the RSA private key formats (section types X'02', X'05', and X'06') and the OPK key in section type X'06'. The EDE*x* processes make successive use of single-key DES CBC processes. EDE2, EDE3, and EDE5 processes have been defined, based on the number of keys and initialization vectors used in the process. See Figure 35 on page 938 and Figure 36 on page 939. K1, K2, and K3 are true keys while “K4” and “K5” are initialization vectors. See Figure 35 on page 938 and Figure 36 on page 939.

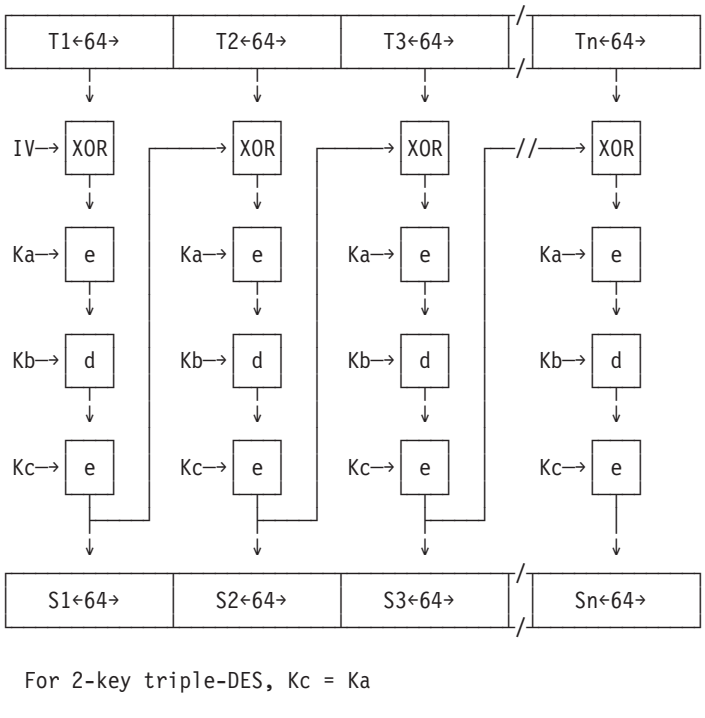


Figure 33. Triple-DES CBC encryption process

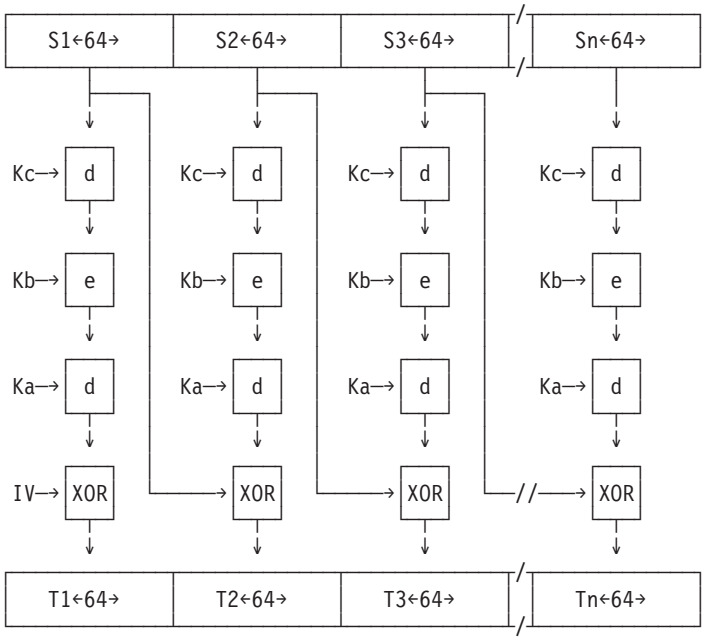


Figure 34. Triple-DES CBC decryption process

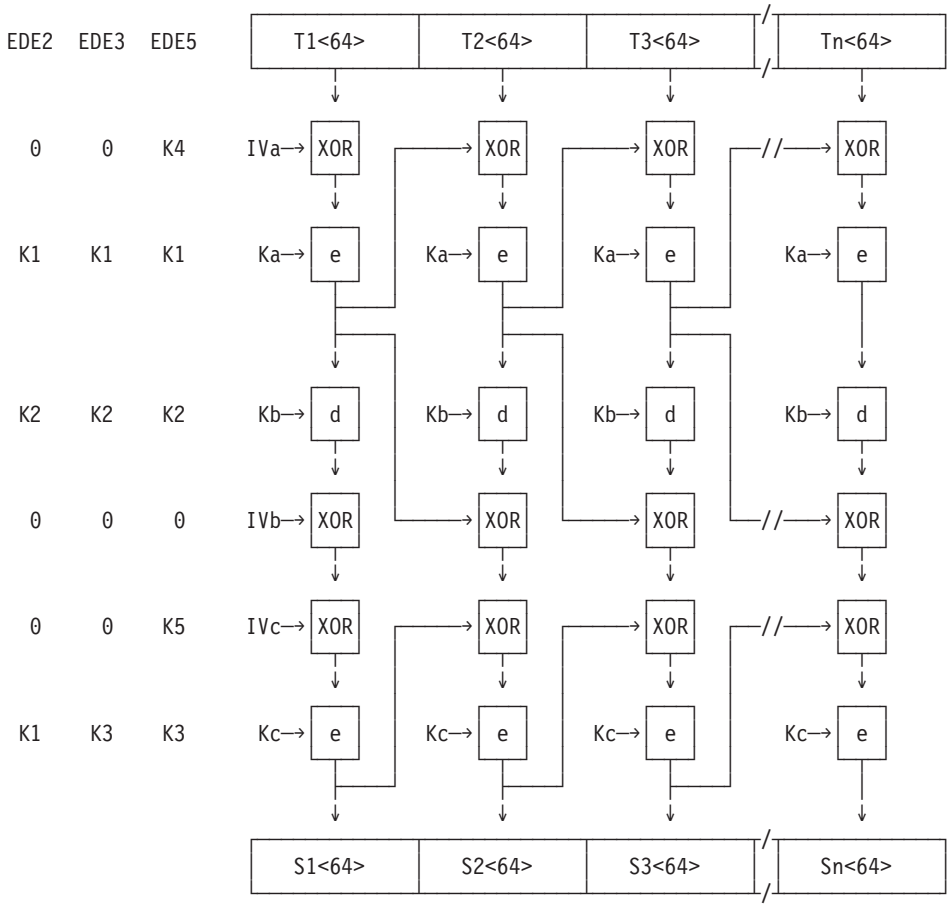


Figure 35. EDE algorithm

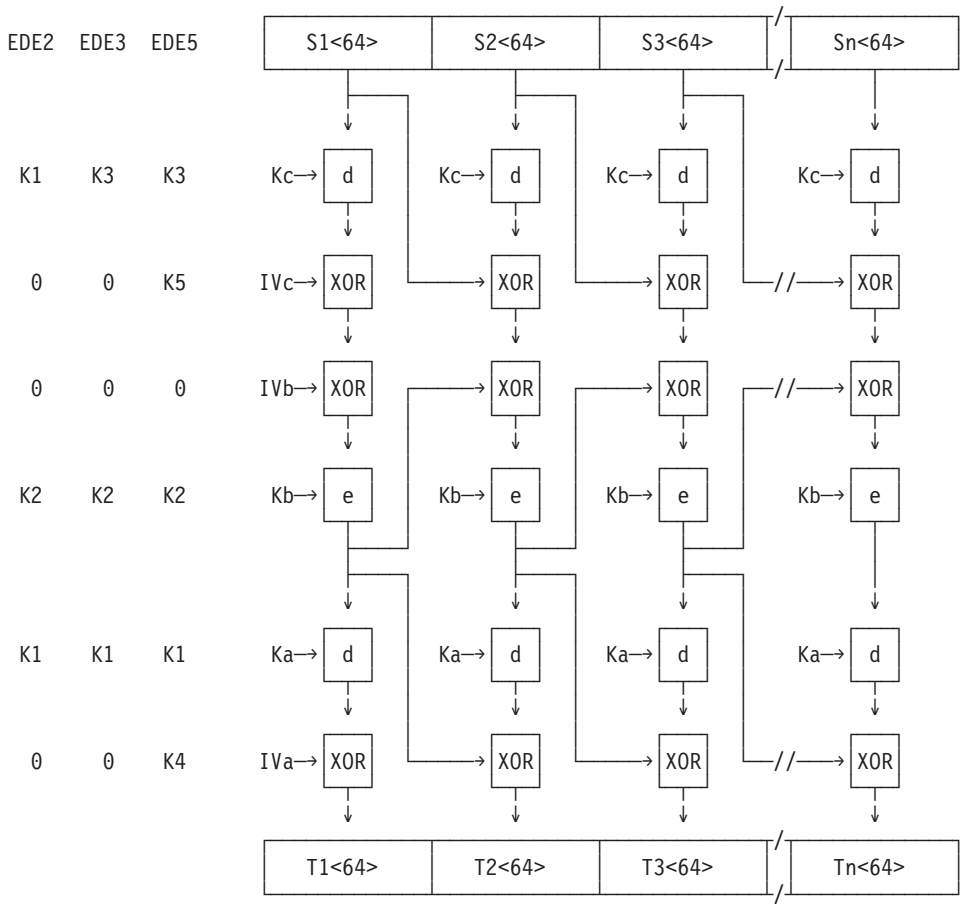


Figure 36. DED process

MAC calculation methods

Four variations of DES-based message authentication can be used by the MAC Generate and MAC Verify verbs.

These variations are:

- "ANSI X9.9 MAC"
- "ANSI X9.19 Optional Procedure 1 MAC" on page 940
- "EMV MAC" on page 940
- "ISO 16609 TDES MAC" on page 940

A keyed-hash MAC (HMAC) based message authentication can be used by the HMAC Generate and HMAC Verify verbs.

ANSI X9.9 MAC

The Financial Institution (Wholesale) Message Authentication Standard (ANSI X9.9-1986) defines a process for the authentication of messages from originator to recipient, and this process is called the Message Authentication Code (MAC) calculation method.

The ANSI X9.9 standard defines five options. The MAC Generate and MAC Verify verbs implement option 1, binary data.

Figure 37 shows the MAC calculation for binary data. In this figure, KEY is a 64-bit key, and T_1 - T_n are 64-bit data blocks of text. If T_n is less than 64 bits long, binary zeros are appended to the right of T_n. Data blocks T_1...T_n are DES CBC-encrypted with all output discarded except for the final output block, O_n.

ANSI X9.19 Optional Procedure 1 MAC

The Financial Institution (Retail) Message Authentication Standard, ANSI X9.19 Optional Procedure 1 (ISO/IEC 9797-1, Algorithm 3) specifies additional processing of the 64-bit O_n MAC value.

The CCA "X9.19OPT" process employs a double-length DES key. After calculating the 64-bit MAC as above with the left half of the double-length key, the result is decrypted using the right half of the double-length key. This result is then encrypted with the left half of the double-length key. The resulting MAC value is processed according to other specifications supplied to the verb call.

EMV MAC

The EMV smart card standards define MAC generation and verification processes that are the same as ANSI X9.9 and ANSI X9.19 Optional Procedure 1 (ISO/IEC 9797-1, Algorithm 3), except for padding added to the end of the message.

Append one byte of X'80' to the original message. Then append additional bytes, as required, of X'00' to form an extended message, which is a multiple of eight bytes in length.

In the ANSI X9.9 and ANSI X9.19 Optional Procedure 1 standards, the leftmost 32 bits (4 bytes) of O_n are taken as the MAC. In the EMV standards, the MAC value is between four and eight bytes in length. CCA provides support for the leftmost four, six, and eight bytes of MAC value.

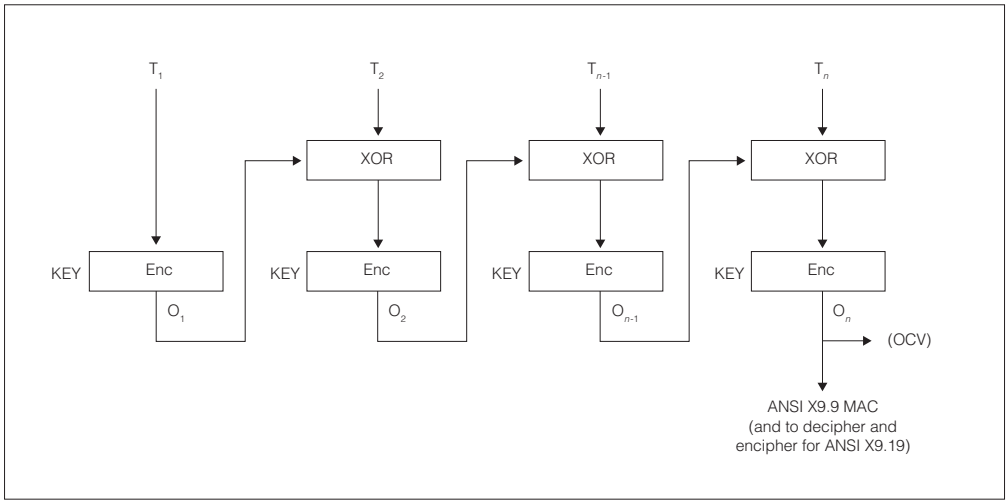


Figure 37. MAC calculation method

ISO 16609 TDES MAC

ISO 16609 defines a process for protecting the integrity of transmitted banking messages and for verifying that a message has originated from an authorized source and this process is called the ISO 16609 TDES MAC method.

The ISO 16609 TDES MAC method corresponds to ISO/IEC 9797-1, algorithm 1 using T-DEA (ANSI X9.52:1998). ISO/FDIS 16609 identifies this method as one of the recommended ways to generate a MAC using symmetric techniques.

The ISO 16609 TDES MAC method uses a double-length DES key and operates on data blocks that are a multiple of eight bytes. If the last input data block is not a multiple of eight bytes, binary zeros are appended to the right of the block. A CBC mode triple-DES (TDES) encryption operation is performed on the data, with all output discarded except for the final output block.

The resulting MAC value is processed according to other specifications supplied to the verb call.

Keyed-hash MAC (HMAC)

The Keyed-Hash Message Authentication Code (HMAC) standard (FIPS PUB 198-1) describes a mechanism for message authentication using cryptographic hash functions.

HMAC can be used with a hash function in combination with a shared secret key.

To see how to compute a MAC over the data, see FIPS PUB 198-1 available at: http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf.

RSA key-pair generation

RSA key-pair generation is determined based on user input of the modulus bit length, public exponent, and key type.

The output is based on creating primes p and q in conformance with ANSI X9.31 requirements as follows:

- prime p bit length = ((modulus_bit_length +1)/2)
- prime q bit length = modulus_bit_length - p_bit_length
- p and q are randomly chosen prime numbers
- $p > q$
- The Rabin-Miller Probabilistic Primality Test is iterated 8 times for each prime. This test determines that a false prime is produced with probability no greater then $1/4^c$, where c is the number of iterations. Refer to the ANSI X9.31 standard and see the section entitled "Miller-Rabin Probabilistic Primality Test."
- Primes p and q are relatively prime with the public exponent.
- Primes p and q are different in at least one of the first 100 most significant bits, that is, $|p-q| > 2^{(\text{prime bit length - 100})}$. For example, when the modulus bit length is 1024, then both primes bit length are 512 bits and the difference of the two primes is $|p-q| > 2^{412}$.

1. For each key generation, and for any size of key, the PKA manager seeds an internal FIPS-approved, SHA-1 based psuedo random number generator (PRNG) with the first 24 bytes of information that it receives from three successive calls to the random number generator (RNG) manager's PRNG interface.
2. The RNG manager can supply random number in two ways, but with the CCA Support Program only one way is used, namely, the PRNG method. The PKA manager seeds an internal FIPS-approved, SHA-1 based PRNG with 24 bytes obtained.

The RNG manager can respond to requests for random numbers from other processes with such responses interspersed between responses to PKA manager requests. An RSA key is generated from random information obtained from two cascaded SHA-1 PRNGs.

3. An RSA key is based on one or more 24-byte seeds from the RNG manager source, depending on the dynamic mix of tasks running inside the coprocessor.

There exists a system RNG manager (ANSI X9.31 compliant) that is used as the source for pseudo random numbers. The PKA manager also has a PRNG that is DSA compliant for generating primes. The PKA manager PRNG is re-seeded from the system RNG manager, for every new key pair generation, which is for every generation of a public/private key pair.

Multiple decipherment and encipherment

CCA uses multiple encipherment and decipherment to protect and retrieve cryptographic keys and PIN data.

CCA uses multiple encipherment whenever it enciphers a key under a key-encrypting key such as the master key or the transport key and in triple-DES encipherment for data privacy. Multiple encipherment is superior to single encipherment because multiple encipherment increases the work needed to "break" a key. CCA provides extra protection for a key by enciphering it under an enciphering key multiple times rather than once. The multiple encipherment method for keys enciphered under a key-encrypting key uses a double-length (128-bit) key split into two 64-bit halves. Like single encipherment, multiple encipherment uses a DES based on the electronic code book (ECB) mode of encipherment.

Keys can either be double-length or single-length depending on the installation and their cryptographic function. When a single-length key is encrypted under a double-length key, multiple encipherment is performed on the key. In the multiple encipherment method, the key is encrypted under the left half of the enciphering key. The result is then decrypted under the right half of the enciphering key. Finally, this result is encrypted under the left half of the enciphering key again.

When a double-length key is encrypted with multiple encipherment, the method is similar, except CCA uses two enciphering keys. One enciphering key encrypts each half of the double-length key. Double-length keys active on the system have two master key variants used when enciphering them.

Multiple encipherment and decipherment is not only used to protect or retrieve a cryptographic key, but they are also used to protect or retrieve 64-bit data in the area of PIN applications. For example, the following two sections use a double-length *KEK as an example to cipher a single-length key even though the same algorithms apply to cipher 64-bit data by a double-length PIN-related cryptographic key.

CCA also supports triple-DES encipherment for data privacy using double-length and triple-length DATA keys. For this procedure the data is first enciphered using the first DATA key. The result is then deciphered using the second DATA key. This second result is then enciphered using the third DATA key when a triple-length key is provided or reusing the first DATA key when a double-length key is provided.

Note that an asterisk (*) preceding the key means the key is double-length. Notations in this chapter have the following meaning:

- eK(x), where x is enciphered under K
- dK(y) represents plaintext, where K is the key and y is the ciphertext

Therefore, dK(eK(x)) equals x for any 64-bit key K and any 64-bit plaintext x.

When a key (*K) to be protected is double-length, two double-length *KEKs are used. One *KEK is used for protecting the left half of the key (*K); another is for the right half. Multiple encipherment is used with the appropriate *KEK for protecting each half of the key.

Multiple encipherment of single-length keys

Definition of the multiple encipherment of a single-length key (K) using a double-length *KEK.

The multiple encipherment of a single-length key (K) using a double-length *KEK is defined as follows:

```
e*KEK(K) = eKEKL(dKEKR(eKEKL(K)))
```

where KEKL is the left 64 bits of *KEK and KEKR is the right 64 bits of *KEK.

Figure 38 illustrates the definition.

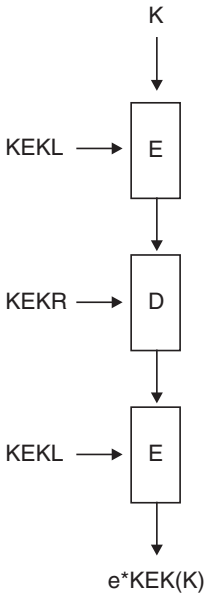


Figure 38. Multiple encipherment of single-length keys

Multiple decipherment of single-length keys

Definition of the multiple encipherment of an encrypted single-length key (Y = e*KEK(K)) using a double-length *KEK.

The multiple encipherment of an encrypted single-length key (Y = e*KEK(K)) using a double-length *KEK is defined as follows:

```
d*KEK(Y) = dKEKL(eKEKR(dKEKL(Y)))
         = d*KEK(e*KEK(K))
         = K
```

where KEKL is the left 64 bits of *KEK and KEKR is the right 64 bits of *KEK.

Figure 39 illustrates the definition.

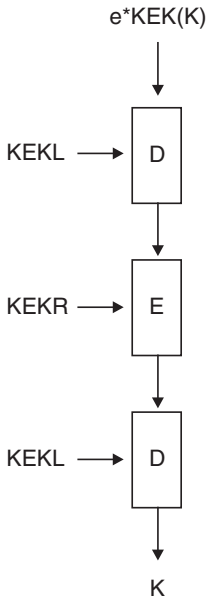


Figure 39. Multiple decipherment of single-length keys

Multiple encipherment of double-length keys

Definition of the multiple encipherment of a double-length key (*K) using two double-length *KEKs, *KEKa, and *KEKb.

The multiple encipherment of a double-length key (*K) using two double-length *KEKs, *KEKa, and *KEKb is defined as follows:

```
e*KEKa(KL) || e*KEKb(KR) =
              eKEKaL(dKEKaR(eKEKaL(KL))) ||
              eKEKbL(dKEKbR(eKEKbL(KR)))
```

where:

- KL is the left 64 bits of *K
- KR is the right 64 bits of *K
- KEKaL is the left 64 bits of *KEKa
- KEKaR is the right 64 bits of *KEKa
- KEKbL is the left 64 bits of *KEKb
- KEKbR is the right 64 bits of *KEKb
- `||` means concatenation

Figure 40 on page 945 illustrates the definition.

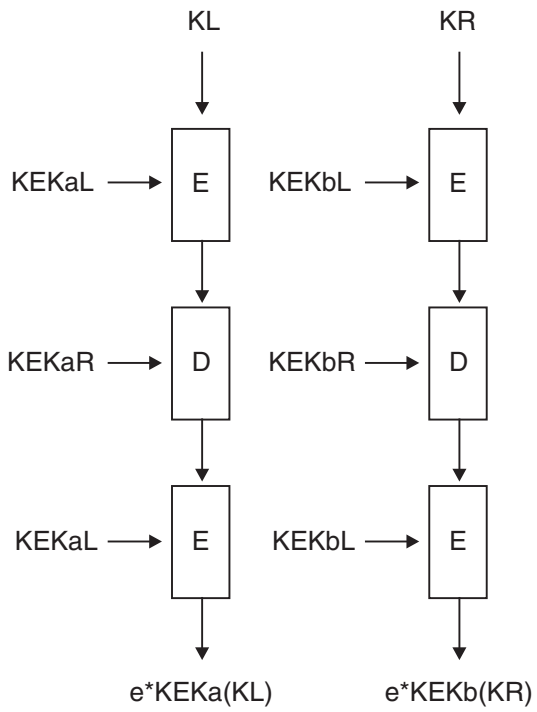


Figure 40. Multiple encipherment of double-length keys

Multiple decipherment of double-length keys

Definition of the multiple decipherment of an encrypted double-length key, *Y = e*KEKa(KL) || e*KEKb(KR), using two double-length *KEKs, *KEKa, and *KEKb.

The multiple decipherment of an encrypted double-length key, *Y = e*KEKa(KL) || e*KEKb(KR), using two double-length *KEKs, *KEKa, and *KEKb, is defined as follows:

```
D*KEKa(YL) || d*KEKb(YR)
        = dKEKaL(eKEKaR(dKEKaL(YL))) ||
          dKEKbL(eKEKbR(dKEKbL(YR)))
        = d*KEKa(e*KEKa(KL)) ||
          d*KEKb(e*KEKb(KR))
        = *K
```

where

- YL is the left 64 bits of *Y
- YR is the right 64 bits of *Y
- KEKaL is the left 64 bits of *KEKa
- KEKaR is the right 64 bits of *KEKa
- KEKbL is the left 64 bits of *KEKb
- KEKbR is the right 64 bits of *KEKb
- || means concatenation

Figure 41 on page 946 illustrates the definition.

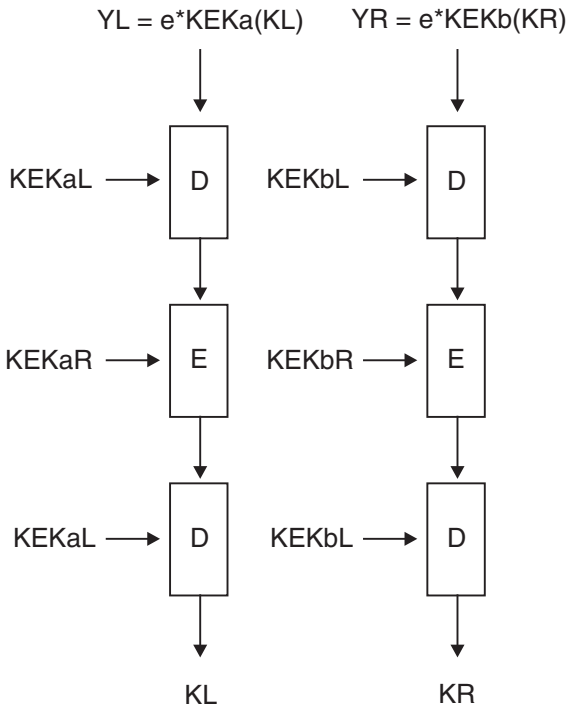


Figure 41. Multiple decipherment of double-length keys

Multiple encipherment of triple-length keys

Definition of the multiple encipherment of a triple-length key (**K) using two double-length *KEKs, *KEKa, and *KEKb.

The multiple encipherment of a triple-length key (**K) using two double-length *KEKs, *KEKa, and *KEKb is defined as follows:

```
e*KEKa(KL) || e*KEKb(KM) || e*KEKa(KR) =
    eKEKaL(dKEKaR(eKEKaL(KL))) ||
    eKEKbL(dKEKbR(eKEKbL(KM))) ||
    eKEKaL(dKEKaR(eKEKaL(KR)))
```

where:

- KL is the left 64 bits of **K
- KM is the next 64 bits of **K
- KR is the right 64 bits of **K
- KEKaL is the left 64 bits of *KEKa
- KEKaR is the right 64 bits of *KEKa
- KEKbL is the left 64 bits of *KEKb
- KEKbR is the right 64 bits of *KEKb
- || means concatenation

Figure 42 on page 947 illustrates the definition.

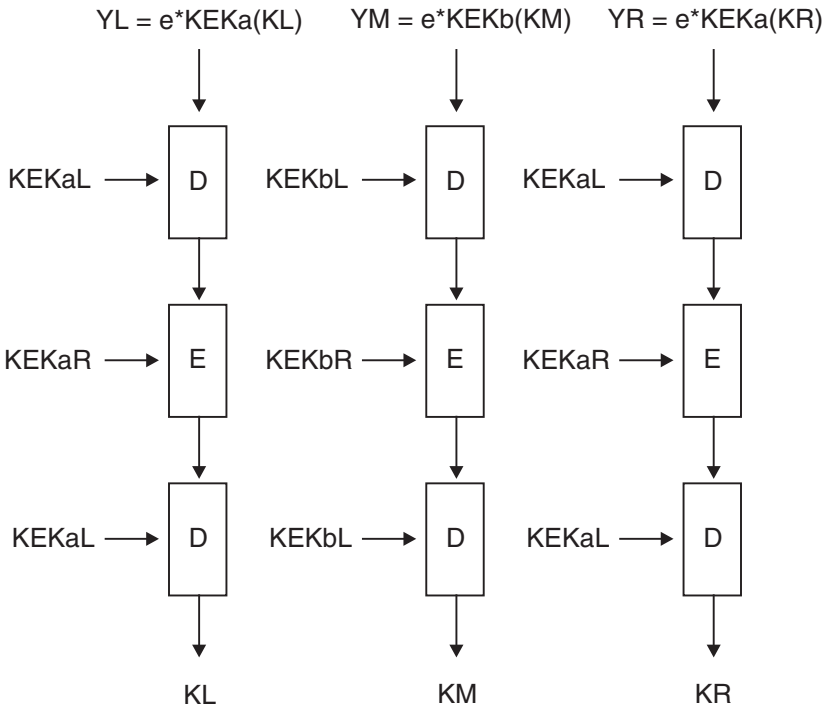


Figure 42. Multiple encipherment of triple-length keys

Multiple decipherment of triple-length keys

Definition of the multiple decipherment of an encrypted triple-length key **Y = e*KEKa(KL) || e*KEKb(KM) || e*KEKa(KR), using two double-length *KEKs, *KEKa, and *KEKb.

The multiple decipherment of an encrypted triple-length key **Y = e*KEKa(KL) || e*KEKb(KM) || e*KEKa(KR), using two double-length *KEKs, *KEKa, and *KEKb, is defined as follows:

```
d*KEKa(YL) || d*KEKb(YM) || d*KEKa(YR)
   = dKEKaL(eKEKaR(dKEKaL(YL))) ||
     dKEKbL(eKEKbR(dKEKbL(YM))) ||
     dKEKaL(eKEKaR(dKEKaL(YR)))
   = d*KEKa(e*KEKa(KL)) ||
     d*KEKb(e*KEKb(KM)) ||
     d*KEKa(e*KEKa(KR))
   = **K
```

where:

- YL is the left 64 bits of **Y
- YM is the next 64 bits of **Y
- YR is the right 64 bits of **Y
- KEKaL is the left 64 bits of *KEKa
- KEKaR is the right 64 bits of *KEKa
- KEKbL is the left 64 bits of *KEKb
- KEKbR is the right 64 bits of *KEKb
- || means concatenation

Figure 43 illustrates the definition.

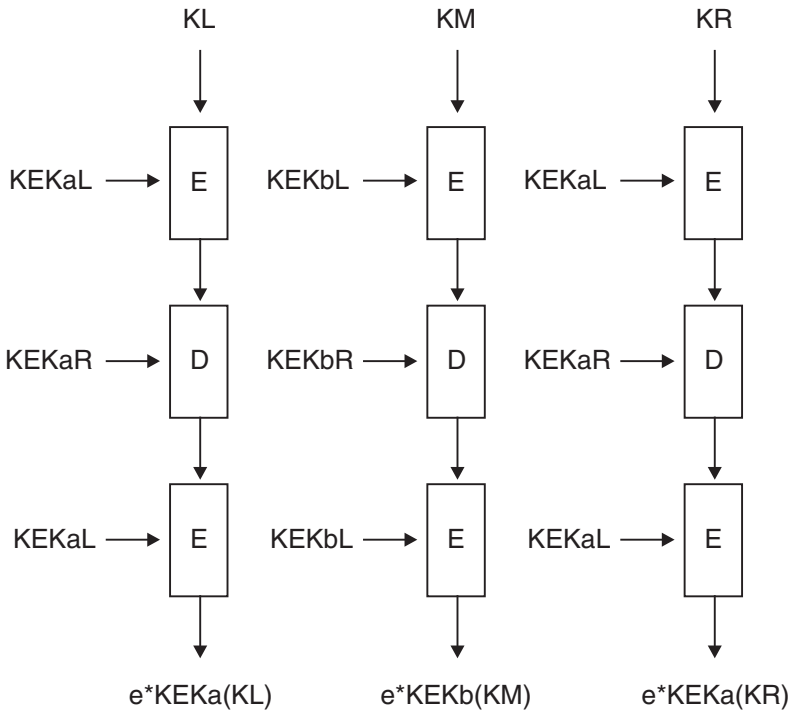


Figure 43. Multiple decipherment of triple-length keys

PKA92 key format and encryption process

The Symmetric Key Generate and the Symmetric Key Import verbs optionally support a **PKA92** method of encrypting a DES key with an RSA public key.

This format is adapted from the IBM Transaction Security System (TSS) 4753 and 4755 product's implementation of "PKA92". The verbs do not create or accept the complete PKA92 AS key token as defined for the TSS products. Rather, the verbs support only the actual RSA-encrypted portion of a TSS PKA92 key token, the *AS External Key Block*.

Forming an external key block - The PKA96 implementation forms an AS External Key Block by RSA-encrypting a key block using a public key. The key block is formed by padding the key record detailed in Table 269 with zero bits on the left, high-order end of the key record. The process completes the key block with three sub-processes: masking, overwriting, and RSA encrypting.

Table 269. PKA96 clear DES key record

Offset (bytes)	Length (bytes)	Description
Zero-bit padding to form a structure as long as the length of the public key modulus. The implementation constrains the public key modulus to a multiple of 64 bits in the range of 512 - 1024 bits. Note that government export or import regulations can impose limits on the modulus length. The maximum length is validated by a check against a value in the Function Control Vector.		
000	005	Header and flags: X'01 0000 0000.'
005	016	Environment Identifier (EID), encoded in ASCII.

Table 269. PKA96 clear DES key record (continued)

Offset (bytes)	Length (bytes)	Description
021	008	Control vector base for the DES key.
029	008	Repeat of the CV data at offset 021.
037	008	The single-length DES key or the left half of a double-length DES key.
045	008	The right half of a double-length DES key or a random number. This value is locally designated "K."
053	008	Random number, "IV."
061	001	Ending byte, X'00.'

Masking Sub-process - Create a mask by CBC encrypting a multiple of eight bytes of binary zeros using K as the key and IV as the initialization vector as defined in the key record at offsets 45 and 53. XOR the mask with the key record and call the result PKR.

Overwriting Sub-process - Set the high-order bits of PKR to B'01' and set the low-order bits to B'0110'.

XOR K and IV and write the result at offset 45 in PKR.

Write IV at offset 53 in PKR. This causes the masked and overwritten PKR to have IV at its original position.

Encrypting Sub-process - RSA encrypt the overwritten PKR masked key record using the public key of the receiving node.

Recovering a key from an external key block - Recover the encrypted DES key from an AS External Key Block by performing decrypting, validating, unmasking, and extraction sub-processes.

Decrypting Sub-process - RSA decrypt the AS External Key Block using an RSA private key and call the result of the decryption PKR. The private key must be usable for key management purposes.

Validating Sub-process - Verify the high-order two bits of the PKR record are valued to B'01' and the low-order four bits of the PKR record are valued to B'0110'.

Unmasking Sub-process - Set IV to the value of the eight bytes at offset 53 of the PKR record. Note that there is a variable quantity of padding prior to offset 0. See Table 269 on page 948.

Set K to the XOR of IV and the value of the eight bytes at offset 45 of the PKR record.

Create a mask equal in length to the PKR record by CBC encrypting a multiple of eight bytes of binary zeros using K as the key and IV as the initialization vector. XOR the mask with PKR and call the result the key record.

Copy K to offset 45 in the PKR record.

Extraction Sub-process. Confirm that:

- The four bytes at offset 1 in the key record are valued to X'0000 0000' .
- The two control vector fields at offsets 21 and 29 are identical.
- If the control vector is an IMPORTER or EXPORTER key class, the Environment Identifier (EID) in the key record is not the same as the EID stored in the cryptographic engine.

The control vector base of the recovered key is the value at offset 21. If the control vector base bits 40 - 42 are valued to B'010' or B'110', the key is double length. Set the right half of the received key's control vector equal to the left half and reverse bits 41 and 42 in the right half.

The recovered key is at offset 37 and is either 8 or 16 bytes long based on the control vector base bits 40 - 42. If these bits are valued to B'000', the key is single length. If these bits are valued to B'010' or B'110', the key is double length.

Formatting hashes and keys in public-key cryptography

The Digital Signature Generate and Digital Signature Verify verbs support several methods for formatting a hash and, in some cases, a descriptor for the hashing method, into a bit-string to be processed by the cryptographic algorithm.

This section provides information about the ANSI X9.31 and PKCS #1 methods. The ISO 9796-1 method can be found in the ISO standard.

This section also describes the PKCS #1, version 1, 1.5, and 2.0, methods for placing a key in a bit string for RSA ciphering as part of a key exchange.

ANSI X9.31 hash format

With ANSI X9.31, the string that is processed by the RSA algorithm is formatted by the concatenation of a header, padding, the hash value and a trailer, from the most significant bit to the least significant bit, so that the resulting string is the same length as the modulus of the key.

For CCA, the modulus length must be a multiple of 8 bits.

- The header consists of the value X'6B'.
- The padding consists of the value X'BB', repeated as many times as required, and ended with X'BA'.
- The hash value follows the padding.
- The trailer consists of a hashing mechanism specifier and final byte. The hashing mechanism specifier is defined as one of the following values:

  X'31' RIPEMD-160

  X'32' RIPEMD-128

  X'33' SHA-1

  X'34' SHA-256 (Release 3.30.05 or later)
- The final byte is X'CC'.

PKCS #1 hash formats

Version 2.0 of the PKCS #1 standard defines methods for formatting keys and hashes prior to RSA encryption of the resulting data structures.

The earlier versions of the PKCS #1 standard defined block types 0, 1, and 2, but in the current standard that terminology is dropped.

CCA implemented these processes using the terminology of the Version 2.0 standard:

- For formatting keys for secured transport Symmetric Key Export, Digital Signature Generate, Symmetric Key Import):
  - RSAES-OAEP, the preferred method for key-encipherment [3] when exchanging DATA keys between systems. Keyword **PKCSOAEP** (Version 2.0) and **PKOAEP2** (Version 2.1) is used to invoke this formatting technique. The P parameter described in the standard is not used and its length is set to zero.
  - RSAES-PKCS1-v1_5, is an older method for formatting keys. It is included for compatibility with existing applications. Keyword **PKCS-1.2** is used to invoke this formatting technique.
- For formatting hashes for digital signatures (Digital Signature Generate and Digital Signature Verify):
  - RSASSA-PKCS1-v1_5, the newer name for the block-type 1 format. Keyword **PKCS-1.1** is used to invoke this formatting technique.
  - The PKCS #1 specification no longer describes the use of block-type 0. Keyword **PKCS-1.0** is used to invoke this formatting technique. Use of block-type 0 is discouraged.

Using the terminology from older versions of the PKCS #1 standard, block types 0 and 1 are used to format a hash and block type 2 is used to format a DES key. The blocks consist of the following ("||" means concatenation):

- X'00' || BT || PS || X'00' || D
-
- where:

  BT Is the block type, X'00', X'01', or X'02'.

  PS Is the padding of as many bytes as required to make the block the same length as the modulus of the RSA key. Padding of X'00' is used for block type 0, X'FF' for block type 1, and random and non-X'00' for block type 2. The length of PS must be a minimum of eight bytes.

  D Is the key, or the concatenation of the BER-encoded hash identifier and the hash value.

You can create the ASN.1 BER encoding of an MD5, SHA-1, or SHA-256 value by prepending these strings to the 16-byte or 20-byte hash values, respectively:

MD5 `X'3020300C 06082A86 4886F70D 02050500 0410'`

SHA-1
`X'30213009 06052B0E 03021A05 000414'`

SHA-256
`X'3031300D 06096086 48016503 04020105 000420'`

2. PKCS standards can be retrieved from http://www.rsasecurity.com/rsalabs/pkcs.

3. The PKA 92 method and the method incorporated into the SET standard are other examples of the Optimal Asymmetric Encryption Padding (OAEP) technique. The OAEP technique is attributed to Bellare and Rogaway.

Chapter 22. Access control points and verbs

Verbs use Access Control Points (ACPs). ACPs are also referred to as commands.

Important: By default, you should disable commands. Do not enable an ACP unless you know why you are enabling it.

For instructions on how to enable and disable these ACPs using the TKE workstation, see *z/OS Cryptographic Services ICSF: Trusted Key Entry PCIX Workstation User's Guide*.

For systems that do not use the optional TKE workstation, most ACPs (current and new) are enabled in the DEFAULT role with the appropriate licensed internal code on the CEX*C.

Note that each domain in the CEX*C (with hardware enforced access permissions) starts out with its own DEFAULT role with the default ACP values as shown. However, it is possible to use the TKE to change ACP values in the DEFAULT role or to define other roles. The role to which a user is assigned determines the ACPs available to that user. Full coverage of TKE use for configuration is outside the scope of this document. For details, see *z/OS Cryptographic Services ICSF: Trusted Key Entry PCIX Workstation User's Guide*.

Table 270 on page 954 lists the CCA ACPs. The name of each ACP is given as it appears on the panels of the TKE user interface. Note that the group names are also given to aid locating the ACPs. The table includes the following columns:

ACP number
The hexadecimal offset, or ACP code, for the command. Offsets between X'0000' and X'FFFF' that are not listed in this table are reserved.

Name of ACP from TKE interface
The name of the ACP as it appears on the TKE interface

Verb name
The names of the verbs that require that ACP to be enabled; for example, the Encipher (CSNBENC) verb fails without permission to use the **Encipher - DES** ACP.

Entry point
The entry-point name of the verb.

Initial setting
Whether the ACP is ON or OFF by default.

Usage Usage recommendations for the ACP. The abbreviations in this column are explained at the end of the table.

See the **Restrictions**, **Required commands**, or **Usage notes** sections at the end of each verb description for access control information.

Table 270. Access Control Points and corresponding CCA verbs

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
0001 GROUP: ISPF Services **Note:** This group name refers to ISPF, a z/OS feature. Although ISPF is not relevant to Linux on z Systems, it is listed here as shown on the TKE panels to avoid confusion.					
X'0018'	**DES Master Key - Load first key part**	Master Key Process[1]	CSNBMKP	ON	SC, SEL
X'0019'	**DES Master Key - Combine key parts**	Master Key Process[1]	CSNBMKP	ON	SC, SEL
X'001A'	**DES Master Key - Set master key**	Master Key Process[1]	CSNBMKP	ON	SC, SEL
X'001E'	**Reencipher CKDS** **Note:** The TKE name for this ACP refers to z/OS key storage (CKDS). However z/OS key storage is not impacted. This ACP refers to a service for Linux, see verb for details.	Key Token Change	CSNBKTC	ON	O
X'0032'	**DES Master Key - Clear new master key register**	Master Key Process[1]	CSNBMKP	ON	O, SUP
X'0053'	**RSA Master Key - Load first key part**	Master Key Process[1]	CSNBMKP	ON	SC, SEL
X'0054'	**RSA Master Key - Combine key parts**	Master Key Process[1]	CSNBMKP	ON	SC, SEL
X'0057'	**RSA Master Key - Set master key**	Master Key Process[1]	CSNBMKP	ON	SC, SEL
X'0060'	**RSA Master Key - Clear new master key register**	Master Key Process[1]	CSNBMKP	ON	SC, SEL
X'00F0'	**Reencipher CKDS2**				
X'0124'	**AES Master Key - Clear new master key register**	Master Key Process[1]	CSNBMKP	ON	O, SUP
X'0125'	**AES Master Key - Load first key part**	Master Key Process[1]	CSNBMKP	ON	O, SUP
X'0126'	**AES Master Key - Combine key parts**	Master Key Process[1]	CSNBMKP	ON	O, SUP
X'0128'	**AES Master Key - Set master key**	Master Key Process[1]	CSNBMKP	ON	O, SUP
X'0146'	**CKDS Conversion2 - Allow wrapping override keywords**	Key Token Change	CSNBKTC	ON	O
X'0147'	**CKDS Conversion2 - Convert from enhanced to original**	Key Token Change Key Translate2	CSNBKTC CSNBKTR2	ON	O, NRP
X'014C'	**CKDS Conversion2 - Allow use of REFORMAT**	Key Token Change	CSNBKTC	ON	O
X'0240'	**Authorize UDX**	- no verb -	- no verb -	ON	O
X'0241'	**Reencipher PKDS**	PKA Key Token Change	CSNDKTC	ON	O, R
X'0303'	**PCF CKDS conversion utility**	PCF/CUSP Key Conversion - indirect verb	- indirect verb/indirect usage -	ON	R

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'031F'	**ECC Master Key - Clear new master key register**	Master Key Process	CSNBMKP	ON	O
X'0320'	**ECC Master Key - Load first key part**	Master Key Process	CSNBMKP	ON	O
X'0321'	**ECC Master Key - Combine key parts**	Master Key Process	CSNBMKP	ON	O
X'0322'	**ECC Master Key - Set master key**	Master Key Process	CSNBMKP	ON	O
X'0330'	**DES master key - 24-byte key**	Master Key Process **Note:** This ACP forces the SYM and ASYM master keys to be full 24 byte DES keys.	CSNBMKP	OFF	O
0002 GROUP: Coprocessor Configuration					
X'0034'	**Log Query: System**	Log Query	CSUALGQ	ON	O
X'0035'	**Log Query: CCA**	Log Query	CSUALGQ	ON	O
X'0036'	**Log Query: Set Log Level -4-**	Log Query	CSUALGQ	ON	O
X'0037'	**Log Query: Set Log Level -8-**	Log Query	CSUALGQ	ON	O
X'0116'	**Access Control Manager - Read role**	**Note:** This ACP is included for reference only. The service impacted is available only for IBM System x, IBM System p®, or using the TKE interface.		ON	O
X'0139'	**Symmetric token wrapping - internal enhanced method**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		ON	O
X'013A'	**Symmetric token wrapping - internal original method**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		ON	O
X'013B'	**Symmetric token wrapping - external enhanced method**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		ON	O
X'013C'	**Symmetric token wrapping - external original method**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		ON	O
X'0328'	**Variable-length Symmetric Token - disallow weak wrap**	EC Diffie-Hellman[1] Key Generate2[1] PKA Key Generate[1] Symmetric Key Export[1]	CSNDEDH CSNBKGN2 CSNDPKG CSNDSYX	OFF	O, R
X'032C'	**Variable-length Symmetric Token - warn when weak wrap**	EC Diffie-Hellman[1] Key Generate2[1] Symmetric Key Export[1] Symmetric Key Import2[1]	CSNDEDH CSNBKGN2 CSNDSYX CSNDSYI2J	OFF	O, R
X'032D'	**Disallow 24-byte DATA wrapped with 16-byte Key**	PKA Key Generate	CSNDPKG	OFF	O

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'0332'	**Warn when weak wrap - Master keys**	Clear Key Import Data Key Import Diversified Key Generate EC Diffie-Hellman Key Generate Key Generate2 Key Import Key Part Import Key Part Import2 Key Token Change Key Token Change2 Master Key Process Multiple Clear Key Import PKA Key Generate PKA Key Import PKA Key Token Change Prohibit Export Restrict Key Attribute Symmetric Key Generate Symmetric Key Import Symmetric Key Import2 TR31 Key Import Unique Key Derive	CSNBCKI CSNBDKM CSNBDKG CSNDEDH CSNBKGN CSNBKGN2 CSNBKIM CSNBKPI CSNBKPI2 CSNBKTC CSNBKTC2 CSNBMKP CSNBCKM CSNDPKG CSNDPKI CSNDKTC CSNBPEX CSNBRKA CSNDSYG CSNDSYI CSNDSYI2 CSNBT31I CSNBUKD	ON	O, R
X'0333'	**Prohibit weak wrapping - Master keys**	Same as ACP X'0332'	Same as ACP X'0332'	ON	O, R
0003 GROUP: API Cryptographic Services					
X'000E'	**Encipher - DES**	Encipher	CSNBENC	ON	O
X'000F'	**Decipher - DES**	Decipher	CSNBDEC	ON	O
X'0010'	**MAC Generate**	MAC Generate	CSNBMGN	ON	O
X'0011'	**MAC Verify**	MAC Verify	CSNBMVR	ON	O
X'0012'	**Key Import**	Key Import	CSNBKIM	ON	O
X'0013'	**Key Export**	Key Export	CSNBKEX	ON	O
X'001B'	**Key Part Import - first key part**	Key Part Import[1]	CSNBKPI	ON	SC, SEL
X'001C'	**Key Part Import - middle and last**	Key Part Import[1]	CSNBKPI	ON	SC, SEL

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'001D'	**Key Test and Key Test2** - continued -	AES Key Record Create	CSNBAKRC	ON	R
		AES Key Record Delete	CSNBAKRD		
		AES Key Record List	CSNBAKRL		
		AES Key Record Read	CSNBAKRR		
		AES Key Record Write	CSNBAKRW		
		Authentication Parameter Generate	CSNBAPG		
		Cipher Text Translate2	CSNBCTT2		
		Clear Key Import	CSNBCKI		
		Clear PIN Encrypt	CSNBCPE		
		Clear PIN Generate	CSNBPGN		
		Clear PIN Generate Alternate	CSNBCPA		
		Control Vector Translate	CSNBCVT		
		Cryptographic Variable Encipher	CSNBCVE		
		CVV Generate	CSNBCSG		
		CVV Key Combine	CSNBCKC		
		CVV Verify	CSNBCSV		
		Data Key Export	CSNBDKX		
		Data Key Import	CSNBDKM		
		Decipher	CSNBDEC		
		DES Key Record Create	CSNBKRC		
		DES Key Record Delete	CSNBKRD		
		DES Key Record List	CSNBKRL		
		DES Key Record Read	CSNBKRR		
		DES Key Record Write	CSNBKRW		
		Digital Signature Generate	CSNDDSG		
		Digital Signature Verify	CSNDDSV		
		Diversified Key Generate	CSNBDKG		
		Diversified Key Generate2	CSNBDKG2		
		DK Deterministic PIN Generate	CSNBDDPG		
		DK Migrate PIN	CSNBDMP		
		DK PAN Modify in Transaction	CSNBDPMT		
		DK PAN Translate	CSNBDPT		
		DK PIN Change	CSNBDPC		
		DK PIN Verify	CSNBDPV		
		DK PRW Card Number Update	CSNBDPNU		
		DK PRW CMAC Generate	CSNBDPCG		
		DK Random PIN Generate	CSNBDRPG		
		DK Regenerate PRW	CSNBDRP		
		EC Diffie-Hellman	CSNDEDH		
		Encipher	CSNBENC		
		Encrypted PIN Generate	CSNBEPG		
		Encrypted PIN Translate	CSNBPTR		
		Encrypted PIN Translate Enhanced	CSNBPTRE		
		Encrypted PIN Verify	CSNBPVR		
		FPE Decipher	CSNBFPED		
		FPE Encipher	CSNBFPEE		
		FPE Translate	CSNBFPET		
		HMAC Generate	CSNBHMG		
		HMAC Verify	CSNBHMV		
		Key Export	CSNBKEX		
		Key Export to TR31	CSNBT31X		
		Key Generate	CSNBKGN		
		Key Generate2	CSNBKGN2		
		Key Import	CSNBKIM		

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'001D'	**Key Test and Key Test2** - continued -	Key Part Import	CSNBKPI	ON	R
		Key Part Import2	CSNBKPI2		
		Key Storage Initialization	CSNBKSI		
		Key Test	CSNBKYT		
		Key Test2	CSNBKYT2		
		Key Test Extended	CSNBKYTX		
		Key Token Change	CSNBKTC		
		Key Token Change2	CSNBKTC2		
		Key Translate	CSNBKTR		
		Key Translate2	CSNBKTR2		
		MAC Generate	CSNBMGN		
		MAC Generate2	CSNBMGN2		
		MAC Verify	CSNBMVR		
		MAC Verify2	CSNBMVR2		
		Multiple Clear Key Import	CSNBCKM		
		PIN Change/Unblock	CSNBPCU		
		PKA Decrypt	CSNDPKD		
		PKA Encrypt	CSNDPKE		
		PKA Key Generate	CSNDPKG		
		PKA Key Import	CSNDPKI		
		PKA Key Record Create	CSNDKRC		
		PKA Key Record Delete	CSNDKRD		
		PKA Key Record List	CSNDKRL		
		PKA Key Record Read	CSNDKRR		
		PKA Key Record Write	CSNDKRW		
		PKA Key Token Change	CSNDKTC		
		PKA Key Translate	CSNDPKT		
		PKA Public Key Extract	CSNDPKX		
		Prohibit Export	CSNBPEX		
		Prohibit Export Extended	CSNBPEXX		
		Recover PIN from Offset	CSNBPFO		
		Remote Key Export	CSNDRKX		
		Restrict Key Attribute	CSNBRKA		
		Secure Messaging for Keys	CSNBSKY		
		Secure Messaging for PINs	CSNBSPN		
		Symmetric Algorithm Decipher	CSNBSAD		
		Symmetric Algorithm Encipher	CSNBSAE		
		Symmetric Key Export	CSNDSYX		
		Symmetric Key Export with Data	CSNDSXD		
		Symmetric Key Generate	CSNDSYG		
		Symmetric Key Import	CSNDSYI		
		Symmetric Key Import2	CSNDSYI2		
		TR31 Key Import	CSNBT31I		
		Transaction Validation	CSNBTRV		
		Trusted Block Create	CSNDTBC		
		Unique Key Derive	CSNBUKD		
X'001F'	**Key Translate**	Key Translate	CSNBKTR	ON	O
X'0021'	**Key Test2 - AES, ENC-ZERO**	Key Test2[1]	CSNBKYT2	ON	O
X'0040'	**Diversified Key Generate - CLR8-ENC**	Diversified Key Generate[2]	CSNBDKG	ON	O, SEL
X'0041'	**Diversified Key Generate - TDES-ENC**	Diversified Key Generate[2]	CSNBDKG	ON	O, SEL

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'0042'	**Diversified Key Generate - TDES-DEC**	Diversified Key Generate[2]	CSNBDKG	ON	O, SEL
X'0043'	**Diversified Key Generate - SESS-XOR**	Diversified Key Generate[2]	CSNBDKG	ON	O, SEL
X'0044'	**Diversified Key Generate - Single length or same halves**	Diversified Key Generate[2]	CSNBDKG	ON	SC, SEL
X'0045'	**Diversified Key Generate - TDES-XOR**	Diversified Key Generate[2]	CSNBDKG	ON	O, SEL
X'0046'	**Diversified Key Generate - TDESEMV2/TDESEMV4**	Diversified Key Generate[2]	CSNBDKG	ON	O, SEL
X'008A'	**MDC Generate**	MDC Generate	CSNBMDG	OFF	R
X'008C'	**Key Generate - Key set**	Key Generate[2]	CSNBKGN	ON	O
X'008E'	**Key Generate - OP**	Key Generate[2] Random Number Generate	CSNBKGN CSNBRNG	ON	R
X'0090'	**Symmetric Key Token Change - RTCMK**	Key Token Change	CSNBKTC	ON	R
X'00A0'	**Clear PIN Generate - 3624**	Clear PIN Generate	CSNBPGN	ON	O
X'00A1'	**Clear PIN Generate - GBP**	Clear PIN Generate	CSNBPGN	ON	O
X'00A2'	**Clear PIN Generate - VISA PVV**	Clear PIN Generate	CSNBPGN	ON	O
X'00A3'	**Clear PIN Generate - Interbank**	Clear PIN Generate	CSNBPGN	ON	O
X'00A4'	**Clear PIN Generate Alternate - 3624 Offset**	Clear PIN Generate Alternate[1]	CSNBCPA	ON	O
X'00AB'	**Encrypted PIN Verify - 3624**	Encrypted PIN Verify[1]	CSNBPVR	ON	O
X'00AC'	**Encrypted PIN Verify - GBP**	Encrypted PIN Verify[1]	CSNBPVR	ON	O
X'00AD'	**Encrypted PIN Verify - VISA PVV**	Encrypted PIN Verify[1]	CSNBPVR	ON	O
X'00AE'	**Encrypted PIN Verify - Interbank**	Encrypted PIN Verify[1]	CSNBPVR	ON	O
X'00AF'	**Clear PIN Encrypt**	Clear PIN Encrypt	CSNBCPE	ON	O
X'00B0'	**Encrypted PIN Generate - 3624**	Encrypted PIN Generate[1]	CSNBEPG	ON	O
X'00B1'	**Encrypted PIN Generate - GBP**	Encrypted PIN Generate[1]	CSNBEPG	ON	O
X'00B2'	**Encrypted PIN Generate - Interbank**	Encrypted PIN Generate[1]	CSNBEPG	ON	O
X'00B3'	**Encrypted PIN Translate - Translate**	Encrypted PIN Translate[1]	CSNBPTR	ON	O
X'00B7'	**Encrypted PIN Translate - Reformat**	Encrypted PIN Translate[1]	CSNBPTR	ON	O
X'00BB'	**Clear PIN Generate Alternate - VISA PVV**	Clear PIN Generate Alternate[1]	CSNBCPA	ON	O
X'00BC'	**PIN Change/Unblock - change EMV PIN with OPINENC**	PIN Change/Unblock[1]	CSNBPCU	ON	O
X'00BD'	**PIN Change/Unblock - change EMV PIN with IPINENC**	PIN Change/Unblock[1]	CSNBPCU	ON	O

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'00C3'	**Clear Key Import/Multiple Clear Key Import - DES**	Clear Key Import Multiple Clear Key Import	CSNBCKI CSNBCKM	ON	SC
X'00C4'	**Secure Key Import - DES, OP**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		ON	O
X'00CD'	**Prohibit Export**	Prohibit Export	CSNBPEX	ON	O
X'00D6'	**Control Vector Translate**	Control Vector Translate	CSNBCVT	ON	SC
X'00D7'	**Key Generate - Key set extended**	Key Generate[2]	CSNBKGN	ON	SC, SUP
X'00DA'	**Cryptographic Variable Encipher**	Cryptographic Variable Encipher	CSNBCVE	ON	NRP, O, SUP
X'00DB'	**Key Generate - SINGLE-R**	Key Generate[2] Remote Key Export[2]	CSNBKGN CSNDRKX	ON	NR, SC
X'00DC'	**Secure Key Import - DES, IM**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		ON	O
X'00DF'	**VISA CVV Generate**	CVV Generate	CSNBCSG	ON	O
X'00E0'	**VISA CVV Verify**	CVV Verify	CSNBCSV	ON	O
X'00E1'	**UKPT - PIN Verify_ PIN Translate**	Encrypted PIN Translate[1] Encrypted PIN Verify[1]	CSNBPTR CSNBPVR	ON	O
X'00E4'	**HMAC Generate - SHA-1**	HMAC Generate	CSNBHMG	ON	O
X'00E5'	**HMAC Generate - SHA-224**	HMAC Generate	CSNBHMG	ON	O
X'00E6'	**HMAC Generate - SHA-256**	HMAC Generate	CSNBHMG	ON	O
X'00E7'	**HMAC Generate - SHA-384**	HMAC Generate	CSNBHMG	ON	O
X'00E8'	**HMAC Generate - SHA-512**	HMAC Generate	CSNBHMG	ON	O
X'00E9'	**Restrict Key Attribute - Export Control**	Restrict Key Attribute	CSNBRKA	ON	O
X'00EA'	**Key Generate2 - OP**	Key Generate2	CSNBKGN2	ON	O
X'00EB'	**Key Generate2 - Key set**	Key Generate2	CSNBKGN2	ON	O
X'00EC'	**Key Generate2 - Key set extended**	Key Generate2	CSNBKGN2	ON	O
X'00F1'	**Symmetric Key Token Change2 - RTCMK**	Key Token Change2	CSNBKTC2	ON	O
X'00F2'	**Secure Key Import2 - OP**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		ON	O
X'00F3'	**Secure Key Import2 - IM**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		ON	O
X'00F4'	**Symmetric Key Import2 - HMAC, PKCSOAEP2**	Symmetric Key Import2	CSNDSYI2	ON	O
X'00F5'	**Symmetric Key Export - HMAC, PKCSOAEP2**	Symmetric Key Export	CSNDSYX	ON	O
X'00F7'	**HMAC Verify - SHA-1**	HMAC Verify	CSNBHMV	ON	O

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'00F8'	**HMAC Verify - SHA-224**	HMAC Verify	CSNBHMV	ON	O
X'00F9'	**HMAC Verify - SHA-256**	HMAC Verify	CSNBHMV	ON	O
X'00FA'	**HMAC Verify - SHA-384**	HMAC Verify	CSNBHMV	ON	O
X'00FB'	**HMAC Verify - SHA-512**	HMAC Verify	CSNBHMV	ON	O
X'00FC'	**Symmetric Key Export - AES, PKOAEP2**	Symmetric Key Export[1]	CSNDSYX	ON	O
X'00FD'	**Symmetric Key Import2 - AES,PKOAEP2**	Symmetric Key Import2[1]	CSNDSYI2	ON	O
X'00FE'	**PKA Key Translate - Translate internal key token**	PKA Key Translate	CSNDPKT	ON	O
X'00FF'	**PKA Key Translate - Translate external key token**	PKA Key Translate	CSNDPKT	ON	O
X'0100'	**Digital Signature Generate**	Digital Signature Generate	CSNDDSG	ON	O, SC
X'0101'	**Digital Signature Verify**	Digital Signature Verify	CSNDDSV	ON	O
X'0102'	**PKA Key Token Change RTCMK**	PKA Key Token Change	CSNDKTC	ON	O
X'0103'	**PKA Key Generate**	PKA Key Generate[1]	CSNDPKG	ON	O, SUP
X'0104'	**PKA Key Import**	PKA Key Import	CSNDPKI	ON	O, SUP
X'0105'	**Symmetric Key Export - DES, PKCS-1.2**	Symmetric Key Export	CSNDSYX	ON	SC
X'0106'	**Symmetric Key Import - DES, PKCS-1.2**	Symmetric Key Import[1]	CSNDSYI	ON	O
X'0109'	**Data Key Import**	Data Key Import	CSNBDKM	ON	O
X'010A'	**Data Key Export**	Data Key Export	CSNBDKX	ON	O
X'010B'	**SET Block Compose**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		ON	O
X'010C'	**SET Block Decompose**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		ON	O
X'010D'	**Symmetric Key Generate - DES, PKA92**	Symmetric Key Generate[1]	CSNDSYG	ON	SC
X'011E'	**PKA Encrypt**	PKA Encrypt	CSNDPKE	ON	O, SEL
X'011F'	**PKA Decrypt**	PKA Decrypt	CSNDPKD	ON	SC, SEL
X'0129'	**Multiple Clear Key Import/Multiple Secure Key Import - AES**	Multiple Clear Key Import	CSNBCKM	ON	SC
X'012A'	**Symmetric Algorithm Encipher - secure AES keys**	Symmetric Algorithm Encipher[1]	CSNBSAE	ON	O
X'012B'	**Symmetric Algorithm Decipher - secure AES keys**	Symmetric Algorithm Decipher[1]	CSNBSAD	ON	O
X'012C'	**Symmetric Key Generate - AES, PKCSOAEP, PKCS-1.2**	Symmetric Key Generate	CSNDSYG	ON	SC
X'012D'	**Symmetric Key Generate - AES, ZERO-PAD**	Symmetric Key Generate	CSNDSYG	ON	SC

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'012E'	**Symmetric Key Import - AES, PKCSOAEP, PKCS-1.2**	Symmetric Key Import	CSNDSYI	ON	O
X'012F'	**Symmetric Key Import - AES, ZERO-PAD**	Symmetric Key Import	CSNDSYI	ON	O
X'0130'	**Symmetric Key Export - AES, PKCSOAEP. PKCS-1.2**	Symmetric Key Export	CSNDSYX	ON	SC
X'0131'	**Symmetric Key Export - AES, ZERO-PAD**	Symmetric Key Export	CSNDSYX	ON	SC
X'013D'	**Diversified Key Generate - Allow wrapping override keywords**	Diversified Key Generate	CSNBDKG	ON	O
X'013E'	**Symmetric Key Generate - Allow wrapping override keywords**	Symmetric Key Generate	CSNDSYG	ON	O
X'013F'	**Remote Key Export - include RKX in default wrap config**	Remote Key Export	CSNDRKX	ON	SC
X'0140'	**Key Part Import - Allow wrapping override keywords**	Key Part Import	CSNBKPI	ON	O
X'0141'	**Multiple Clear Key Import - Allow wrapping override keywords**	Multiple Clear Key Import	CSNBCKM	ON	O
X'0144'	**Symmetric Key Import - Allow wrapping override keywords**	Symmetric Key Import	CSNDSYI	ON	O
X'0149'	**Key Translate2**	Key Translate2	CSNBKTR2	ON	O
X'014A'	**Key Translate2 - Allow wrapping override keywords**	Key Translate2	CSNBKTR2	ON	O
X'014B'	**Key Translate2 - Allow use of REFORMAT**	Key Translate2	CSNBKTR2	ON	O
X'014D'	**TR31 Export - Permit version A TR-31 key blocks**	Key Export to TR31[1]	CSNBT31X	ON	O
X'014E'	**TR31 Export - Permit version B TR-31 key blocks**	Key Export to TR31[1]	CSNBT31X	ON	O
X'014F'	**TR31 Export - Permit version C TR-31 key blocks**	Key Export to TR31[1]	CSNBT31X	ON	O
X'0150'	**TR31 Import - Permit version A TR-31 key blocks**	TR31 Key Import[1]	CSNBT31I	ON	O
X'0151'	**TR31 Import - Permit version B TR-31 key blocks**	TR31 Key Import[1]	CSNBT31I	ON	O
X'0152'	**TR31 Import - Permit version C TR-31 key blocks**	TR31 Key Import[1]	CSNBT31I	ON	O
X'0153'	**TR31 Import - Permit override of default wrapping method**	TR31 Key Import[1]	CSNBT31I	ON	O, SC
X'0154'	**Restrict Key Attribute - Permit setting the TR-31 export bit**	Restrict Key Attribute[1]	CSNBRKA	ON	O
X'0155'	**CVV Key Combine**	CVV Key Combine	CSNBCKC	ON	O

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'0156'	**CVV Key Combine - Allow wrapping override keywords**	CVV Key Combine[1]	CSNBCKC	ON	O, SC
X'0157'	**CVV Key Combine - Permit mixed key types**	CVV Key Combine[1]	CSNBCKC	ON	O, SC
X'0158'	**TR31 Export - Permit any CCA key if INCL-CV is specified**	Key Export to TR31[1]	CSNBT31X	ON	O, SC
X'015A'	**TR31 Import - Permit C0 to MAC/MACVER:CVVKEY-A**	TR31 Key Import[1]	CSNBT31I	OFF	O, SC
X'015B'	**TR31 Import - Permit C0 to MAC/MACVER:AMEX-CSC**	TR31 Key Import[1]	CSNBT31I	OFF	O, SC
X'015C'	**TR31 Import - Permit K0:E to EXPORTER/OKEYXLAT**	TR31 Key Import[1]	CSNBT31I	OFF	O, SC
X'015D'	**TR31 Import - Permit K0:D to IMPORTER/IKEYXLAT**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'015E'	**TR31 Import - Permit K0:B to EXPORTER/OKEYXLAT**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'015F'	**TR31 Import - Permit K0:B to IMPORTER/IKEYXLAT**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'0160'	**TR31 Import - Permit K1:E to EXPORTER/OKEYXLAT**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'0161'	**TR31 Import - Permit K1:D to IMPORTER/IKEYXLAT**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'0162'	**TR31 Import - Permit K1:B to EXPORTER/OKEYXLAT**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'0163'	**TR31 Import - Permit K1:B to IMPORTER/IKEYXLAT**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'0164'	**TR31 Import - Permit M0/M1/M3 to MAC/MACVER:ANY-MAC**	TR31 Key Import[1]	CSNBT31I	ON	O
X'0165'	**TR31 Import - Permit P0:E to OPINENC**	TR31 Key Import[1]	CSNBT31I	ON	O
X'0166'	**TR31 Import - Permit P0:D to IPINENC**	TR31 Key Import[1]	CSNBT31I	ON	O
X'0167'	**TR31 Import - Permit V0 to PINGEN:NO-SPEC**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'0168'	**TR31 Import - Permit V0 to PINVER:NO-SPEC**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'0169'	**TR31 Import - Permit V1 to PINGEN:IBM-PIN/IBM-PINO**	TR31 Key Import[1]	CSNBT31I	ON	O
X'016A'	**TR31 Import - Permit V1 to PINVER:IBM-PIN/IBM-PINO**	TR31 Key Import[1]	CSNBT31I	ON	O
X'016B'	**TR31 Import - Permit V2 to PINGEN:VISA-PVV**	TR31 Key Import[1]	CSNBT31I	ON	O
X'016C'	**TR31 Import - Permit V2 to PINVER:VISA-PVV**	TR31 Key Import[1]	CSNBT31I	ON	O
X'016D'	**TR31 Import - Permit E0 to DKYGENKY:DKYL0+DMAC**	TR31 Key Import[1]	CSNBT31I	OFF	O

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'016E'	**TR31 Import - Permit E0 to DKYGENKY:DKYL0+DMV**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'016F'	**TR31 Import - Permit E0 to DKYGENKY:DKYL1+DMAC**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'0170'	**TR31 Import - Permit E0 to DKYGENKY:DKYL1+DMV**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'0171'	**TR31 Import - Permit E1 to DKYGENKY:DKYL0+DMPIN**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'0172'	**TR31 Import - Permit E1 to DKYGENKY:DKYL0+DDATA**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'0173'	**TR31 Import - Permit E1 to DKYGENKY:DKYL1+DMPIN**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'0174'	**TR31 Import - Permit E1 to DKYGENKY:DKYL1+DDATA**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'0175'	**TR31 Import - Permit E2 to DKYGENKY:DKYL0+DMAC**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'0176'	**TR31 Import - Permit E2 to DKYGENKY:DKYL1+DMAC**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'0177'	**TR31 Import - Permit E3 to ENCIPHER**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'0178'	**TR31 Import - Permit E4 to DKYGENKY:DKYL0+DDATA**	TR31 Key Import[1]	CSNBT31I	ON	O
X'0179'	**TR31 Import - Permit E5 to DKYGENKY:DKYL0+DMAC**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'017A'	**TR31 Import - Permit E5 to DKYGENKY:DKYL0+DDATA**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'017B'	**TR31 Import - Permit E5 to DKYGENKY:DKYL0+DEXP**	TR31 Key Import[1]	CSNBT31I	OFF	O
X'017C'	**TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER**	TR31 Key Import[1]	CSNBT31I	OFF	O, SC
X'0180'	**TR31 Export - Permit KEYGENKY:UKPT to B0**	Key Export to TR31[1]	CSNBT31X	ON	O
X'0181'	**TR31 Export - Permit MAC/MACVER:AMEX-CSC to C0:G/C/V**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'0182'	**TR31 Export - Permit MAC/MACVER:CVV-KEYA to C0:G/C/V**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'0183'	**TR31 Export - Permit MAC/MACVER:ANY-MAC to C0:G/C/V**	Key Export to TR31[1]	CSNBT31X	ON	O
X'0184'	**TR31 Export - Permit DATA to C0:G/C**	Key Export to TR31[1]	CSNBT31X	ON	O
X'0185'	**TR31 Export - Permit ENCIPHER/DECIPHER/CIPHER to D0:E/D/B**	Key Export to TR31[1]	CSNBT31X	ON	O

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'0186'	**TR31 Export - Permit DATA to D0:B**	Key Export to TR31[1]	CSNBT31X	ON	O
X'0187'	**TR31 Export - Permit EXPORTER/OKEYXLAT to K0:E**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'0188'	**TR31 Export - Permit IMPORTER/IKEYXLAT to K0:D**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'0189'	**TR31 Export - Permit EXPORTER/OKEYXLAT to K1:E**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'018A'	**TR31 Export - Permit IMPORTER/IKEYXLAT to K1:D**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'018B'	**TR31 Export - Permit MAC/DATA/DATAM to M0:G/C**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'018C'	**TR31 Export - Permit MACVER/DATAMV to M0:V**	Key Export to TR31[1]	CSNBT31X	ON	O
X'018D'	**TR31 Export - Permit MAC/DATA/DATAM to M1:G/C**	Key Export to TR31[1]	CSNBT31X	ON	O
X'018E'	**TR31 Export - Permit MACVER/DATAMV to M1:V**	Key Export to TR31[1]	CSNBT31X	ON	O
X'018F'	**TR31 Export - Permit MAC/DATA/DATAM to M3:G/C**	Key Export to TR31[1]	CSNBT31X	ON	O
X'0190'	**TR31 Export - Permit MACVER/DATAMV to M3:V**	Key Export to TR31[1]	CSNBT31X	ON	O
X'0191'	**TR31 Export - Permit OPINENC to P0:E**	Key Export to TR31[1]	CSNBT31X	ON	O
X'0192'	**TR31 Export - Permit IPINENC to P0:D**	Key Export to TR31[1]	CSNBT31X	ON	O
X'0193'	**TR31 Export - Permit PINVER:NO-SPEC to V0**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'0194'	**TR31 Export - Permit PINGEN:NO-SPEC to V0**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'0195'	**TR31 Export - Permit PINVER:NO-SPEC/IBM-PIN/IBM-PINO to V1**	Key Export to TR31[1]	CSNBT31X	ON	O
X'0196'	**TR31 Export - Permit PINGEN:NO-SPEC/IBM-PIN/IBM-PINO to V1**	Key Export to TR31[1]	CSNBT31X	ON	O
X'0197'	**TR31 Export - Permit PINVER:NO-SPEC/VISA-PVV to V2**	Key Export to TR31[1]	CSNBT31X	ON	O
X'0198'	**TR31 Export - Permit PINGEN:NO-SPEC/VISA-PVV to V2**	Key Export to TR31[1]	CSNBT31X	ON	O
X'0199'	**TR31 Export - Permit DKYGENKY:DKYL0+DMAC to E0**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'019A'	**TR31 Export - Permit DKYGENKY:DKYL0+DMV to E0**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'019B'	**TR31 Export - Permit DKYGENKY:DKYL0+DALL to E0**	Key Export to TR31[1]	CSNBT31X	OFF	O

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'019C'	**TR31 Export - Permit DKYGENKY:DKYL1+DMAC to E0**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'019D'	**TR31 Export - Permit DKYGENKY:DKYL1+DMV to E0**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'019E'	**TR31 Export - Permit DKYGENKY:DKYL1+DALL to E0**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'019F'	**TR31 Export - Permit DKYGENKY:DKYL0+DDATA to E1**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'01A0'	**TR31 Export - Permit DKYGENKY:DKYL0+DMPIN to E1**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'01A1'	**TR31 Export - Permit DKYGENKY:DKYL0+DALL to E1**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'01A2'	**TR31 Export - Permit DKYGENKY:DKYL1+DDATA to E1**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'01A3'	**TR31 Export - Permit DKYGENKY:DKYL1+DMPIN to E1**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'01A4'	**TR31 Export - Permit DKYGENKY:DKYL1+DALL to E1**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'01A5'	**TR31 Export - Permit DKYGENKY:DKYL0+DMAC to E2**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'01A6'	**TR31 Export - Permit DKYGENKY:DKYL0+DALL to E2**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'01A7'	**TR31 Export - Permit DKYGENKY:DKYL1+DMAC to E2**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'01A8'	**TR31 Export - Permit DKYGENKY:DKYL1+DALL to E2**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'01A9'	**TR31 Export - Permit DATA/MAC/CIPHER/ENCIPHER to E3**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'01AA'	**TR31 Export - Permit DKYGENKY:DKYL0+DDATA to E4**	Key Export to TR31[1]	CSNBT31X	ON	O
X'01AB'	**TR31 Export - Permit DKYGENKY:DKYL0+DALL to E4**	Key Export to TR31[1]	CSNBT31X	ON	O
X'01AC'	**TR31 Export - Permit DKYGENKY:DKYL0+DEXP to E5**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'01AD'	**TR31 Export - Permit DKYGENKY:DKYL0+DMAC to E5**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'01AE'	**TR31 Export - Permit DKYGENKY:DKYL0+DDATA to E5**	Key Export to TR31[1]	CSNBT31X	OFF	O
X'01AF'	**TR31 Export - Permit DKYGENKY:DKYL0+DALL to E5**	Key Export to TR31[1]	CSNBT31X	ON	O
X'01B0'	**TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N**	Key Export to TR31[1]	CSNBT31X	OFF	O, SC
X'01C0'	**Cipher Text Translate2**	Cipher Text Translate2	CSNBCTT2	ON	O
X'01C1'	**Cipher Text Translate2 - Allow translate from AES to TDES**	Cipher Text Translate2	CSNBCTT2	ON	SC

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'01C2'	**Cipher Text Translate2 - Allow translate to weaker AES**	Cipher Text Translate2	CSNBCTT2	ON	SC
X'01C3'	**Cipher Text Translate2 - Allow translate to weaker DES**	Cipher Text Translate2	CSNBCTT2	ON	SC
X'01C4'	**Cipher Text Translate2 - Allow only cipher text translate types**	Cipher Text Translate2	CSNBCTT2	OFF	O
X'01C8'	**Unique Key Derive**	Unique Key Derive	CSNBUKD	ON	O
X'01C9'	**Unique Key Derive - Allow PIN-DATA processing**	Unique Key Derive	CSNBUKD	OFF	NR
X'01CA'	**Unique Key Derive - Override default wrapping**	Unique Key Derive	CSNBUKD	ON	O
X'01CB'	**Key Test - Warn when keyword inconsistent with key length**	Key Test Extended	CSNBKYTX	OFF	O
X'0203'	**Retained Key Delete**	Retained Key Delete	CSNDRKD	ON	O, SEL
X'0204'	**PKA Key Generate - Clone**	PKA Key Generate[1]	CSNDPKG	ON	O
X'0205'	**PKA Key Generate - Clear**	PKA Key Generate[1]	CSNDPKG	ON	O, SUP
X'0206'	**PKA Encipher - Clear Key Disallow PKCS-1.2**	PKA Encrypt	CSNDPKE	ON	O
X'0207'	**PKA Encipher - Clear Key Disallow ZERO-PAD**	PKA Encrypt	CSNDPKE	ON	O
X'0208'	**PKA Encipher - Clear Key Disallow MRP**	PKA Encrypt	CSNDPKE	ON	O
X'0209'	**PKA Encipher - Clear Key Disallow PKCSOAEP**	PKA Encrypt	CSNDPKE	ON	O
X'020A'	**PKA Decipher - Key Data Disallow PKCS-1.2**	PKA Decrypt	CSNDPKD	ON	O
X'020B'	**PKA Decipher - Key Data Disallow ZERO-PAD**	PKA Decrypt	CSNDPKD	ON	O
X'020C'	**PKA Decipher - Key Data Disallow PKCSOAEP**	PKA Decrypt	CSNDPKD	ON	O
X'0230'	**Retained Key List**	Retained Key List	CSNDRKL	ON	O
X'0235'	**Symmetric Key Import - DES, PKA92 KEK**	Symmetric Key Import[1]	CSNDSYI	ON	O
X'023C'	**Symmetric Key Generate - DES, ZERO-PAD**	Symmetric Key Generate[1]	CSNDSYG	ON	O, SC
X'023D'	**Symmetric Key Import - DES, ZERO-PAD**	Symmetric Key Import[1]	CSNDSYI	ON	O, SC
X'023E'	**Symmetric Key Export - DES, ZERO-PAD**	Symmetric Key Export[1]	CSNDSYX	ON	O, SC
X'023F'	**Symmetric Key Generate - DES, PKCS-1.2**	Symmetric Key Generate[1]	CSNDSYG	ON	O, SC
X'0261'	**TKE Authorization for domain 0**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'0262'	**TKE Authorization for domain 1**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'0263'	**TKE Authorization for domain 2**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'0264'	**TKE Authorization for domain 3**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'0265'	**TKE Authorization for domain 4**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'0266'	**TKE Authorization for domain 5**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'0267'	**TKE Authorization for domain 6**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'0268'	**TKE Authorization for domain 7**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'0269'	**TKE Authorization for domain 8**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'026A'	**TKE Authorization for domain 9**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'026B'	**TKE Authorization for domain 10**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'026C'	**TKE Authorization for domain 11**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'026D'	**TKE Authorization for domain 12**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'026E'	**TKE Authorization for domain 13**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'026F'	**TKE Authorization for domain 14**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'0270'	**TKE Authorization for domain 15**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'0273'	**Secure Messaging for Keys**	Secure Messaging for Keys	CSNBSKY	ON	O
X'0274'	**Secure Messaging for PINs**	Secure Messaging for PINs	CSNBSPN	ON	O

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'0275'	**DATAM Key Management Control**	Diversified Key Generate Data Key Import Data Key Export Key Export Key Generate Key Import	CSNBDKG CSNBDKM CSNBDKX CSNBKEX CSNBKGN CSNBKIM	ON	O
X'0276'	**Key Export - Unrestricted**	Key Export	CSNBKEX	ON	O, SC
X'0277'	**Data Key Export - Unrestricted**	Data Key Export	CSNBDKX	ON	O, SC
X'0278'	**Key Part Import - ADD-PART**	Key Part Import[1]	CSNBKPI	ON	SC, SEL
X'0279'	**Key Part Import - COMPLETE**	Key Part Import[1]	CSNBKPI	ON	SC, SEL
X'027A'	**Key Part Import - Unrestricted**	Key Part Import	CSNBKPI	ON	O, SC
X'027B'	**Key Import - Unrestricted**	Key Import	CSNBKIM	ON	O, SC
X'027C'	**Data Key Import - Unrestricted**	Data Key Import	CSNBDKM	ON	O, SC
X'027D'	**PKA Key Generate - Permit Regeneration Data**	PKA Key Generate[1]	CSNDPKG	ON	O, NRP, SC
X'027E'	**PKA Key Generate - Permit Regeneration Data Retain**	PKA Key Generate[1]	CSNDPKG	ON	O, NRP, SC
X'0290'	**Diversified Key Generate - DKYGENKY - DALL**	Diversified Key Generate[2] PIN Change/Unblock[2]	CSNBDKG CSNBPCU	OFF	O, SC
X'0291'	**Transaction Validation - Generate**	Transaction Validation[1]	CSNBTRV	ON	O, SEL
X'0292'	**Transaction Validation - Verify CSC-3**	Transaction Validation[1]	CSNBTRV	ON	O
X'0293'	**Transaction Validation - Verify CSC-4**	Transaction Validation[1]	CSNBTRV	ON	O
X'0294'	**Transaction Validation - Verify CSC-5**	Transaction Validation[1]	CSNBTRV	ON	O
X'0295'	**Symmetric Key Encipher/Decipher - Encrypted DES keys**	Enables CPACF key translation for DES keys.	N/A	ON	O
X'0296'	**Symmetric Key Encipher/Decipher - Encrypted AES keys**	Enables CPACF key translation for AES keys.	N/A	ON	O
X'0297'	**Key Part Import2 - Load first key part, require 3 key parts**	Key Part Import2	CSNBKPI2	ON	O
X'0298'	**Key Part Import2 - Load first key part, require 2 key parts**	Key Part Import2	CSNBKPI2	ON	O
X'0299'	**Key Part Import2 - Load first key part, require 1 key parts**	Key Part Import2	CSNBKPI2	ON	O
X'029A'	**Key Part Import2 - Add second of 3 or more key parts**	Key Part Import2	CSNBKPI2	ON	O
X'029B'	**Key Part Import2 - Add last required key part**	Key Part Import2	CSNBKPI2	ON	O
X'029C'	**Key Part Import2 - Add optional key part**	Key Part Import2	CSNBKPI2	ON	O

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'029D'	**Key Part Import2 - Complete key**	Key Part Import2	CSNBKPI2	ON	SEL
X'029E'	**Operational Key Load - Variable-Length Tokens**	Key Part Import2	CSNBKPI2	ON	O
X'02B0'	**Recover PIN from Offset**	Recover PIN from Offset	CSNBPFO	ON	O
X'02B1'	**Authentication Parameter Generate**	Authentication Parameter Generate	CSNBAPG	ON	O
X'02B2'	**Authentication Parameter Generate - Clear**	Authentication Parameter Generate[1]	CSNBAPG	ON	O
X'02B3'	**Symmetric Key Export - AESKWCV**	Symmetric Key Export	CSNDSYX	ON	O
X'02B4'	**Symmetric Key Import2 - AESKWCV**	Symmetric Key Import2	CSNDSYI2	ON	O
X'02B5'	**Symmetric Key Export with Data**	Symmetric Key Export with Data	CSNDSXD	ON	O
X'02B6'	**Symmetric Key Export with Data - Special**	Symmetric Key Export with Data	CSNDSXD	ON	O
X'02B8'	**Diversified Key Generate - TDES-CBC**	Diversified Key Generate	CSNBDKG2	ON	O
X'02B9'	**Symmetric Key Import2 - Allow wrapping override keywords**	Symmetric Key Import2[1]	CSNDSYI2	ON	O
X'02BA'	**Remote Key Export - Allow wrapping override keywords**	Remote Key Export[1]	CSNDRKX	ON	O
X'02BB'	**Key Generate2 - DK PIN key set**	Key Generate2[1]	CSNBKGN2	ON	O
X'02BC'	**Key Generate2 - DK PIN print key**	Key Generate2[1]	CSNBKGN2	ON	O
X'02BD'	**Key Generate2 - DK PIN admin1 key set PINPROT**	Key Generate2[1]	CSNBKGN2	ON	O
X'02BE'	**Key Generate2 - DK PIN admin1 key set MAC**	Key Generate2[1]	CSNBKGN2	ON	O
X'02BF'	**Key Generate2 - DK PIN admin2 key set MAC**	Key Generate2[1]	CSNBKGN2	ON	O
X'02C0'	**DK Random PIN Generate**	DK Random PIN Generate	CSNBDRPG	ON	O
X'02C1'	**DK PIN Verify**	DK PIN Verify	CSNBDPV	ON	O
X'02C2'	**DK PIN Change**	DK PIN Change	CSNBDPC	ON	O
X'02C3'	**DK PRW Card Number Update**	DK PRW Card Number Update	CSNBDPNU	ON	O
X'02C4'	**DK PRW CMAC Generate**	DK PRW CMAC Generate	CSNBDPCG	ON	O
X'02C5'	**DK PAN Modify in Transaction**	DK PAN Modify in Transaction	CSNBDPMT	ON	O
X'02C6'	**DK Deterministic PIN Generate**	DK Deterministic PIN Generate	CSNBDDPG	ON	O
X'02C7'	**DK PAN Translate**	DK PAN Translate	CSNBDPT	ON	O
X'02C8'	**DK Regenerate PRW**	DK Regenerate PRW	CSNBDRP	ON	O
X'02CC'	**Diversified Key Generate2 - AES EMV1 SESS**	Diversified Key Generate2[1]	CSNBDKG2	ON	O
X'02CD'	**Diversified Key Generate2 - DALL**	Diversified Key Generate2[1]	CSNBDKG2	ON	O

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'02CE'	**DK Migrate PIN**	DK Migrate PIN	CSNBDMP	ON	O
X'02CF'	**FPE Encrypt**	FPE Encipher	CSNBFPEE	ON	ID, R
X'02D0'	**FPE Decrypt**	FPE Decipher	CSNBFPED	ON	ID, R
X'02D1'	**FPE Translate**	FPE Translate	CSNBFPET	ON	ID, R
X'02D2'	**Diversified Key Generate2 - MK-OPTC**	Diversified Key Generate2	CSNBDKG2		
X'02D3'	**Diversified Key Generate2 - KDFFM-DK**	Diversified Key Generate2	CSNBDKG2		
X'02D4'	**Diversified Key Generate2 - KDFFM-DK**	Diversified Key Generate2	CSNBDKG2		
X'02D5'	**Encrypted PIN Translate Enhanced**	Encrypted PIN Translate Enhanced	CSNBPTRE		
X'0300'	**NOCV KEK usage for export-related functions**	Data Key Export Key Export Key Generate Remote Key Export	CSNBDKX CSNBKEX CSNBKGN CSNDRKX	ON	O
X'0301'	**Prohibit Export Extended**	Prohibit Export Extended	CSNBPEXX	ON	O
X'0309'	**Operational Key Load**	Key Part Import	CSNBKPI	ON	O
X'030A'	**NOCV KEK usage for import-related functions**	Data Key Import Key Import Key Generate Remote Key Export	CSNBDKM CSNBKIM CSNBKGN CSNDRKX	ON	O
X'030C'	**DSG ZERO-PAD unrestricted hash length**	Digital Signature Generate	CSNDDSG	OFF	O, SC
X'030F'	**Trusted Block Create - Create Block in inactive form**	Trusted Block Create	CSNDTBC	ON	O, SUP
X'0310'	**Trusted Block Create - Activate an inactive block**	Trusted Block Create	CSNDTBC	ON	O, SUP
X'0311'	**PKA Key Import - Import an external trusted block**	PKA Key Import	CSNDPKI	ON	O, SEL
X'0312'	**Remote Key Export - Gen or export a non-CCA node key**	Remote Key Export	CSNDRKX	ON	O, SEL
X'0313'	**Enhanced PIN Security**	Clear PIN Generate Alternate Clear PIN Encrypt Encrypted PIN Generate Encrypted PIN Translate Encrypted PIN Verify PIN Change/Unblock	CSNBCPA CSNBCPE CSNBEPG CSNBPTR CSNBPVR CSNBPCU	OFF	O, SC, SEL
X'0318'	**PKA Key Translate - from CCA RSA to SC Visa Format**	PKA Key Translate	CSNDPKT	ON	O
X'0319'	**PKA Key Translate - from CCA RSA to SC ME Format**	PKA Key Translate	CSNDPKT	ON	O
X'031A'	**PKA Key Translate - from CCA RSA to SC CRT Format**	PKA Key Translate	CSNDPKT	ON	O

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'031B'	**PKA Key Translate - from source EXP KEK to target EXP KEK**	PKA Key Translate	CSNDPKT	ON	O
X'031C'	**PKA Key Translate - from source IMP KEK to target EXP KEK**	PKA Key Translate	CSNDPKT	ON	O
X'031D'	**PKA Key Translate - from source IMP KEK to target IMP KEK**	PKA Key Translate	CSNDPKT	ON	O
X'0326'	**PKA Key Generate - Clear ECC keys**	PKA Key Generate	CSNDPKG	ON	O
X'0327'	**Symmetric Key Export - AESKW**	Symmetric Key Export	CSNDSYX	ON	O, R
X'0329'	**Symmetric Key Import2 - AESKW**	Symmetric Key Import2[1]	CSNDSYI2	ON	O, R
X'032A'	**Key Translate2 - Disallow AES ver 5 to ver 4 conversion**	Key Translate2[1]	CSNBKTR2	OFF	O, R
X'032B'	**Symmetric Key Import2 - disallow weak import**	Symmetric Key Import2[1]	CSNDSYI2	OFF	O, R
X'032E'	**TBC - Disallow triple-length MAC key**	Trusted Block Create	CSNDTBC	OFF	O
X'0331'	**Allow weak DES wrap of RSA**	PKA Key Generate	CSNDPKG	OFF	O, R
X'0334'	**Key Translate2 - Translate fixed to variable payload**	Key Translate2	CSNBKTR2	ON	SC
X'0335'	**Unique Key Derive - K3IPEK**	Unique Key Derive	CSNBUKD	ON	SC
X'0336'	**MAC Generate2 - AES CMAC**	MAC Generate2	CSNBMGN2	ON	O
X'0337'	**MAC Verify2 - AES CMAC**	MAC Verify2	CSNBMVR2	ON	O
X'0338'	**PKA Key Translate - from CCA RSA CRT to EMV DDA format**	PKA Key Translate[1]	CSNDPKT	ON	O
X'0339'	**PKA Key Translate - from CCA RSA CRT to EMV DDAE format**	PKA Key Translate[1]	CSNDPKT	ON	O
X'033A'	**PKA Key Translate - from CCA RSA CRT to EMV CRT format**	PKA Key Translate[1]	CSNDPKT	ON	O
X'0350'	**ANSI X9.8 PIN - Enforce PIN block restrictions**	Clear PIN Generate Alternate Encrypted PIN Translate Secure Messaging for PINs	CSNBCPA CSNBPTR CSNBSPN	OFF	O, R
X'0351'	**ANSI X9.8 PIN - Allow modification of PAN_01_0350**	Encrypted PIN Translate Secure Messaging for PINs	CSNBPTR CSNBSPN	OFF	O, SC
X'0352'	**ANSI X9.8 PIN - Allow only ANSI PIN blocks**	Encrypted PIN Translate Secure Messaging for PINs	CSNBPTR CSNBSPN	OFF	O, SC
X'0353'	**ANSI X9.8 PIN - Load Decimalization Tables**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'0354'	**ANSI X9.8 PIN - Delete Decimalization Tables**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'0355'	**ANSI X9.8 PIN - Activate Decimalization Tables**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'0356'	**ANSI X9.8 PIN - Use stored decimmalization tables only**	Clear PIN Generate[1] Clear PIN Generate Alternate[1] Encrypted PIN Generate[1] Encrypted PIN Verify[1]	CSNBPGN CSNBCPA CSNBEPG CSNBPVR	OFF	O, R
X'0360'	**ECC Diffie-Hellmann**	EC Diffie-Hellman[1]	CSNDEDH	ON	O
X'0361'	**EC Diffie-Hellman - Allow PASSTHRU**	EC Diffie-Hellman[1]	CSNDEDH	ON	O
X'0362'	**ECC Diffie-Hellmann - Allow key wrap override**	EC Diffie-Hellman[1]	CSNDEDH	ON	O
X'0363'	**ECC Diffie-Hellmann - Allow Prime Curve 192**	EC Diffie-Hellman[1]	CSNDEDH	ON	O
X'0364'	**ECC Diffie-Hellmann - Allow Prime Curve 224**	EC Diffie-Hellman[1]	CSNDEDH	ON	O
X'0365'	**ECC Diffie-Hellmann - Allow Prime Curve 256**	EC Diffie-Hellman[1]	CSNDEDH	ON	O
X'0366'	**ECC Diffie-Hellmann - Allow Prime Curve 384**	EC Diffie-Hellman[1]	CSNDEDH	ON	O
X'0367'	**ECC Diffie-Hellmann - Allow Prime Curve 521**	EC Diffie-Hellman[1]	CSNDEDH	ON	O
X'0368'	**ECC Diffie-Hellmann - Allow BP Curve 160**	EC Diffie-Hellman[1]	CSNDEDH	ON	O
X'0369'	**ECC Diffie-Hellmann - Allow BP Curve 192**	EC Diffie-Hellman[1]	CSNDEDH	ON	O
X'036A'	**ECC Diffie-Hellmann - Allow BP Curve 224**	EC Diffie-Hellman[1]	CSNDEDH	ON	O
X'036B'	**ECC Diffie-Hellmann - Allow BP Curve 256**	EC Diffie-Hellman[1]	CSNDEDH	ON	O
X'036C'	**ECC Diffie-Hellmann - Allow BP Curve 320**	EC Diffie-Hellman[1]	CSNDEDH	ON	O
X'036D'	**ECC Diffie-Hellmann - Allow BP Curve 384**	EC Diffie-Hellman[1]	CSNDEDH	ON	O
X'036E'	**ECC Diffie-Hellmann - Allow BP Curve 512**	EC Diffie-Hellman[1]	CSNDEDH	ON	O
X'036F'	**ECC Diffie-Hellman - Prohibit weak key generate**	EC Diffie-Hellman	CSNDEDH	OFF	O
X'0370'	**CSNBKPIT: Allow load 1st key part for a key with min 3 key parts**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'0371'	**CSNBKPIT: Allow load 1st key part for a key with min 2 key parts**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O

Table 270. Access Control Points and corresponding CCA verbs (continued)

ACP number (hex)	Name of ACP from TKE interface	Verb name	Entry point	Initial setting	Usage
X'0372'	**CSNBKPIT: Allow load 1st key part for a key with min 1 key part**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'0373'	**CSNBKPIT: Allow load 2nd and later key part for a key requiring more key parts**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'0374'	**CSNBKPIT: Allow load last key part for a key**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'0375'	**CSNBKPIT: Allow load an optional key part for a key**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'0376'	**CSNBKPIT: Allow completing a key that has all key parts loaded**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O
X'0377'	**CSNBKPIT: Allow clearing a key part register**	**Note:** This ACP is included for TKE reference only, the service impacted is available only (for IBM z Systems) on z/OS.		OFF	O

The following codes are used in this table:

ID Initial default.

O Usage of this command is optional; enable it as required for authorized usage.

R Enabling this command is recommended.

NR Enabling this command is **not** recommended.

NRP Enabling this command is **not** recommended for production.

SC Usage of this command requires special consideration.

SEL Usage of this command is normally restricted to one or more selected roles.

SUP This command is normally restricted to one or more supervisory roles.

1 This verb performs more than one function, as determined by the keyword in the *rule_array* parameter of the verb call. Not all functions of the verb require the command in this row.

2 This verb does not always require the command in this row. Use as determined by the control vector for the key and the action being performed.

Managing ACPs using a TKE workstation

The TKE workstation allows you to enable or disable access control points for verbs.

For systems that do not use the optional TKE workstation, most access control points (current and new) are enabled in the DEFAULT Role with the appropriate licensed internal code on the CEX*C. For more information about the TKE workstation, see *z/OS Cryptographic Services ICSF: Trusted Key Entry PCIX Workstation User's Guide*.

For information about required TKE versions for accessing the various CEX*C features, see "CEX5C information" on page 1002.

Use of particular cryptographic or key management verb functions with the CEX*C are controlled through access control points. You can see the default settings of an access control point in Table 270 on page 954 as “Initial setting”.

Note:

1. Access control points DKYGENKY-DALL and DSG ZERO-PAD unrestricted hash length are always disabled in the DEFAULT role for all customers (TKE and non-TKE). A TKE workstation is required to enable these access control points.
2. When you modify the setting of an access control point, please be sure to use a procedure according to your organization's security policy. TKE workstation versions earlier than V6.0 do not show the current setting of the access control points. TKE workstation versions 6.0 and higher show the current setting, but neither show the default settings nor a change history of the listed access control points. If you do not remember the change history, note that using the **Zeroize** function of the card or of the domain in order to reset all access control point settings to their default values discards all keys.

Chapter 23. Sample verb call routines

This appendix contains sample verb call routines for both C and Java.

Important: The user must load the Symmetric Master Key before the verb calls complete successfully. Otherwise return code 12 and reason code 764 is returned.

To illustrate the practical application of CCA verb calls, this appendix describes the sample routines included with the RPM. A sample in C, and one in Java is included.

The sample routines generate a Message Authentication Code (MAC) on a text string, and then verifies the MAC. To accomplish this, the routine:

- Calls the Key Generate (CSNBKGN or CSNBKGNJ) verb to create a MAC/MACVER key pair.
- Calls the MAC Generate (CSNBMGN or CSNBMGNJ) verb to generate a MAC on a text string with the MAC key.
- Calls the MAC Verify (CSNBMVR or CSNBMVRJ) verb to verify the text string MAC with the MACVER key.

As you review the sample routines shown in Figure 44 and Figure 45 on page 982, refer to the chapters in this book for descriptions of the called verbs and their parameters. These verbs are listed in Table 271.

Table 271. Verbs called by the sample routines

Verb	Entry point name for C and Java versions
Key Generate	CSNBKGN or CSNBKGNJ
MAC Generate	CSNBMGN or CSNBMGNJ
MAC Verify	CSNBMVR or CSNBMVRJ

Sample program in C

This sample code, which consists of a C program (mac.c) and a makefile (makefile.Inx), can be found in the /opt/IBM/CCA/samples directory.

For reference, a copy of the sample routine is shown in Figure 44.

Figure 44. Syntax, sample routine in C

```
/*****************************************************************/
/*                                                               */
/* Module Name: mac.c                                            */
/*                                                               */
/* DESCRIPTIVE NAME: Cryptographic Coprocessor Support Program   */
/*                   C language source code example              */
/*                                                               */
/*---------------------------------------------------------------*/
/*                                                               */
/* Licensed Materials - Property of IBM                          */
/*                                                               */
/* (C) Copyright IBM Corp. 1997-2001 All Rights Reserved         */
/*                                                               */
```

```
/* US Government Users Restricted Rights - Use duplication or        */
/* disclosure restricted by GSA ADP Schedule Contract with IBM Corp. */
/*                                                                   */
/*-------------------------------------------------------------------*/
/*                                                                   */
/*        NOTICE TO USERS OF THE SOURCE CODE EXAMPLES                */
/*                                                                   */
/* The source code examples provided by IBM are only intended to     */
/* assist in the development of a working software program. The       */
/* source code examples do not function as written: additional       */
/* code is required. In addition, the source code examples may       */
/* not compile and/or bind successfully as written.                  */
/*                                                                   */
/* International Business Machines Corporation provides the source   */
/* code examples, both individually and as one or more groups,       */
/* "as is" without warranty of any kind, either expressed or         */
/* implied, including, but not limited to the implied warranties of  */
/* merchantability and fitness for a particular purpose. The entire  */
/* risk as to the quality and performance of the source code         */
/* examples, both individually and as one or more groups, is with    */
/* you. Should any part of the source code examples prove defective, */
/* you (and not IBM or an authorized dealer) assume the entire cost  */
/* of all necessary servicing, repair or correction.                 */
/*                                                                   */
/* IBM does not warrant that the contents of the source code         */
/* examples, whether individually or as one or more groups, will     */
/* meet your requirements or that the source code examples are       */
/* error-free.                                                       */
/*                                                                   */
/* IBM may make improvements and/or changes in the source code       */
/* examples at any time.                                             */
/*                                                                   */
/* Changes may be made periodically to the information in the        */
/* source code examples; these changes may be reported, for the      */
/* sample code included herein, in new editions of the examples.     */
/*                                                                   */
/* References in the source code examples to IBM products, programs, */
/* or services do not imply that IBM intends to make these           */
/* available in all countries in which IBM operates. Any reference   */
/* to the IBM licensed program in the source code examples is not    */
/* intended to state or imply that IBM's licensed program must be    */
/* used. Any functionally equivalent program may be used.            */
/*                                                                   */
/*-------------------------------------------------------------------*/
/*                                                                   */
/* This example program:                                             */
/*                                                                   */
/* 1) Calls the Key_Generate verb (CSNBKGN) to create a MAC (message */
/*    authentication code) key token and a MACVER key token.         */
/*                                                                   */
/* 2) Calls the MAC_Generate verb (CSNBMGN) using the MAC key token  */
/*    from step 1 to generate a MAC on the supplied text string      */
/*    (INPUT_TEXT).                                                  */
/*                                                                   */
/* 3) Calls the MAC_Verify verb (CSNBMVR) to verify the MAC for the  */
/*    same text string, using the MACVER key token created in        */
/*    step 1.                                                        */
/*                                                                   */
/*********************************************************************/
#include <stdio.h>
#include <string.h>

#ifdef _AIX
  #include <csufincl.h>
#elif __WINDOWS__
  #include "csunincl.h"
#else
```

```
  #include "csulincl.h"     /* else linux */
#endif

/* Defines */
#define KEY_FORM            "OPOP"
#define KEY_LENGTH          "SINGLE  "
#define KEY_TYPE_1          "MAC     "
#define KEY_TYPE_2          "MACVER  "
#define INPUT_TEXT          "abcdefhgijklmn0987654321"
#define MAC_PROCESSING_RULE "X9.9-1  "
#define SEGMENT_FLAG        "ONLY    "
#define MAC_LENGTH          "HEX-9   "
#define MAC_BUFFER_LENGTH   10

void main()
{
  static long          return_code;
  static long          reason_code;
  static unsigned char key_form[4];
  static unsigned char key_length[8];
  static unsigned char mac_key_type[8];
  static unsigned char macver_key_type[8];
  static unsigned char kek_key_id_1[64];
  static unsigned char kek_key_id_2[64];
  static unsigned char mac_key_id[64];
  static unsigned char macver_key_id[64];
  static long          text_length;
  static unsigned char text[26];
  static long          rule_array_count;
  static unsigned char rule_array[3][8];       /* Max 3 rule array elements  */
  static unsigned char chaining_vector[18];
  static unsigned char mac_value[MAC_BUFFER_LENGTH];

/* Print a banner */
printf("Cryptographic Coprocessor Support Program example program.\n");

/* Set up initial values for Key_Generate call */
return_code = 0;
reason_code = 0;
memcpy (key_form,        KEY_FORM, 4);       /* OPOP key pair               */
memcpy (key_length,      KEY_LENGTH, 8);     /* Single-length keys          */
memcpy (mac_key_type,    KEY_TYPE_1, 8);     /* 1st token, MAC key type     */
memcpy (macver_key_type, KEY_TYPE_2, 8);     /* 2nd token, MACVER key type  */
memset (kek_key_id_1,  0x00, sizeof(kek_key_id_1));    /* 1st KEK not used  */
memset (kek_key_id_2,  0x00, sizeof(kek_key_id_2));    /* 2nd KEK not used  */
memset (mac_key_id,    0x00, sizeof(mac_key_id));      /* Init 1st key token */
memset (macver_key_id, 0x00, sizeof(macver_key_id));   /* Init 2nd key token */

/* Generate a MAC/MACVER operational key pair */
CSNBKGN(&return_code,
        &reason_code,
        NULL,                                /* exit_data_length           */
        NULL,                                /* exit_data                  */
        key_form,
        key_length,
        mac_key_type,
        macver_key_type,
        kek_key_id_1,
        kek_key_id_2,
        mac_key_id,
        macver_key_id);

/* Check the return/reason codes. Terminate if there is an error.           */
if (return_code != 0 || reason_code != 0) {
  printf ("Key_Generate failed: ");             /* Print failing verb       */
  printf ("return_code = %ld, ",  return_code); /* Print return code        */
  printf ("reason_code = %ld.\n", reason_code); /* Print reason code        */
  return;
```

```
}
else
  printf ("Key_Generate successful.\n");

/* Set up initial values for MAC_Generate call */
return_code = 0;
reason_code = 0;
text_length = sizeof (INPUT_TEXT) - 1;          /* Length of MAC text     */
memcpy (text, INPUT_TEXT, text_length);         /* Define MAC input text  */
rule_array_count = 3;                           /* 3 rule array elements  */
memset (rule_array, ' ', sizeof(rule_array));   /* Clear rule array       */
memcpy (rule_array[0], MAC_PROCESSING_RULE, 8); /* 1st rule array element */
memcpy (rule_array[1], SEGMENT_FLAG,        8); /* 2nd rule array element */
memcpy (rule_array[2], MAC_LENGTH,          8); /* 3rd rule array element */
memset (chaining_vector, 0x00, 18);             /* Clear chaining vector  */
memset (mac_value, 0x00, sizeof(mac_value));    /* Clear MAC value        */

/* Generate a MAC based on input text */
CSNBMGN ( &return_code,
          &reason_code,
          NULL,                                 /* exit_data_length            */
          NULL,                                 /* exit_data                   */
          mac_key_id,                           /* Output from Key Generate    */
          &text_length,
          text,
          &rule_array_count,
          &rule_array[0][0],
          chaining_vector,
          mac_value);

/* Check the return/reason codes. Terminate if there is an error.             */
if (return_code != 0 || reason_code != 0) {
  printf ("MAC Generate Failed: ");             /* Print failing verb         */
  printf ("return_code = %ld, ",  return_code); /* Print return code          */
  printf ("reason_code = %ld.\n", reason_code); /* Print reason code          */
return;
}
else {
  printf ("MAC_Generate successful.\n");
  printf ("MAC_value = %s\n", mac_value);      /* Print MAC value (HEX-9)     */
}
/* Set up initial values for MAC_Verify call */
return_code = 0;
reason_code = 0;
rule_array_count = 1;                           /* 1 rule array element       */
memset (rule_array, ' ', sizeof(rule_array));/* Clear rule array           */
memcpy (rule_array[0], MAC_LENGTH, 8);          /* Rule array element         */
                                                /* (use default Ciphering     */
                                                /* Method and Segmenting      */
                                                /* Control)                   */
memset (chaining_vector, 0x00, 18);             /* Clear the chaining vector  */

/* Verify MAC value */
CSNBMVR (&return_code,
         &reason_code,
         NULL,                                  /* exit_data_length           */
         NULL,                                  /* exit_data                  */
         macver_key_id,                         /* Output from Key_Generate   */
         &text_length,                          /* Same as for MAC_Generate   */
         text,                                  /* Same as for MAC_Generate   */
         &rule_array_count,
         &rule_array[0][0],
         chaining_vector,
         mac_value);                            /* Output from MAC_Generate   */

/* Check the return/reason codes. Terminate if there is an error.          */
if (return_code != 0 || reason_code != 0) {
  printf ("MAC_Verify failed: ");               /* Print failing verb      */
```

```
  printf ("return_code = %ld, ", return_code);  /* Print return code       */
  printf ("reason_code = %ld.\n", reason_code); /* Print reason code       */
  return;
}
else                                          /* No error occurred         */
  printf ("MAC_Verify successful.\n");
}
```

Sample program in Java

Before running this program, review the information about the JNI interface.

You can find this information in "Building Java applications using the CCA JNI" on page 26.

This sample code consists of a Java program named **mac.java**. For reference, a copy of the sample routine is shown in Figure 45 on page 982. Another sample program named RNG.java is included with the distribution at the same location, but is not copied here because it is a very simple JNI reference exercise to call the Random Number Generate verb.

The default distribution location of the sample code is:

SUSE Linux
/opt/IBM/CCA/samples

Red Hat Linux
/opt/IBM/CCA/samples

Invoke the following command from the directory that contains the sample source code to compile the program:

```
javac -classpath /opt/IBM/CCA/cnm/HIKM.zip mac.java
```

Note:

1. The classpath option points to the **HIKM.zip** file because the **hikmNativeInteger** class is in this file.
2. The path shown for the **HIKM.zip** file is the default distribution location of that file.

When it is compiled, you can run the sample Java program from the directory that contains the compiled output, with these commands.

For a Red Hat Linux system:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib64
/opt/ibm/java-i386-60/jre/bin/java -classpath /opt/IBM/CCA/cnm/HIKM.zip:. mac
```

For a SUSE Linux system:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib64
java -classpath /opt/IBM/CCA/cnm/HIKM.zip:. mac
```

Note:

1. The path shown for the **HIKM.zip** file is the default distribution location of that file.
2. The **libcsulcca.so** library for Linux also contains the C support for the CCA Java Native Interface (JNI).

Figure 45. Syntax, sample routine in Java

```
/*********************************************************************/
/*                                                                   */
/* Module Name: mac.java                                             */
/*                                                                   */
/* DESCRIPTIVE NAME: Cryptographic Coprocessor Support Program       */
/*                   JNI example code                                */
/*                                                                   */
/*-------------------------------------------------------------------*/
/*                                                                   */
/* Licensed Materials - Property of IBM                              */
/*                                                                   */
/* Copyright IBM Corp. 2010 All Rights Reserved                      */
/*                                                                   */
/* US Government Users Restricted Rights - Use duplication or        */
/* disclosure restricted by GSA ADP Schedule Contract with IBM Corp. */
/*                                                                   */
/*-------------------------------------------------------------------*/
/*                                                                   */
/*        NOTICE TO USERS OF THE SOURCE CODE EXAMPLES                */
/*                                                                   */
/* The source code examples provided by IBM are only intended to     */
/* assist in the development of a working software program. The       */
/* source code examples do not function as written: additional       */
/* code is required. In addition, the source code examples may       */
/* not compile and/or bind successfully as written.                  */
/*                                                                   */
/* International Business Machines Corporation provides the source   */
/* code examples, both individually and as one or more groups,       */
/* "as is" without warranty of any kind, either expressed or         */
/* implied, including, but not limited to the implied warranties of  */
/* merchantability and fitness for a particular purpose. The entire  */
/* risk as to the quality and performance of the source code         */
/* examples, both individually and as one or more groups, is with    */
/* you. Should any part of the source code examples prove defective, */
/* you (and not IBM or an authorized dealer) assume the entire cost  */
/* of all necessary servicing, repair or correction.                 */
/*                                                                   */
/* IBM does not warrant that the contents of the source code         */
/* examples, whether individually or as one or more groups, will     */
/* meet your requirements or that the source code examples are       */
/* error-free.                                                       */
/*                                                                   */
/* IBM may make improvements and/or changes in the source code       */
/* examples at any time.                                             */
/*                                                                   */
/* Changes may be made periodically to the information in the        */
/* source code examples; these changes may be reported, for the      */
/* sample code included herein, in new editions of the examples.     */
/*                                                                   */
/* References in the source code examples to IBM products, programs, */
/* or services do not imply that IBM intends to make these           */
/* available in all countries in which IBM operates. Any reference   */
/* to the IBM licensed program in the source code examples is not    */
/* intended to state or imply that IBM's licensed program must be    */
/* used. Any functionally equivalent program may be used.            */
/*-------------------------------------------------------------------*/
/*                                                                   */
/* This example program:                                             */
/*                                                                   */
/* 1) Calls the Key_Generate verb (CSNBKGNJ) to create a MAC (message*/
/*    authentication code) key token and a MACVER key token.         */
/*                                                                   */
/* 2) Calls the MAC_Generate verb (CSNBMGNJ) using the MAC key token */
/*    from step 1 to generate a MAC on the supplied text string      */
```

```
/*    (INPUT_TEXT).                                                     */
/*                                                                      */
/* 3) Calls the MAC_Verify verb (CSNBMVRJ) to verify the MAC for the */
/*    same text string, using the MACVER key token created in        */
/*    step 1.                                                           */
/*                                                                      */
/************************************************************************/
import java.io.*;

public class mac
{
  static final String KEY_FORM            = "OPOP";
  static final String KEY_LENGTH          = "SINGLE  ";
  static final String KEY_TYPE_1          = "MAC     ";
  static final String KEY_TYPE_2          = "MACVER  ";
  static final String INPUT_TEXT          = "abcdefhgijklmnopqrstuvwx";
  static final String MAC_PROCESSING_RULE = "X9.9-1  ";
  static final String SEGMENT_FLAG        = "ONLY    ";
  static final String MAC_LENGTH          = "HEX-9   ";
  public static void main (String args[])
  {
    byte [] ByteExitData         = new byte [4];
    byte [] Byte_key_form        = new byte [4];
    byte [] Byte_key_length      = new byte [8];
    byte [] Byte_mac_key_type    = new byte [8];
    byte [] Byte_macver_key_type = new byte [8];
    byte [] Byte_mac_value       = new byte [10];
    byte [] Byte_chaining_vector = new byte [18];
    byte [] Byte_rule_array      = new byte [24];
    byte [] Byte_text            = new byte [26];
    byte [] Byte_kek_key_id_1    = new byte [64];
    byte [] Byte_kek_key_id_2    = new byte [64];
    byte [] Byte_mac_key_id      = new byte [64];
    byte [] Byte_macver_key_id   = new byte [64];

    try
    {
      //setup to pause on non-zero return/reason code
      //and require enter key to continue
      BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));

      hikmNativeInteger IntReturncode     = new hikmNativeInteger(0);
      hikmNativeInteger IntReasoncode     = new hikmNativeInteger(0);
      hikmNativeInteger IntExitDataLength = new hikmNativeInteger(0);

      /* Print beginning banner */
      System.out.println("\nCryptographic Coprocessor Support Program JAVA example program.\n");
/* Set up initial values for Key_Generate call */
      Byte_key_form        = new String(KEY_FORM).getBytes();
/* OPOP key pair              */
      Byte_key_length      = new String(KEY_LENGTH).getBytes();
/* Single-length keys         */
      Byte_mac_key_type    = new String(KEY_TYPE_1).getBytes();
/* 1st token, MAC key type    */
      Byte_macver_key_type = new String(KEY_TYPE_2).getBytes();
/* 2nd token, MACVER key type */

      /* Generate a MAC/MACVER operational key pair */
      new HIKM().CSNBKGNJ (IntReturncode,
                           IntReasoncode,
                           IntExitDataLength,
                           ByteExitData,
                           Byte_key_form,
                           Byte_key_length,
                           Byte_mac_key_type,
                           Byte_macver_key_type,
                           Byte_kek_key_id_1,
```

```
                          Byte_kek_key_id_2,
                          Byte_mac_key_id,
                          Byte_macver_key_id);

      if ( 0 != IntReturncode.getValue() || 0 != IntReasoncode.getValue() )
      {
        System.out.println ("\nKey Generate Failed");
/* Print failing verb.      */
        System.out.println ("Return_code = " + IntReturncode.getValue());
/* Print return code.       */
        System.out.println ("Reason_code = " + IntReasoncode.getValue());
/* Print reason code.       */
        System.out.println ("Press ENTER to continue...");
/* Print Pause message      */
        stdin.readLine();
      }
      else
      {
        System.out.println ("Key_Generate successful.");
      }

      /* Set up initial values for MAC_Generate call */
      IntReturncode     = new hikmNativeInteger(0);
      IntReasoncode     = new hikmNativeInteger(0);
      IntExitDataLength = new hikmNativeInteger(0);

      hikmNativeInteger Int_rule_array_count = new hikmNativeInteger(3);
      hikmNativeInteger Int_text_length      = new hikmNativeInteger(24);

      Byte_text          = new String (INPUT_TEXT).getBytes();
/*Define MAC input text */
      byte [] temp_array = new String (MAC_PROCESSING_RULE).getBytes();
/*1st rule array element*/
      System.arraycopy( temp_array, 0, Byte_rule_array, 0, temp_array.length);
/*1st rule array element*/

      temp_array = new String(SEGMENT_FLAG).getBytes();
/*2nd rule array element*/
      System.arraycopy( temp_array, 0, Byte_rule_array, 8, temp_array.length);
/*2nd rule array element*/
      temp_array = new String(MAC_LENGTH).getBytes();
/*3rd rule array element*/

      System.arraycopy( temp_array, 0, Byte_rule_array, 16, temp_array.length);
/*3rd rule array element*/

 /* Generate a MAC based on input text */
      new HIKM().CSNBMGNJ (IntReturncode,
                          IntReasoncode,
                          IntExitDataLength,
                          ByteExitData,
                          Byte_mac_key_id,
                          Int_text_length,
                          Byte_text,
                          Int_rule_array_count,
                          Byte_rule_array,
                          Byte_chaining_vector,
                          Byte_mac_value);    if ( 0 != IntReturncode.getValue() || 0 != IntReasoncode.getValue() )
      {
        System.out.println ("\nMAC Generate Failed");
/* Print failing verb.      */
        System.out.println ("Return_code = " + IntReturncode.getValue());
/* Print return code.       */
        System.out.println ("Reason_code = " + IntReasoncode.getValue());
/* Print reason code.       */
        System.out.println ("Press ENTER to continue...");
/* Print Pause message      */
```

```
        stdin.readLine();
      }
      else
      {
        System.out.println ("MAC_Generate successful.");
        System.out.println ("MAC_value = [" + new String(Byte_mac_value) + "]");
      }

      /* Set up initial values for MAC_Verify call */
      IntReturncode     = new hikmNativeInteger(0);
      IntReasoncode     = new hikmNativeInteger(0);
      IntExitDataLength = new hikmNativeInteger(0);

      Byte_rule_array  = new String (MAC_LENGTH).getBytes();
/* Rule array element         */
      Int_rule_array_count = new hikmNativeInteger(1);

      new HIKM().CSNBMVRJ (IntReturncode,
                           IntReasoncode,
                           IntExitDataLength,
                           ByteExitData,
                           Byte_macver_key_id,
                           Int_text_length,
                           Byte_text,
                           Int_rule_array_count,
                           Byte_rule_array,
                           Byte_chaining_vector,
                           Byte_mac_value);

      if ( 0 != IntReturncode.getValue() || 0 != IntReasoncode.getValue() )
      {
        System.out.println ("\nMAC_Verify Failed");
/* Print failing verb.      */
        System.out.println ("Return_code = " + IntReturncode.getValue());
/* Print return code.       */
        System.out.println ("Reason_code = " + IntReasoncode.getValue());
/* Print reason code.       */
        System.out.println ("Press ENTER to continue...");
/* Print Pause message      */
        stdin.readLine();
      }
      else
      {
        System.out.println ("MAC_Verify successful.");
      }
    }
    catch (Exception anException)
    {
      System.out.println(anException);
    }

    /* Print ending banner */
    System.out.println("\nCryptographic Coprocessor Support Program JAVA example program finished.\n");
  }//end main
}//end mac class
```

Chapter 24. Initial system set-up tips

Use these tips to help you set up your system for the first time.

The name of the CCA RPM is:

csulcca-<*version*>.<*release_level*>.<*maintenance_level*>-<*fix_level*>.s390x.rpm

where:

version
represents the major version, typically the same for an adapter life

release_level
represents the function available, updated to add function

maintenance_level
represents the level of increment on top of the function release for small functional changes

fix_level
represents a level of fix applied, typically incremented during internal package development.

Example:

```
csulcca-4.2.10-09.s390x.rpm
```

In order to use the full set of CCA functions, a CCA feature is required. This feature must have a CCA code level appropriate for the desired function for RPM platform target s390x. A limited set of CCA Release 4.x functions can be used with a CEX2C feature, by using the CCA 4.x rpm. Note that CEX2C support is not maintained for CCA 5.0 rpm.

Consult the `README.linz` file in the `/opt/IBM/CCA/doc/` directory for this information:

- Release-specific information, if there is any
- Pointers to helpful tools

Note: If the version and release level of the feature is lower than the version and release level of the RPM, then only the functions supported by the feature are supported by the RPM.

Installing and loading the cryptographic device driver

The cryptographic device driver is included in the regular kernel package shipped with your Linux distribution.

In in earlier Linux distributions, the cryptographic device drive is shipped as a single module called **z90crypt**. In more recent distributions, the cryptographic device driver is shipped as set of modules with the **ap** module being the main module that triggers loading all required sub-modules. There is,however, an alias name **z90crypt** that links to the **ap** main module.

Use the **lsmod** command to find out if either the **z90crypt** or the **ap** module is already loaded.

If required, use the **modprobe** command to load the **z90crypt** or **ap** module. When loading the **z90crypt** or **ap** module, you can use the following optional module parameters:

domain=
specifies a particular cryptographic domain. By default, the device driver attempts to use the domain with the maximum number of devices. To use all CCA Release 4.2.x functions, the domain must include at least one CEX3C or above feature. To use CCA 5.x functions, the domain must include at least one CEX5C or above feature.

After loading the device driver, use the **lszcrypt** command with the `-b` option to confirm that the correct domain is used. If your distribution does not include this command, see the version of *Device Drivers, Features, and Commands* that applies to your distribution about how to use the sysfs interface to find out the domain. This publication also provides more information about loading and configuring the cryptographic device driver.

To change the domain, you must unload the **z90crypt** or **ap** module (see "Unloading the cryptographic device driver") and reload it.

poll_thread=
enables the polling thread for instances of Linux on z/VM and for Linux instances that run in LPAR mode on an IBM mainframe system earlier than System z10.

For Linux instances that run in LPAR mode on a System z10 or later mainframe, this setting is ignored and AP interrupts are used instead.

For more information about these module parameters, the polling thread, and AP interrupts, see the version of *Device Drivers, Features, and Commands* that applies to your distribution.

Unloading the cryptographic device driver

You might need to unload the cryptographic device driver, for example, to change the domain setting.

Unloading the device driver is complicated slightly because the catcher.exe daemon is always running, to be ready to receive TKE requests. To unload the device driver, for example, in preparation for a reload, you must stop the catcher.exe daemon. This can be done with the service management script `/etc/init.d/CSUTKEcat` using the **stop** argument, or by using the **ps** command to find the PID for the daemon, and then using the `kill -9 <PID>` command to kill it.

After reloading the device driver, you can restart the catcher.exe daemon (restarting TKE access) using the **/etc/init.d/CSUTKEcat start** command. See "Files in the RPM" on page 992 for more details.

Confirming your cryptographic devices

Use the **lszcrypt** command with the **-V** option to display a list of available cryptographic devices and their online status.

Example 1)
```
lszcrypt -V
card00: CEX4A online
card01: CEX4A online
card02: CEX4C offline
card03: CEX4C online
```

Example 2)
```
lszcrypt -V
card00: CEX5C online
card01: CEX5C online
```

In the second example, you see two CEX5C devices online on an IBM z13 machine, for example.

In the first example, device `card02` is offline. You can use the **chzcrypt** command to set a cryptographic device online. For example, to set a cryptographic device `card02` online, issue:

```
chzcrypt -e 2
```

In the command, `2` is the decimal representation of the hexadecimal `02` index in the device name, `card02`.

For more details about the **lszcrypt** and **chzcrypt** commands, see the man pages. If your distribution does not include these commands, see the version of *Device Drivers, Features, and Commands* that applies to your distribution about how to use the sysfs interface to find out this information and to set devices online.

Checking the adapter settings

Use the **lszcrypt** command with the **-b** option to display information about the settings for your cryptographic adapters.

```
lszcrypt -b
ap_domain=8
ap_interrupts are enabled
config_time=30 (seconds)
poll_thread is disabled
poll_timeout=250000 (nanoseconds)
```

As seen in the example, the command output shows:

- The domain used

  Note: To change the domain you must unload the z90crypt module and reload it with the appropriate `domain=` parameter.
- Whether AP interrupts are used
- Whether the polling thread is used and, if so, at which frequency

  Note: Polling threads cannot be enabled if AP interrupts are available.

If your distribution does not include the **lszcrypt** command, see the version of *Device Drivers, Features, and Commands* that applies to your distribution about how use the sysfs interface to retrieve this information.

Performance tuning

If no AP interrupts are available to your Linux instance, you can use the settings for the polling thread and the high resolution polling timer to tune the performance of your cryptographic adapters.

See the version of *Device Drivers, Features, and Commands* that applies to your distribution for more information about these settings.

Running secure key under a z/VM guest

In order to use the CEX*C feature under z/VM, you need to apply specific APAR fixes.

These fixes are described in “Hardware requirements” on page xxv.

Also, to get secure key running under a z/VM guest, a directory control statement (CRYPTO APDED) for a given z/VM guest needs to be used. This requires that the AP's with this domain are owned by the LPAR. There is no virtualization done by z/VM.

For secure key, z/VM does not virtualize the AP's. The AP's need to be dedicated, which is done by the user statement:

```
CRYPTO DOMAIN 12 APDED 5 7
```

This statement dedicates AP's 5 and 7 for domain 12 to one Linux guest.

Note: Shared crypto adapters, as defined with the z/VM user directory statement CRYPTO APVIRT, cannot be used for secure key cryptographic operations. Because dedicated and shared cryptographic adapters cannot be mixed in a z/VM guest virtual machine, additional crypto adapters for use with clear key cryptography, coprocessors or accelerators, must be defined as dedicated adapters.

Chapter 25. CCA installation instructions

Use these instructions when you install, configure, or uninstall CCA release 4.0.0 and later.

Before you begin

In the installation instructions, specific terms are used to describe CCA RPMs, referring to several different versions.

CCA 5.x
CCA 5.0.0 and subsequent versions.

CCA 4.x
CCA 4.0.0, CCA 4.1.0, CCA 4.2.0, and subsequent versions.

CCA 3.x
Any CCA version 3 release.

Before you begin the CCA installation, review these points:

- If you are **upgrading** from a prior installation, be sure to review "Default installation directory."
- Ensure that you are using supported hardware. See "Hardware requirements" on page xxv.
- Ensure that your Linux distribution has the cryptographic device driver support. For details, see "Installing and loading the cryptographic device driver" on page 987.
- If you are going to use the Java Native Interface (JNI), see "Building Java applications using the CCA JNI" on page 26 for supported Java levels and installation instructions.

Default installation directory

The CCA RPM chooses a default installation directory.

In the past a part of the path name was the name of the specific CCA coprocessor being supported, such as *CEX3C*. As of CEX4C support (introduced with RPM level 'csulcca-4.2.10-*'), multiple CCA coprocessors are supported from the same install tree, and it is no longer appropriate to use the CCA coprocessor name in the path. The new default install path is:

```
/opt/IBM/CCA/
```

A soft-link is added to ease migration, such that references to `/opt/IBM/CEX3C/` will still work.

If you are upgrading, note that the 4.2.10 RPM and later RPMs will copy your key storage files from the old default location to the new default location. The old directory will be kept and renamed to `/opt/IBM/CEX3C-old/`.

Download and install the RPM

The CCA RPM contains files, samples, and groups.

About this task

To download the RPM, complete these steps:

Procedure

1. Point your Web browser at this location:
   `http://www.ibm.com/security/cryptocards/`
2. Click the appropriate link from the **Cryptocards** box for your cryptographic coprocessor / HSM.
3. Click the **Software download** link on the appearing sub-menu.
4. On the next page, find the heading **IBM z Systems servers running Linux**.
5. In the paragraph underneath this heading will be the links to download:
   - the README file for the RPM
   - the RPM itself that installs the host code.

Files in the RPM

These files are included in the RPM.

/etc/profile.d/csulcca.sh
Environment variables are created in this file. Customers should read this file for up-to-date information. The key storage environment variables are added here. See "Environment variables for the key storage file" on page 402.

/etc/profile.d/csulcca.csh
Environment variables are created in this file. Customers should read this file for up-to-date information. The key storage environment variables are added here. See "Environment variables for the key storage file" on page 402.

Note: /etc/profile.d/csulcca.sh and **/etc/profile.d/csulcca.csh** are exactly the same in what they do, but have syntax differences. Only one of these two files is used, depending on the configuration of the particular user running an application.

/etc/init.d/CSUTKEcat
System initialization script that automatically starts the `catcher.exe` daemon when Linux starts. You can also use this script to start or stop `catcher.exe` from the command line. To start `catcher.exe`, issue:

```
/etc/init.d/CSUTKEcat start
```

To stop `catcher.exe`, issue:

```
/etc/init.d/CSUTKEcat stop
```

/etc/rc.d/rc2.d/S14CSUTKEcat
Link to: **/etc/init.d/CSUTKEcat**

/etc/rc.d/rc3.d/S14CSUTKEcat
Link to: **/etc/init.d/CSUTKEcat**

/etc/rc.d/rc5.d/S14CSUTKEcat
Link to: **/etc/init.d/CSUTKEcat**

/opt/IBM/CCA/bin/TKECM.dat

/opt/IBM/CCA/bin/acpoints.dat

/opt/IBM/CCA/bin/catcher.exe
Daemon executable that is controlled by the `/etc/init.d/CSUTKEcat` script. This daemon listens on TCP port 50003 for requests from the Trusted Key Entry workstation (TKE) for secure commands to administer any adapters configured as available to the system. See the TKE documentation for usage and capabilities. The daemon depends on libcsulcca.so and on the **z90crypt** or **ap** device driver. While the daemon is running, it has an open file handle to the `/dev/z90crypt` device node maintained by the cryptographic module. Therefore, **catcher.exe** as a running process affects whether you can load or unload **z90crypt** or **ap** respectively. Use `/etc/init.d/CSUTKEcat` to start**catcher.exe** or to stop it if you want to unload **z90crypt** or **ap** respectively. Remember that TKE secure administration requires a running catcher.exe process.

/opt/IBM/CCA/bin/panel.exe
Utility: run with no arguments for help

/opt/IBM/CCA/bin/ivp.e
Utility: run with no arguments for install verification

/opt/IBM/CCA/cnm/CNM.jar

/opt/IBM/CCA/cnm/CNMMK.jar

/opt/IBM/CCA/doc/README.linz

/opt/IBM/CCA/doc/license.txt

/opt/IBM/CCA/include/csulincl.h

/opt/IBM/CCA/doc/hikmNativeInteger.html

/usr/lib64/libcsulcca.so
This file is a link to the most recent library file, with name **libcsulcca.so.V.R.M** where: V = version, R = release, and M = maintenance level.

/usr/lib64/libcsulccamk.so
This file is a link to the most recent library file, with name **libcsulccamk.so.V.R.M** where: V = version, R = release, and M = maintenance level.

/opt/IBM/CCA/keys/README.keys

Samples in the RPM

These samples are included in the RPM.

/opt/IBM/CCA/samples/mac.c
C code sample

/opt/IBM/CCA/samples/makefile.lnx
Used to build mac.c

/opt/IBM/CCA/samples/mac.java
Java code sample

/opt/IBM/CCA/samples/RNGpk.java
Java code sample

Groups in the RPM

These groups are created for the purpose of loading master keys.

They are added during RPM installation as updates to **/etc/groups**. See Table 272 on page 997.

- cca_admin
- cca_clrmk
- cca_lfmkp
- cca_cmkp
- cca_setmk

Install and configure the RPM

Use the following steps to install and configure the CCA RPM.

Procedure

1. Copy the RPM to the host where it will be installed. For example, `/root` on your host image.
2. Login to the host as root. Change to the directory where the RPM is located by issuing these commands:

   ```
   <login to host>
   cd /root/
   ```

3. Install the RPM by issuing the following command:

   ```
   rpm -i <rpm name>
   ```

   Note:

   a. For compatibility reasons a softlink will be created from `/opt/IBM/CCA` to `/opt/IBM/CEX3C`.

   b. If this is an upgrade, you can use this command: `rpm -Uvh`.

   c. If you are installing the RPM on a SUSE distribution of Linux, it is possible that you might receive the following warning messages because of an unsupported **groupadd** option.

   ```
   groupadd: You are using an undocumented option (-f)!
   groupadd: You are using an undocumented option (-f)!
   groupadd: You are using an undocumented option (-f)!
   groupadd: You are using an undocumented option (-f)!
   groupadd: You are using an undocumented option (-f)!
   ```

   No action on your part is needed. The installation proceeds with another call if this happens.

4. Reboot the host by issuing the following command: **shutdown -r now** This is necessary because of the defaults added to `/etc/profile.d/csulcca.sh` and `/etc/profile.d/csulcca.csh` for using CCA must be propagated to all user login sessions.

   If all users that use the CCA logout and then login again, and if all applications that use CCA are re-started, then a reboot may be avoided. It is always recommended to reboot to ensure for new users or new system administrators that the updated profiles are actually in force after the install procedure is completed.

5. Login to the host as root. Change to the directory where the RPM binaries are installed by issuing the following command:

   ```
   <login as root to host>
    cd /opt/IBM/CCA/bin/
   ```

6. Verify that at least one card is present and active:

   a. Ensure that the device driver is loaded by issuing the following command:

   ```
   lsmod
   ```

You should see the module **z90crypt** or the **ap** module loaded. If it is not loaded, verify the contents of the kernel modules directory by issuing the following command:

```
ls /lib/modules/<Linux kernel version>/kernel/drivers/s390/crypto/
```

You should see **z90crypt.ko** or **ap.ko**. If one of them is present, then load it with the appropriate command:

```
modprobe z90crypt
modprobe ap
```

Note:

1) This works if you have only one domain assigned to the LPAR (or z/VM guest using a dedicated CEX*C card). If more than one domain is available, you need the domain parameter in the `modprobe` command to assign a domain. If no domain parameter is specified, the Linux kernel always chooses the lowest available defined domain.
2) SUSE Linux uses its own start script, named **rcz90crypt**, to do all the work. Settings can be specified using YaST.

Note: If you do not see any of these kernel modules, or if there are any errors reported from the call to `modprobe`, contact IBM Service.

b. When you are sure that the device driver is loaded, run one of the RPM-installed utilities to verify accessibility by running the following commands:

1)

```
/opt/IBM/CCA/bin/ivp.e
```

This command health checks all active cards.

2)

```
/opt/IBM/CCA/bin/panel.exe -x
```

This command shows the serial numbers and master key register states of all active cards running CCA that are visible to this Linux host. The total number of active cards and any errors is also reported.

Note:

1) To be able to use `/opt/IBM/CCA/panel.exe` the user must be either root or a member of the **cca_admin** group (the owner of `/usr/lib64/libcsulccamk.so`).
2) If there is not at least one active card at this point, double check earlier steps and, if necessary, involve IBM service because the rest of the setup is designed around having active cards.
3) Unload of the device driver requires killing the **catcher.exe** program, and then restarting it when the driver is reloaded. See the note in "Installing and loading the cryptographic device driver" on page 987 for specific instructions.

7. **Master key load** - This procedure is for using the Linux on z Systems native API or the utility (panel.exe) to load the master keys for the active cards.

   If you want to use TKE instead, refer to a TKE manual, such as the IBM Redbooks® publication *Exploiting S/390 Hardware Cryptography with Trusted Key Entry* for proper use. After completing this step using the TKE procedure, go to Step 8 on page 998.

   a. Setup the groups for the users who will be loading the master keys to the cards. Each part of the load process is owned by a different Linux group created by the RPM install procedure, and verified in the host library

implementing the API allowing master key processing. To complete a specific step the user must have membership in the proper group. There are a couple ways to change group membership depending on your Linux distribution. A third option is to create the users specifically for these roles.

If a user does not have the proper group membership for a particular master key operation, the error X'0008005a' is returned and an error message is printed to the system log.

Note: To be able to use */opt/IBM/CCA/panel.exe*, the user must be either root or a member of the **cca_admin** group (the owner of `/usr/lib64/libcsulccamk.so`).

1) Group membership for Red Hat based Linux distributions:
   a) Use the **groups** command to see a list of the user's current group membership:

```
groups <user name>
    ---output is
     <user name> : <grouplist>
     <grouplist> is a single-space separated list
```

   b) <grouplist> must be passed along with the new group to the **usermod** command as a comma-separated list, followed by the <user name>. For example, if you wanted to add **cca_lfmkp** membership to user named **admin**, you would use the following commands:

```
groups admin
    ---output:
     admin : admin bin daemon sys wheel
    usermod -G admin,bin,daemon,sys,wheel,cca_lfmkp admin
    ---output:
     [none if successful]
```

      Note: Ensure the user logs out and logs back in, otherwise the group membership in the active session will not be updated.
2) Group membership for SUSE-based Linux distributions:
   a) Use the **usermod** command to add membership for a specific group for a specific user. For example, if you wanted to add **cca_lfmkp** membership to user **admin**, you would use the following commands:

```
usermod -A cca_lfmkp admin
```

      Note: Ensure the user logs out and logs back in, otherwise the group membership in the active session will not be updated.
3) Create users for each role with correct group memberships (Same commands for Red Hat and SUSE):
   a) Create user **cca_user**, which will own default key storage by issuing the following commands:
      i. `useradd -g cca_admin -d /home/cca_user -m cca_user`

         This command creates the user with primary group **cca_admin** and a new home directory.
      ii. `passwd cca_user`

         This command sets the new user's password.
   b) Create user **cca_lfmkp** by issuing the following commands:
      i. `useradd -g cca_admin -d /home/cca_lfmkp -G cca_admin,cca_lfmkp -m cca_lfmkp`

This command creates the user with primary group **cca_admin**, secondary group **cca_lfmkp**, and a new home directory.

ii. `passwd cca_lfmkp`

This command sets the new user's password.

c) Create user **cca_cmkp** by issuing the following commands:

i. `useradd -g cca_admin -d /home/cca_cmkp -G cca_admin,cca_cmkp -m cca_cmkp`

This command creates the user with primary group **cca_admin**, secondary group **cca_cmkp**, and a new home directory.

ii. `passwd cca_cmkp`

This command sets the new user's password.

d) Create user **cca_clrmk** by issuing the following commands:

i. `useradd -g cca_admin -d /home/cca_clrmk -G cca_admin,cca_clrmk -m cca_clrmk`

This command creates the user with primary group **cca_admin**, secondary group **cca_clrmk**, and a new home directory.

ii. `passwd cca_clrmk`

This command sets the new user's password.

e) Create user **cca_setmk** by issuing the following commands:

i. `useradd -g cca_admin -d /home/cca_setmk -G cca_admin,cca_setmk -m cca_setmk`

This command creates the user with primary group **cca_admin**, secondary group **cca_setmk**, and a new home directory.

ii. `passwd cca_setmk`

This command sets the new user's password.

b. Add group membership privileges to users based on their required function.

Table 272. CCA groups.

Group Name	Description
cca_admin	All users who will run part of the master key load process must be in this group because the library itself is owned by **root.cca_admin**, with no permissions for 'world' as a protective measure. Reasons for this separate group also include allowing one owner of `/usr/lib64/libcsulccamk.so`, and allowing use of **panel.exe** without allowing any of the master key processing calls.
cca_lfmkp	The user to LOAD the first key part must be in this group.
cca_cmkp	The users to LOAD the middle and last key parts must be in this group.
cca_clrmk	The new master-key register can be CLEARed using the same Master Key Process call in case a mistake was made entering a key part (use the key verification patterns to check for this). To perform the clear, the user must be a member of this group.
cca_setmk	The user to call SET after the last key part has been successfully loaded must be a member of this Linux group.

c. Load FIRST, MIDDLE (optional), and LAST key parts for the AES, SYM, ASYM, and APKA master keys and then call SET for each master key. This step can be done using the **panel.exe** utility provided or by writing your own application to call the Master Key Process (CSNBMKP) verb directly. The application must link with the correct library (installed to

/usr/lib64/libcsulccamk.so by the RPM), and must be executed at each step by a user with the appropriate group memberships. The utility supports scripted as well as prompt-driven access.

Repeat this step for each configured adapter. See “Changing the master key for two or more adapters that have the same master key, with shared CCA key storage” on page 406.

For details about **panel.exe**, see “The panel.exe utility” on page 1003.

See “Master Key Process (CSNBMKP)” on page 122 about parity requirements for master key parts.

Note: Loading master key parts modifies state information inside the card. For example you cannot load a 'FIRST' master key part twice in a row without clearing the new master-key register in between attempts. The same goes for setting the 'LAST' register. Any number of 'MIDDLE' parts can be loaded - with each call changing the contents of the new master-key register. Similarly a 'SET' operation changes the state of the 'new' register back to 'empty', while updating the 'current' register.

8. **Key storage initialization** - To perform this step, see “Using panel.exe for key storage initialization” on page 1006.
9. **Key storage re-encipher when changing the master key** - To perform this step, see “Using panel.exe for key storage re-encipher when changing the master key” on page 1007.
10. If you are going to be using Central Processor Assist for Cryptographic Functions (CPACF), it must be configured. See “CPACF support” on page 12.

Uninstall the RPM

Use the following steps to uninstall the CCA RPM.

Procedure

1. Uninstall any RPMs that depend on the CCA RPM. If you try to uninstall the CCA RPM and dependent RPMs are still installed, the uninstall RPM command will fail and list the names of dependent RPMs. Therefore, you can skip to Step 2 and come back to this step if Step 2 fails for that reason.
2. Uninstall the CCA RPM.
   a. Login as root. You have to be root to uninstall the RPM.
   b. You have to use the full name. You can find the name by issuing the following command:

      ```
      rpm -qa | grep csulcca
      ```

   c. Uninstall the RPM with the following command:

      ```
      rpm -e <rpm name>
      ```

   Note:

   a. Groups are no longer deleted during the uninstall of CCA RPM. If you created any users with one of the groups created by the RPM install as their primary (note that the RPM install does NOT create any users, just groups), you can delete those users/groups yourself after uninstall, or remove such users before the uninstall of the RPM. This will remove any potential security holes.

b. Card master keys (and other state information) are untouched by the host-side uninstall of the RPM.
c. Key storage files are not deleted by the uninstall. All default and non-default key storage files are left as is. If you reinstall or install an upgraded package and load any new cards with the same master keys, you still can use your old key storage (old cards still have the old keys, see step 7b on page 997 of "Install and configure the RPM" on page 994).

Chapter 26. Coexistence of CEX5C and previous CEX*C features

While CEX5S is supported only on processors starting with IBM z13, and CEX4S and CEX3C are not supported on a z13 machine, the CCA 5.0 rpm is useful on IBM zEnterprise EC12 and other legacy systems. The CCA 5.0 rpm specifically adds support for the new CCA 4.x firmware on these platforms. This information unit discusses co-existence considerations for the CCA 5.0 rpm.

- Use CCA 5.0 to update existing 4.*z installations or to install it to a fresh client. Running CCA 5.0 in parallel with a prior 4.*z rpm is not specifically supported, though you might achieve such a hybrid installation by manipulating the RPM.
- Running CCA 5.0 in parallel with the CEX2C focused prior release rpm and a CEX3C adapter configured to the same system image is not tested. Issues are the same as concurrency issues with the 4.0.0z release running in parallel to 3.28z (see "Concurrent installations").
- The latest CCA RPM supersedes and replaces earlier RPMs if installed in the default manner. It is not recommended to manually alter the installation in order to run newer versions of CCA RPMs in parallel with older versions. This configuration is not supported.

Concurrent installations

These are background considerations for installation of the CEX5C RPM alongside with earlier CEX*C RPMs.

1. The **libcsulcca.so** and **libcsulsapi.so** libraries for CCA 5.0, CCA 4.x, and CCA 3.x have many symbols with the same names. An application cannot deterministically link with both libraries. The first library in the link statement is what will be used for all symbols that can be resolved there, after that the second library will be examined. At this point, either the linker will not allow link to continue, by throwing an error on the duplicate symbols, or will produce a hybrid-linked application. Either case will give the user the wrong answer.

   A new or updated library cannot itself resolve this kind of conflict because:

   - There is no way to have a default set of symbols or card support in an updated host library. The link operation is a fundamental step in building the customer application and outside the control of the library or library installation process.
   - One way to resolve name collisions is to change all of the function names in the new library. However, this would have greatly impacted the customer's ability to port applications forward, and this option was rejected.
2. The key storage environment variables in the default user profile (`/etc/profile.d/csulcca.sh` and `/etc/profile.d/csulcca.csh`) are changed at installation time to point to the `/opt/IBM/CCA/keys/` path. Softlinks are added to older default installation paths to assist migration, but double-check that all of your key storage is accessible.
3. See "Interaction between the *default card* and use of Protected Key CPACF" on page 17 for a concurrency and CPACF.

CEX3C or later and CEX2C co-installation toleration

The CCA 5.x RPMs do not support accessing the CEX2C feature. There is no added function to CEX2C features or co-installation feasibility to justify keeping it. Refer to the most recent CCA 4.x RPM for CEX2C support.

The CCA 4.x RPMs all support accessing the CEX2C feature as well as the CEX3C or later feature, with the first CEX3C or later feature becoming the 'default' adapter.

This can be changed using environment variables. See "Environment variables that affect CPACF usage" on page 12. Using CCA 4.x is your best option for accessing a CEX2C feature as well as a CEX3C or later feature going forward, even in a CEX2C-only installation.

CEX5C information

Since the TKE catcher daemon listens on 1 particular numbered port, it is impossible to allow a legacy catcher daemon to co-exist with a newer catcher daemon. This must be the new TKE catcher to support the newest adapters.

There are the following impacts:

- The newest TKE catcher must support all in-service CPRB types:
  - T3 (CCA 3.*), ⌂ removed from service for CCA 5.0 rpm. Customers will never have CEX2C adapters on the same system as CEX5C adapters, and function at 4.2.10 is considered sufficient host library support on z systems where a CEX2C may be installed with a CEX3C.
  - T5 (CCA 4.*)
  - T6 (CCA 5.*)
- libcsulcca.so supports access to both types of adapters.

CEX2C - CEX4C information

The TKE catcher daemon listens on a single port for management communication. This port number has not changed for the CEX3C or later release. Therefore, the new daemon supports TKE management communication to both the CEX2C and CEX3C or later adapters. Special steps are taken in the install/uninstall and daemon management for the CEX3C or later release to ensure that the new daemon is running when it is available.

TKE versions for CEX*Cs

- TKE V4-V5 workstations only see CEX2Cs.
- A TKE V6.0 or higher workstation is required to see supported CEX3Cs or CEX4Cs: If you are running in toleration mode, and Linux reports CEX4Cs as CEX3Cs, then TKE 6.0 is able to manage them as CEX3Cs.
- TKE V7.2: CEX4Cs are reported by TKE7.2 as CEX3Cs. CEX4Cs are only supported by TKE 7.2 or later if the Linux driver reports them as CEX4s.
- A TKE V8.0 or higher workstation is required to see CEX5Cs. CEX4C, CEX3C, CEX2C are also supported by this TKE.

For more information about the TKE workstation, see *z/OS Cryptographic Services ICSF: Trusted Key Entry PCIX Workstation User's Guide*.

Chapter 27. Utilities

You can use a utility to verify an installation. An additional utility is available to administer cryptographic coprocessors in setups without a TKE.

These two utilities are called *ivp.e* and *panel.exe*.

ivp.e This is an easy-to-use utility used to verify an installation.

It calls the Cryptographic Facility Query verb for all available adapters to report the firmware and configuration details.

This utility is installed by default to the following path in the Linux system:

```
/opt/IBM/CCA/bin/ivp.e
```

You can invoke this utility without any arguments. It presents information for all available CEX*C features on the system.

panel.exe

This utility provides a Linux native mechanism for administering and initializing certain characteristics of active cryptographic coprocessors. It is intended as a basic administration tool for Linux-only IBM z Systems configurations, where a Trusted Key Entry (TKE) solution is not available.

For mixed z/OS and Linux configurations, it is recommended that administration be accomplished using the z/OS TSO panels as described in the *z/OS ICSF Administrator's Guide*. The utility is installed by the Linux on z Systems Cryptographic Coprocessor install package or RPM to this path in the Linux system:

```
/opt/IBM/CCA/bin/panel.exe
```

There is more detailed information in "The panel.exe utility."

The panel.exe utility

The panel.exe utility is installed by the Linux on z Systems Cryptographic Coprocessor install package or RPM.

It is installed to this path in the Linux system:

```
/opt/IBM/CCA/bin/panel.exe
```

The panel.exe utility numbers cards from card0 to card63, while verbs such as Cryptographic Resource Allocate number cards from CRP01 to CRP64, and therefore card0 corresponds to CRP01, card1 corresponds to CRP02, and so forth.

panel.exe syntax

Precise usage information can be obtained by running the panel.exe utility with no arguments on the Linux shell command line.

This is an example output:

```
Panel usage ([-k,-a <num>,-o,-g][-?,-h,-x,-m,-l,-s,-c,-q,-t,-i,-r,-p,-n,-v,-y[l|c|s]]
>> [CC] Arg >> arg must precede non-[CC] args
        [CC] -k:       Can TKE administer a card?
        [CC] -a <num>: use non-default card
```

```
                        <num> is the card number [0 - 63]
                [CC] -o:       Disable output to stdout:
                [CC] -g <level>: Set the log level:
                        <level> can be NONE, TRANSACTIONS, NONZERO,
                                       ALL, DEBUG, and FUNCTIONS

        >> non-[CC] Args >>: (all are mutually exclusive)


        ---BASIC ADMIN---

                -? , -h:       Usage

                     -v:       Print CCA Host Library version
                     -x:       List crypto resources (and basic status)
                     -m:       List CPACF (local CPU crypto) resources

                     -y:       Get SYSLOG or CCALOG info
                               -y [no options] :: sizes of all card logs and card logging level
                               -yl [4|8|12]    :: sets 4, 8 or 12 as card logging level
                                                  4,8,12 are CCA return code trigger levels
                               -yc             :: dump card CCA log to stdout
                               -ys [0|1|2|3|4] :: dump card SYS log to stdout
                                                  0 = current/default, 4 = oldest
                               -a <num> -y [options]:: perform on non-default card

        ---MASTER KEY (MK)---

                To LOAD  a Master Key (MK) PART:
                          -l (for interactive)
                        OR====>
                          -l -t [A|S|E|P] -p [F|M|L] KEYPART
                          where: -t [A|S|E|P] is which MK: A=ASYM, S=SYM, E=AES P=APKA
                          where: -p [F|M|L] is the part: F=FIRST, M=MIDDLE, L=LAST
                          where: KEYPART is string in hex 2* size of key
                                   (recall: 2 text chars = 1 binary Byte)
                To SET   a Master Key:
                          -s (for interactive)
                        OR====>
                          -s -t [A|S|E|P]
                          where: -t [A|S|E|P] is which MK: A=ASYM, S=SYM, E=AES P=APKA
                To CLEAR a Master Key 'New' Register:
                          -c (for interactive)
                        OR====>
                          -c -t [A|S|E|P]
                          where: -t [A|S|E|P] is which MK: A=ASYM, S=SYM, E=AES P=APKA
                To QUERY a Master Key Verification Pattern:
                          -q (for interactive)
                        OR====>
                          -q -t [A|S|E|P] -r [N|C|O]
                          where: -t [A|S|E|P] is which MK: A=ASYM, S=SYM, E=AES P=APKA
                          where: -r [N|C|O] is which register: N=NEW, C=CURRENT, O=OLD
        ---KEY STORAGE---

                To INIT     a KEY STORAGE file:
                        -t <type> -i
                To REENCipher KEY STORAGE:
                        -t <type> -r
                To LIST     a KEY STORAGE:
                        -t <type> -p
                where:
                   <type> can be AES , DES , PKA
                   **environment variables for key storage files**
                            --the key DISK STORAGE   (DS) variables are:

                                 CSUDESDS, CSUAESDS, CSUPKADS
```

```
                                        **the path must exist and specify a filename. The file
                                          will be created on key storage initialization.
                                --the key LIST DIRECTORY (LD) variables are:

                                      CSUDESLD, CSUAESLD, CSUPKALD

                                        **the full path must exist.


---RETAINED KEYS---

        To LIST     RETAINED KEYS (this domain ONLY):
                -n
```

Note: For security reasons, only a root user (real user id equal to '0') is allowed to use panel.exe to load master key parts or to clear previously loaded master key parts. This is enforced at the shared library level in the implementation of the Master Key Process verb, not in the utility itself. Additionally, only the user who created a set of key storage files or the 'root' user will be able to take actions with respect to those key storage files, based on Linux file system permissions.

panel.exe functions

Different uses for the panel.exe utility.

The panel.exe utility can be used to:

- Determine if a TKE is currently able to administer a specific active coprocessor
- List the labels and key types for all the keys in a designated key storage file.
- List the labels for all of the retained keys (RSA private keys stored in the adapter) in the current domain of the CEX*C.
- List the coprocessors currently active in the Linux system and their master key status
- Load master key parts to the coprocessor
- Set a master key that was loaded to the coprocessor. Note that panel.exe, is not designed to change the master keys for all the cards in a group; this is a more sophisticated operation.
- Clear master key parts which were previously loaded to the coprocessor but not yet 'set' or confirmed (used for when a mistake in entering master key parts has been detected)
- List serial numbers and master key register states of all active cards running CCA that are visible to this Linux host. The total number of active cards and any errors will also be reported.
- Query the master key verification pattern for any master-key register in the current domain
- Initialize a local host key storage file. See “Using panel.exe for key storage initialization” on page 1006.
- Re-encipher a local host key storage file (use this when the master key has been changed to ensure currency with key storage). See “Using panel.exe for key storage re-encipher when changing the master key” on page 1007.
- List available CPACF functions, and whether they are supported in the current system image.

The panel.exe utility does **not** support access control point manipulation or more sophisticated administration. Refer to “Trusted Key Entry support” on page 50 for that functionality.

Using panel.exe for key storage initialization

Using panel.exe for key storage initialization

Each application using CCA typically creates key objects that are stored in the host, protected by the master key stored inside the card. Perform these steps for key storage initialization.

1. The default locations for the files are setup by the RPM in environment variables added in the new profile files `/etc/profile.d/csulcca.sh` and `/etc/profile.d/csulcca.csh` during installation. Key storage is unsupported without a master key loaded, so **Master key load** (Step 7 on page 995) must be completed before this step. The utility panel.exe can be used to initialize both the default key storage and any separate key storage you might want to set up. The full topic is too lengthy for this explanation (see the key storage topics elsewhere in this manual, including the verb "Key Storage Initialization (CSNBKSI)" on page 114). In brief, an application can specify a particular key storage location. That nondefault key storage can be initialized now (or later) by using panel.exe or with a program using the Key Storage Initialization verb.

   For details about panel.exe, see "The panel.exe utility" on page 1003.
2. The key storage environment variables in the default user profile (`/etc/profile.d/csulcca.sh`) are changed at installation time to point to the /opt/IBM/CCA/keys/ path, where before the path contained /*/4764/*. There is one set of environment variables for a profile. The user can override this by setting a local profile in their home profile file that sets the environment variables back to the 4764 version.
3. Key storage ownership

   The default key storage files are actually partially created (but not fully initialized) during the master key load process. This means the ownership and permissions of those files might have to be changed for them to be fully initialized by the user associated with the application that will use the key storage files.

   Because of the mutually exclusive nature of the master key admin groups, there can be some harmless access errors reported to the system log during master key load. The example users created previously in **Master key load** Step 7a3 on page 996 will avoid this and not need to fix key storage ownership because they were all created with the primary group set to 'cca_admin' (the -g argument to useradd). By doing this, the first master key load creates the key storage files with group set to 'cca_admin' and subsequent users all have membership in that group. You still might want to fix the owner of default key storage at the end to be 'root', but the group membership solves the access issue.

   Typically 'root' will need to fix the ownership and permissions. We recommend that the owner of key storage be 'root', and that the group be 'cca_admin' ('cca_admin' group is created during the RPM install process). We recommend that the permissions be set to 660, which is rw for owner (root), rw for group (cca_admin), and <none> for 'everyone', for security. Then add the application user to the group 'cca_admin' with the appropriate procedure detailed in **Master key load** Step 7a on page 995.

   RECALL: To be able to use /opt/IBM/CCA/panel.exe the user must be either root OR a member of the 'cca_admin' group (the owner.group of `/usr/lib64/libcsulccamk.so`). The reasons for the separate 'cca_admin' group are to allow one owner of `/usr/lib64/libcsulccamk.so`, and to allow use of the executable without allowing any of the master key processing calls.
4. Key storage initialization with panel.exe. This is the default.

a. Ensure permissions to the default location (/opt/IBM/CCA/keys/) allow your user to perform this operation.
b. Initialize key storage (DES is where DES key tokens will be kept, AES is where AES key tokens will be kept, PKA is for all the RSA public/private internal key tokens, and APKA is for APKA key tokens).

```
/opt/IBM/CCA/bin/panel.exe -t AES -i
/opt/IBM/CCA/bin/panel.exe -t DES -i
/opt/IBM/CCA/bin/panel.exe -t PKA -i
```

5. Key storage initialization with panel.exe (non-default)
   a. Ensure that you are using the account that uses the key storage. If you are not, you must fix its ownership and permissions later.
   b. Initialize all types of key storage (DES is where DES key tokens will be kept, AES is where AES key tokens will be kept, PKA is for all the RSA public/private internal key tokens). Use a different name for AES, DES, and PKA, because the second initialization would overwrite the first if different names are not used. Export new environment variables in the session where you will initialize the key storage (and where you sill use it), then initialize key storage again:

```
export CSUAESDS=<AES file name>
export CSUDESDS=<DES file name>
export CSUPKADS=<PKA file name>
/opt/IBM/CCA/bin/panel.exe -t DES -i
/opt/IBM/CCA/bin/panel.exe -t DES -i
/opt/IBM/CCA/bin/panel.exe -t PKA -i
```

   For example, if you entered the following commands:

```
export CSUAESDS=/tmp/a
export CSUDESDS=/tmp/d
export CSUPKADS=/tmp/p
/opt/IBM/CCA/bin/panel.exe -t AES -i
/opt/IBM/CCA/bin/panel.exe -t DES -i
/opt/IBM/CCA/bin/panel.exe -t PKA -i
```

   these files woould be created:

```
/tmp/a
/tmp/a.NDX
/tmp/d
/tmp/d.NDX
/tmp/p
/tmp/p.NDX
```

Using panel.exe for key storage re-encipher when changing the master key

Because all the key tokens are protected by the master key for the domain, a preexisting key storage must be re-enciphered when the master key is changed. I

f the example group scheme is used this is very simple because the key storage files will be owned by the group 'cca_admin' and the user making the reencipher call will also be in group 'cca_admin'. If this is not the case then, after changing the master key, the owner of key storage will need to log in and drive the reencipher. This can be done programmatically (using several verbs) or with /opt/IBM/CCA/panel.exe. Of course, as noted, the user of panel.exe must also be a member of 'cca_admin' because of ownership of `/usr/lib64/libcsulccamk.so`.

Perform these steps for key storage reencipher when changing the master key.

1. To re-encipher default key storage with panel.exe use:

```
/opt/IBM/CCA/bin/panel.exe -t AES -r
/opt/IBM/CCA/bin/panel.exe -t DES -r
/opt/IBM/CCA/bin/panel.exe -t PKA -r
```

2. To reencipher non-default key storage with panel.exe use:
   - Export new versions of the environment variables specifying your key storage file locations.
   - Run the previously shown commands as you would for the default key storage, but ensure to do so in the session with the new environment variables.

Chapter 28. Security API command and sub-command codes

View an alphabetical list of security API command and sub-command codes returned by the output **rule-array** for option STATDIAG of the Cryptographic Facility Query verb.

Elements 9, 11, 13, 15, and 17 contain these API command and sub-command codes to indicate the last five commands run. See Table 13 on page 81.

Note: The security API command is *T2* padded on the right with 2 space characters, and the subcommand code is a 2-byte uppercase alphabetic value padded on the right with 2 space characters.

Table 273. Alphabetical list of security API command and subcommand codes returned by STATDIAG.

Security API command code	Subcommand code	Description
T2	AD	Symmetric Algorithm Decipher (CSNBSAD)
T2	AE	Symmetric Algorithm Encipher (CSNBSAE)
T2	AI	Access Control Initialization (CSUAACI)
T2	AK	Diversified Key Generate2 (CSNBDKG2)
T2	AM	Access Control Maintenance (CSUAACM)
T2	AP	Authentication Parameter Generate (CSNBAPG)
T2	CA	Cryptographic Resource Allocate (CSUACRA)
T2	CD	Cryptographic Resource Deallocate (CSUACRD)
T2	CI	Clear Key Import (CSNBCKI)
T2	CM	Multiple Clear Key Import (CSNBCKM)
T2	CT	Key Token Change (CSNBKTC)
T2	CV	MAC Generate2 (CSNBMGN2)
T2	CX	Control Vector Translate (CSNBCVT)
T2	CY	Key Token Change2 (CSNBKTC2)
T2	DA	DK Deterministic PIN Generate (CSNBDDPG)
T2	DB	DK PAN Translate (CSNBDPT)
T2	DC	Decipher (CSNBDEC)
T2	DE	Data Key Export (CSNBDKX)
T2	DH	EC Diffie-Hellman (CSNDEDH)
T2	DI	Data Key Import (CSNBDKM)
T2	DK	Retained Key Delete (CSNDRKD)
T2	DM	DK PRW CMAC Generate (CSNBDPCG)
T2	DN	DK PIN Change (CSNBDPC)
T2	DQ	DK Migrate PIN (CSNBDMP)
T2	DR	DK Random PIN Generate (CSNBDRPG)

Table 273. Alphabetical list of security API command and subcommand codes returned by STATDIAG (continued).

Security API command code	Subcommand code	Description
T2	DS	Symmetric Key Export with Data (CSNDSXD)
T2	DT	DK PAN Modify in Transaction (CSNBDPMT)
T2	DU	DK PRW Card Number Update (CSNBDPNU)
T2	DV	DK PIN Verify (CSNBDPV)
T2	DW	DK Regenerate PRW (CSNBDRP)
T2	EC	Encipher (CSNBENC)
T2	EK	Key Encryption Translate (CSNBKET)
T2	EP	Encrypted PIN Generate (CSNBEPG)
T2	EX	Prohibit Export Extended (CSNBPEXX)
T2	EY	Restrict Key Attribute (CSNBRKA)
T2	FC	Cryptographic Facility Control (CSUACFC)
T2	FQ	Cryptographic Facility Query (CSUACFQ)
T2	GC	MAC Verify2 (CSNBMVR2)
T2	GD	Diversified Key Generate (CSNBDKG)
T2	GH	HMAC Generate (CSNBHMG) - SHA-1, SHA-224, SHA-256
T2	GK	Key Generate2 (CSNBKGN2)
T2	GL	HMAC Generate (CSNBHMG) - SHA-384, SHA-512
T2	GM	MDC Generate (CSNBMDG)
T2	GS	Symmetric Key Generate (CSNDSYG)
T2	IP	Key Part Import2 (CSNBKPI2)
T2	KC	PKA Public Key Register (CSNDPKR)
T2	KD	Master Key Distribution (CSUAMKD)
T2	KE	Key Export (CSNBKEX)
T2	KG	Key Generate (CSNBKGN)
T2	KH	PKA Public Key Hash Register (CSNDPKH)
T2	KI	Key Import (CSNBKIM)
T2	KN	Key Translate2 (CSNBKTR2)
T2	KP	Key Part Import (CSNBKPI)
T2	KR	Key Translate (CSNBKTR)
T2	KT	Key Test (CSNBKYT)
T2	KX	Key Test Extended (CSNBKYTX)
T2	KY	Key Test2 (CSNBKYT2)
T2	LK	Retained Key List (CSNDRKL)
T2	LQ	Log Query (CSUALGQ)
T2	MG	MAC Generate (CSNBMGN)
T2	MP	Master Key Process (CSNBMKP)
T2	MV	MAC Verify (CSNBMVR)

Table 273. Alphabetical list of security API command and subcommand codes returned by STATDIAG (continued).

Security API command code	Subcommand code	Description
T2	OH	One Way Hash (CSNBOWH)
T2	PA	Clear PIN Generate Alternate (CSNBCPA)
T2	PC	Clear PIN Generate (CSNBPGN)
T2	PD	PKA Decrypt (CSNDPKD)
T2	PE	Clear PIN Encrypt (CSNBCPE)
T2	PG	PKA Key Generate (CSNDPKG)
T2	PI	PKA Key Import (CSNDPKI)
T2	PK	PKA Encrypt (CSNDPKE)
T2	PO	Recover PIN from Offset (CSNBPFO)
T2	PT	Encrypted PIN Translate (CSNBPTR)
T2	PU	PIN Change/Unblock (CSNBPCU)
T2	PV	Encrypted PIN Verify (CSNBPVR)
T2	PX	Prohibit Export (CSNBPEX)
T2	RE	TR31 Key Export (CSNBT31X)
T2	RG	Random Number Generate (CSNBRNG)
T2	RI	TR31 Key Import (CSNBT31I)
T2	RK	Remote Key Export (CSNDRKX)
T2	RL	Random Number Generate Long (CSNBRNGL)
T2	RT	Random Number Tests (CSUARNT)
T2	SC	SET Block Compose (CSNDSBC)
T2	SD	SET Block Decompose (CSNDSBD)
T2	SG	Digital Signature Generate (CSNDDSG)
T2	SI	Symmetric Key Import (CSNDSYI)
T2	SJ	Symmetric Key Import2 (CSNDSYI2)
T2	SP	Secure Messaging for PINs (CSNBSPN)
T2	SV	Digital Signature Verify (CSNDDSV)
T2	SX	Symmetric Key Export (CSNDSYX)
T2	SY	Secure Messaging for Keys (CSNBSKY)
T2	TB	Trusted Block Create (CSNDTBC)
T2	TC	PKA Key Token Change (CSNDKTC)
T2	TK	PKA Key Translate (CSNDPKT)
T2	TT	Cipher Text Translate2 (CSNBCTT2)
T2	TV	Transaction Validation (CSNBTRV)
T2	UD	Unique Key Derive (CSNBUKD)
T2	VC	CVV Key Combine (CSNBCKC)
T2	VE	Cryptographic Variable Encipher (CSNBCVE)
T2	VG	CVV Generate (CSNBCSG)

Table 273. Alphabetical list of security API command and subcommand codes returned by STATDIAG (continued).

Security API command code	Subcommand code	Description
T2	VH	HMAC Verify (CSNBHMV) - SHA-1, SHA-224, SHA-256
T2	VL	HMAC Verify (CSNBHMV) - SHA-384, SHA-512
T2	VV	CVV Verify (CSNBCSV)

Chapter 29. openCryptoki support

openCryptoki is an open source implementation of the *Cryptoki* API as defined by the industry-wide PKCS #11 Cryptographic Token Interface Standard.

openCryptoki supports several cryptographic algorithms according to the PKCS #11 standards. The openCryptoki library loads plug-ins that provide hardware or software-specific support for cryptographic functions.

In PKCS #11 terminology, and hence in openCryptoki terminology, these plug-ins are called *tokens*. Do not confuse *token* in the context of PKCS #11 with *token* in the CCA context.

openCryptoki and PKCS #11
: In the context of openCryptoki and PKCS #11, a token is a representation of a hardware or software component that implements cryptographic functions. For example, a PKCS #11 token can represent a cryptographic adapter, a smart card, or a cryptographic library.

  The term *CCA token* is used to denote the openCryptoki plug-in for CCA. The CCA token represents a library that extends the openCryptoki token library and links to the CCA library (libcsulcca).

CCA
: In the context of CCA, a *token* is a representation of a key and the attributes belonging to the key.

For more information about the openCryptoki services or about the interfaces between the openCryptoki main module and its tokens obtain an openCryptoki package from sourceforge.net/projects/opencryptoki and see the documents in the `doc` directory within the package.

The openCryptoki token directories

Each token has a separate token directory, which is used by openCryptoki to store token-specific information, like key objects, the user PIN, or the SO PIN. For most Linux distributions, the CCA token directory is `/var/lib/opencryptoki/ccatok`.

Note: The data in the token directory applies to all applications that use the token.

Prerequisites

- The libcsulcca library must be installed, see Chapter 25, "CCA installation instructions," on page 991.
- The cryptographic device driver must be loaded, see "Installing and loading the cryptographic device driver" on page 987.
- A Crypto Express coprocessor (see CEX*C in "Terminology" on page xxiv) must be accessible to Linux. For Linux on z/VM, this coprocessor must be dedicated to the guest.
- The master keys must be loaded and set in the coprocessor, see Chapter 27, "Utilities," on page 1003.

Configuring openCryptoki for CCA support

After installing openCryptoki, you must configure tokens, start a daemon, and then initialize the tokens.

openCryptoki can handle several tokens with support for different hardware devices or software solutions. The CCA token interacts with the host part of the CCA library.

Perform the following tasks to configure the Linux on z Systems CCA enablement:

- "Confirming the openCryptoki configuration file"
- "Starting the openCryptoki slot daemon" on page 1015
- "Initializing the token" on page 1016

Confirming the openCryptoki configuration file

The global openCryptoki configuration file must define a slot entry for the CCA token.

As a minimum requirement, a slot entry must specify the library that implements the token. Most Linux distributions provide a default configuration file with a valid slot entry for the CCA token.

Table 274 lists the libraries that might be in place after you successfully installed openCryptoki. The actual list for your installation depends on your distribution and on the installed RPM packages.

Table 274. openCryptoki libraries.

Library	Explanation
`/usr/lib64/opencryptoki/libopencryptoki.so`	openCryptoki base library
`/usr/lib64/opencryptoki/stdll/libpkcs11_ica.so`	openCryptoki libica token library
`/usr/lib64/opencryptoki/stdll/libpkcs11_sw.so`	openCryptoki software token library
`/usr/lib64/opencryptoki/stdll/libpkcs11_tpm.so`	openCryptoki TPM token library (not supported by Linux on z Systems)
`/usr/lib64/opencryptoki/stdll/libpkcs11_cca.so`	openCryptoki CCA token library
`/usr/lib64/opencryptoki/stdll/libpkcs11_ep11.so`	openCryptoki EP11 token library
`/usr/lib64/opencryptoki/stdll/libpkcs11_icsf.so`	openCryptoki ICSF token library

Note: The CCA token library is available only in 64-bit mode.

The `/etc/opencryptoki/opencryptoki.conf` file must exist and it must contain an entry for the CCA token to make the token available. By default, this entry is available after installing the CCA token. See the `slot 2` entry in the following sample configuration:

```
-------------- content of opencryptoki.conf ---------
version opencryptoki-3.2
# The following defaults are defined:
#       hwversion = 0.0
#       firmwareversion = 0.0
#       description = Linux
#       manufacturer = IBM
#
# The slot definitions below may be overriden and/or customized.
# For example:
#       slot 0
#       {
#          stdll = libpkcs11_cca.so
#          description = "OCK CCA Token"
#          manufacturer = "MyCompany Inc."
#          hwversion = 2.32
#          firmwareversion = 1.0
#       }
# See man(5) opencryptoki.conf for further information.
#
slot 0
{
stdll = libpkcs11_tpm.so
}

slot 1
{
stdll = libpkcs11_ica.so
}

slot 2
{
stdll = libpkcs11_cca.so
}

slot 3
{
stdll = libpkcs11_sw.so
}

slot 4
{
stdll = libpkcs11_ep11.so
confname = ep11tok.conf
}
---------------------------- end ---------------------------------
```

Note: The default path for slot token dynamic link libraries (STDLLs) is `/usr/lib64/opencryptoki/stdll/`.

A token is available only if the token library is installed and the software and hardware support that is required by the token is also installed. For example, the CCA token is available only if all parts of the CCA library software are installed and a CCA coprocessor is detected. Use the following command to display the list of available tokens:

```
$ pkcsconf -t
```

Starting the openCryptoki slot daemon

The openCryptoki slot daemon reads the configuration information and sets up the tokens.

Before you begin

The openCryptoki configuration file must list all available tokens that are ready to register to the openCryptoki slot daemon, see "Confirming the openCryptoki configuration file" on page 1014.

Procedure

Start the slot daemon by using one of the following commands:

```
$ pkcsslotd
$ pkcsslotd start
$ systemctl start pkcsslotd.service
```

Use the following command to make the daemon start automatically after a reboot:

```
$ chkconfig pkcsslotd on
```

Initializing the token

After the openCryptoki library and the global configuration file are set up and the **pkcsslotd** daemon is started, the CCA token must be initialized.

PKCS #11 defines two users for each token: a security officer (SO) whose responsibility is the administration of the token, and a standard user (User) who wants to use the token to perform cryptographic operations. openCryptoki requires that for both the SO and the User a login PIN is defined as part of the token initialization.

The following command provides some useful slot information:

```
# pkcsconf -s
Slot #1 Info
        Description: OCK ICA Token
        Manufacturer: IBM
        Flags: 0x1 (TOKEN_PRESENT)
        Hardware Version: 0.0
        Firmware Version: 0.0
Slot #2 Info
        Description: OCK CCA Token
        Manufacturer: IBM
        Flags: 0x1 (TOKEN_PRESENT)
        Hardware Version: 0.0
        Firmware Version: 0.0
Slot #3 Info
        Description: Software Token
        Manufacturer: IBM
        Flags: 0x1 (TOKEN_PRESENT)
        Hardware Version: 0.0
        Firmware Version: 0.0
```

Find your preferred token in the details list and select the correct slot number. This number is used in the next initialization steps to identify your token:

```
$ pkcsconf -I -c <slot> // Initialize the Token and setup a Token Label

$ pkcsconf -P -c <slot> // change the SO PIN (recommended)

$ pkcsconf -u -c <slot> // Initialize the User PIN (SO PIN required)

$ pkcsconf -p -c <slot> // change the User PIN (optional)
```

pkcsconf -I
During token initialization, you are asked for a token label. Provide a meaningful name that helps you later to identify the token.

pkcsconf -P
openCryptoki security practices require that you change the default SO PIN (87654321) to a different value. Use the `pkcsconf -P` option to change the SO PIN.

pkcsconf -u
When you enter the user PIN initialization, you are asked for the newly set SO PIN. The length of the user PIN must be 4 - 8 characters.

pkcsconf -p
openCryptoki security practices require that you change the user PIN at least once with `pkcsconf -p` option. The length of the user PIN must be 4 - 8 characters. After you completed the PIN setup, the token is prepared and ready for use.

Note: Specify a user PIN that is different from `12345678` because this pattern is checked internally and marked as default PIN. A login attempt with this user PIN is recognized as *not initialized*.

How to recognize the CCA token

Use the **pkcsconf -t** command to display a list of all available tokens. You can check the slot and token information, and the PIN status at any time.

The following example shows information about the CCA token in the `Token #2 Info` section. This section provides information about the token that is plugged into slot number 2.

```
$ pkcsconf -t
...
Token #2 Info:
        Label: CCA Token
        Manufacturer: IBM Corp.
        Model: IBM CCA Token
        Serial Number: 123
        Flags: 0x44D (RNG|LOGIN_REQUIRED|USER_PIN_INITIALIZED|CLOCK_ON_TOKEN|TOKEN_INITIALIZED)
        Sessions: 0/-2
        R/W Sessions: -1/-2
        PIN Length: 4-8
        Public Memory: 0xFFFFFFFF/0xFFFFFFFF
        Private Memory: 0xFFFFFFFF/0xFFFFFFFF
        Hardware Version: 1.0
        Firmware Version: 1.0
        Time: 16:10:40
...
```

The output includes the following information:

Label The token label that was assigned when the token was initialized. You can change token labels with the **pkcsconf -I** command.

Model A unique designation for the token, “IBM CCA Token” for the CCA token.

Flags A mask with information about the token initialization status, the PIN status, and features such as random number generator (RNG). For example, the mask for TOKEN_INITIALIZED is 0x00000400 and it is true if the token was initialized. “Login required” means that there is at least one mechanism that requires a session login to use that cryptographic function.

For more information about the flags, see the description of the TOKEN_INFO structure and the token information flags in the PKCS #11 Cryptographic Token Interface Standard.

PIN length
The PIN length range that was specified for the token.

Using the CCA token

You can use some CCA library functions through the openCryptoki standard interface (PKCS #11 standard C API).

The PKCS #11 Cryptographic Token Interface Standard describes the exact API.

Applications that are designed to work with openCryptoki can use the Linux on z Systems CCA enablement.

Supported mechanisms for the CCA token

Use the **pkcsconf** command to list the mechanisms (algorithms), that are supported by the CCA token.

Issue the **pkcsconf** command with the `-m` parameter to display mechanisms. Use the `-c` parameter to specify the slot number for the token of interest. The following example assumes that this slot number is `2`:

```
# pkcsconf -m -c2
...
Mechanism #11
        Mechanism: 0x1080 (CKM_AES_KEY_GEN)
        Key Size: 16-32
        Flags: 0x8001 (CKF_HW|CKF_GENERATE)
Mechanism #12
        Mechanism: 0x1081 (CKM_AES_ECB)
        Key Size: 16-32
        Flags: 0x60301 (CKF_HW|CKF_ENCRYPT|CKF_DECRYPT|CKF_WRAP|CKF_UNWRAP)
Mechanism #13
        Mechanism: 0x1082 (CKM_AES_CBC)
        Key Size: 16-32
        Flags: 0x60301 (CKF_HW|CKF_ENCRYPT|CKF_DECRYPT|CKF_WRAP|CKF_UNWRAP)
...
```

The command output lists all mechanisms that are supported by the token in the specified slot. The mechanism ID and name correspond to the PKCS #11 specification.

Each entry includes the minimum and maximum supported key size and other properties, like hardware support and mechanism information flags. These flags provide information about the PKCS #11 functions that can use the mechanism. For some mechanisms, the flags show further attributes that describe the supported variants of the mechanism.

The supported Crypto Express coprocessors together with openCryptoki version 3.2 and CCA token support these PKCS #11 mechanisms:

Table 275. PKCS #11 mechanisms supported by the CCA token

Mechanism	Key sizes	Properties
CKM_AES_CBC	16-32	ENCRYPT, DECRYPT, WRAP, UNWRAP
CKM_AES_CBC_PAD	16-32	ENCRYPT, DECRYPT, WRAP, UNWRAP

Table 275. PKCS #11 mechanisms supported by the CCA token (continued)

Mechanism	Key sizes	Properties
CKM_AES_ECB	16-32	ENCRYPT, DECRYPT, WRAP, UNWRAP
CKM_AES_KEY_GEN	16-32	GENERATE
CKM_DES3_CBC	24-24	ENCRYPT, DECRYPT, WRAP, UNWRAP
CKM_DES3_CBC_PAD	24-24	ENCRYPT, DECRYPT, WRAP, UNWRAP
CKM_DES3_KEY_GEN	24-24	GENERATE
CKM_DES_CBC	8-8	ENCRYPT, DECRYPT, WRAP, UNWRAP
CKM_DES_CBC_PAD	8-8	ENCRYPT, DECRYPT, WRAP, UNWRAP
CKM_DES_KEY_GEN	8-8	GENERATE
CKM_ECDSA	160-521	SIGN, VERIFY, EC_F_P, EC_NAMEDCURV
CKM_ECDSA_KEY_PAIR_GEN	160-521	GENERATE_KEY_PAIR, EC_F_P, EC_NAMEDCUVE
CKM_ECDSA_SHA1	160-521	SIGN, VERIFY, EC_F_P, EC_NAMEDCURV
CKM_MD5	0-0	DIGEST
CKM_MD5_RSA_PKCS	512-4096	SIGN, VERIFY
CKM_RSA_PKCS	512-4096	ENCRYPT, DECRYPT, SIGN, VERIFY
CKM_RSA_PKCS_KEY_PAIR_GEN	512-4096	GENERATE_KEY_PAIR
CKM_SHA1_RSA_PKCS	512-4096	SIGN, VERIFY
CKM_SHA256	0-0	DIGEST
CKM_SHA256_RSA_PKCS	512-4096	SIGN, VERIFY
CKM_SHA384	0-0	DIGEST
CKM_SHA512	0-0	DIGEST
CKM_SHA_1	0-0	DIGEST

For explanations of the key object properties, see the PKCS #11 Cryptographic Token Interface Standard.

Restrictions with using the CCA library functions

The CCA library is subject to some limitations.

- Key imports through the C_CreateObject PKCS #11 function are supported for RSA private keys and for RSA public keys in CRT format only.
- The default CKA_SENSITIVE setting for generating a key is CK_FALSE although the openCryptoki CCA token handles only secure keys, which correspond to sensitive keys in PKCS #11.

  Setting the value of CKA_SENSITIVE to CK_FALSE does not inhibit inspecting the value of CKA_VALUE. This setting does not compromise security because inspecting the value of CKA_VALUE does not reveal any sensitive information. The CCA secure key token (*token* as used in the CCA context) is stored in the CKA_IBM_OPAQUE attribute rather than in the CKA_VALUE attribute. Moreover, the key value is never stored in clear text.

Migrating openCryptoki version 2 tokens to version 3

PKCS #11 token objects are persistent in the token. openCryptoki stores token objects in the token directory, `/var/lib/opencryptoki/ccatok/TOK_OBJ`, for the CCA token.

Private openCryptoki token objects that have been saved in openCryptoki version 2 format must be migrated before they can be used with openCryptoki version 3.

Before you begin

Public openCryptoki token objects are not encrypted and no migration is required.

Note: Throughout this section, *token object* is used to mean a PKCS #11 object, like a key, and *CCA token* is used to denote the openCryptoki plug-in for CCA.

About this task

In openCryptoki version 2, private token objects are encrypted and decrypted with a secure key in the cryptographic adapter. In version 3, this encryption and decryption is done with a clear key using cryptographic software functions. Therefore, openCryptoki version 3 cannot decrypt a version 2 private token object.

Migration decrypts the private key objects by using the CSNBDEC CCA verb and the openCryptoki key stored in MK_USER. The objects are then re-encrypted using the cryptographic software functions according to openCryptoki version 3. The key bits that are stored in MK_USER are subsequently used as a clear key.

You can find more detailed information about token migration in the documentation of the openCryptoki 3.2 RPM.

Procedure

1. Back up the CCA token directory. For most Linux distributions, this directory is `/var/lib/opencryptoki/ccatok`. The content of the CCA token directory includes the following items:

   MK_USER
   The TDES key used for internal on-disk encryption, encrypted under the user's PIN by software routines.

   MK_SO
   The TDES key used for internal on-disk encryption, encrypted under the SO's PIN by software routines. This is the same key as in MK_USER, but with a different encrytion.

   NKTOK.DAT
   Token information.

   TOK_OBJ
   The directory in which token objects are stored.

   TOK_OBJ/OBJ.IDX
   A list of current token objects.

2. Upgrade to openCryptoki version 3 or install version 3.
   - Upgrade to openCryptoki version 3.

     For most Linux distributions, upgrading from version 2 to version 3 preserves the contents of the CCA token directory.

- Install openCryptoki version 3.

  For most Linux distributions, installing version 3 removes the contents of the CCA token directory. Restore your backup of this directory to migrate the version 2 tokens.

3. Ensure that no openCryptoki processes are running.
   a. Stop all applications that use openCryptoki.
   b. Find out whether the `pkcsslotd` daemon is running by issuing the following command:

      ```
      $ ps awx |grep pkcsslotd
      ```

      If the daemon is running, the command output shows a process for `pkcsslotd`.
   c. If applicable, stop the daemon by issuing a command of this form:

      ```
      $ killall pkcsslotd
      ```

4. Run **pkcscca**.

   Example:

   ```
   $ pkcscca -m v2objectsv3 -v
   ```

   The `-v` option prints information about which objects did and did not get migrated. This command migrates CCA token objects in the default CCA token directory, `/var/lib/opencryptoki/ccatok`.

   If your distribution uses a different token directory, use the `-d` option to specify this directory.

   Example:

   ```
   $ pkcscca -m v2objectsv3 -v -d /home/ccaadmin/ccatok
   ```

Results

The CCA private token objects are encrypted according to openCryptoki version 3 and ready to be accessed.

What to do next

If openCryptoki version 3 cannot find the newly migrated CCA private token objects, reboot or remove the shared memory file, `/dev/shm/var.lib.opencryptoki.ccatok`.

Attention: Ensure that no openCryptoki processes are running when removing the shared memory.

Chapter 30. List of abbreviations

A list of abbreviations used in this document.

ADB Actual Data Block

AES Advanced Encryption Standard

AESKW
AES Key Wrap (ANS X9.102)

AIX® Advanced Interactive Executive operating system

ANS American National Standards

ANSI American National Standards Institute

API Application Programming Interface

ASCII American National Standard Code for Information Interchange

ASN Abstract Syntax Notation

ATC Application Transaction Counter

ATM Automated Teller Machine

BC Block Contents

BDK Base Derivation Key

BER ASN.1 Basic Encoding Rules

CA Certification Authority

CBC Cipher block chaining

CCA Common Cryptographic Architecture

CEX2A
Crypto Express2 Accelerator

CEX2C
Crypto Express2 Coprocessor

CEX3A
Crypto Express3 Accelerator

CEX3C
Crypto Express3 Coprocessor

CEX4A
Crypto Express4 Accelerator

CEX4C
Crypto Express4 Coprocessor

CEX5A
Crypto Express5 Accelerator

CEX5C
Crypto Express5 Coprocessor

CKDS Cryptographic Key Data Set.

CKSN Current-Key Serial Number

CLU Coprocessor Load Utility

CMAC
Block Cipher-based Message Authentication Code Algorithm (analogous to HMAC), NIST SP 800-38B

CMK Current Master Key

CMOS
Complementary Metal Oxide Semiconductor

CMS Cryptographic Message Syntax

CNI Coprocessor Node Initialization

CNM Cryptographic Node Management (utility)

CPACF
Central Processor Assist for Cryptographic Functions

CRT Chinese Remainder Theorem.

CSC Card Security Code

CV Control Vector

CVC Card verification code used by MasterCard.

CVK Card Verification Key

CVV Card verification value used by VISA.

DEA Data encryption algorithm

DES Data Encryption Standard

DK Deutsche Kreditwirtschaft (German Banking Industry Committee). Formerly known as ZKA.

DMA Direct Memory Access

DOW Day of the Week

DRBG
Deterministic Random Bit Generator

DRBGVS
Deterministic Random Bit Generator Validation System (NIST SP 800-90A)

DSA Digital Signature Algorithm.

DSS Digital Signature Standard.

DUKPT
Derived Unique Key Per Transaction

EBCDIC
Extended Binary Coded Decimal Interchange Code

EC Elliptic Curve

ECB Electronic codebook.

ECC Elliptic Curve Cryptography

ECDH Elliptic Curve Diffie-Hellman

ECDSA
Elliptic Curve Digital Signature Algorithm

ECI Eurocheque International

EDE Encipher-Decipher-Encipher

EEPROM
Electrically Erasable, Programmable Read-Only Memory

EID Environment Identification.

EPP Encrypting PIN PAD

FCV Function Control Vector

FIPS Federal Information Processing Standards

FPE Format Preserving Encryption

GBP German Bank Pool.

HMAC
Keyed Hash MAC

HSM Hardware Security Module

IBM International Business Machines

ICSF Integrated Cryptographic Service Facility.

ICV Initial Chaining Value

IEC International Electrotechnical Commission

IETF Internet Engineering Task Force

I/O Input/Output

IPEK Initial PIN Encryption Key

IPL Initial Program Load

ISO International Organization for Standardization.

ISO/DIS
International Organization for Standardization/Draft International Standard

ISO/FDIS
International Organization for Standardization/Final Draft International Standard

JNI Java Native Interface

KC Key Confirmation

KDF Key Derivation Function

KEK Key-Encrypting Key

KM Master Key

KVP Key Verification Pattern

LRC Longitudinal Redundancy Check

LSB Least significant bit

MB Megabyte

MAC Message Authentication Code

MD5 Message Digest-5 Hash Algorithm

MDC Modification Detection Code

MDK Master-Derivation Key

MFK Master File Key

MK Master Key

MKVP
Master-Key Verification Pattern

MSB Most significant bit

NIST U.S. National Institute of Science and Technology.

NMK New Master Key

OAEP Optimal asymmetric encryption padding.

OCSP Open Certificate Status Protocol

OEM Original Equipment Manufacturer

OID Object Identifier

OMK Old Master Key

OPK Object Protection Key

PAN Personal Account Number

PBF PIN Block Format

PCI Peripheral Component Interconnect

PCIe PCI Express

PCI-X PCI Extended

PCICA
PCI Cryptographic Accelerator.

PCICC
PCI Cryptographic Coprocessor.

PCIXCC
PCI X Cryptographic Coprocessor.

PIN Personal Identification Number

PKA Public Key Algorithm.

PKCS Public Key Cryptographic Standards (RSA Data Security, Inc.)

PKDS Public key data set (PKA cryptographic key data set).

POST Power-On Self Test

PRNG Pseudo Random Number Generator

PROM
Programmable Read-Only Memory

PRW PIN reference word/value

PVV PIN Validation Value

RA Registration Authority

RACF Resource Access Control Facility

RAM Random Access Memory

RFC Request for Comments

RHEL Red Hat Enterprise Linux

RNG Random Number Generator

ROM Read-Only Memory

RPM RPM Package Manager (originally: Red Hat Package Manager)

RSA Rivest, Shamir, and Adleman

SEC 2 Standards for Efficient Cryptography 2

SECG Standards for Efficient Cryptography Group

SET Secure Electronic Transaction.

SHA Secure Hash Algorithm

SLES SUSE Linux Enterprise Server

SNA Systems Network Architecture

SSL Secure Sockets Layer.

TDEA Triple Data Encryption Algorithm (see also TDES).

TDES Triple DES (Data Encryption Standard) or TDEA.

TKE Trusted key entry.

TLV Tag, Length, Value

TMK Terminal Master Key

TVV Token-validation value

UAT UDX Authority Table.

UDF User-defined function.

UDK User-derived key.

UDP User Developed Program.

UDX User Defined Extension.

UKPT Unique Key Per Transaction

UTC Coordinated Universal Time

VIS Visa Integrated Circuit Card Specification

XOR Exlusive-OR

Part 4. Appendixes

Appendix. Accessibility

Accessibility features help users who have a disability, such as restricted mobility or limited vision, to use information technology products successfully.

Documentation accessibility

The Linux on z Systems publications are in Adobe Portable Document Format (PDF) and should be compliant with accessibility standards. If you experience difficulties when you use the PDF file and want to request a Web-based format for this publication, use the Readers' Comments form in the back of this publication, send an email to eservdoc@de.ibm.com, or write to:

IBM Deutschland Research & Development GmbH
Information Development
Department 3282
Schoenaicher Strasse 220
71032 Boeblingen
Germany

In the request, be sure to include the publication number and title.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

IBM and accessibility

See the IBM Human Ability and Accessibility Center for more information about the commitment that IBM has to accessibility at

`www.ibm.com/able`

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

Programming interface information

This book documents intended Programming Interfaces that allow the customer to write programs to obtain the services of the Common Cryptographic Architecture.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at:

`www.ibm.com/legal/copytrade.shtml`

Adobe is either a registered trademark or trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Glossary

This glossary includes some terms and definitions from the *IBM Dictionary of Computing*, New York: McGraw Hill, 1994. This glossary also includes some terms and definitions from:

- The *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the U.S. American National Standards Institute (ANSI). Definitions are identified by the symbol (A) after the definition.
- The *IBM Glossary of Computing Terms*. Definitions are identified by the symbol (D) after the definition.
- The *IBM TotalStorage Enterprise Storage Server®* documentation. Definitions of published parts of this vocabulary are identified by the symbol (E) after the definition.
- The *Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

access A specific type of interaction between a subject and an object that results in the flow of information from one to the other.

access control
Ensuring that the resources of a computer system can be accessed only by authorized users in authorized ways.

access method
A technique for moving data between main storage and input/output devices.

adapter
A printed circuit card that modifies the system unit to allow it to operate in a particular way.

address
In data communication, the unique code assigned to each device or workstation connected to a network. A character or group of characters that identifies a register, a particular part of storage, or some other data source or data destination. (A) To refer to a device or an item of data by its address. (A) (I)

Advanced Encryption Standard (AES)
A data encryption technique that improved upon and officially replaced the Data Encryption Standard (DES). AES is sometimes referred to as Rijndael, which is the algorithm on which the standard is based.

Advanced Interactive Executive (AIX) operating system
IBM's implementation of the UNIX[4] operating system.

American National Standard Code for Information Interchange (ASCII)
The standard code using a coded character set consisting of 7-bit characters (8 bits including parity check) that is used for information exchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

American Institute of Standardization (ANSI)
An organization, consisting of producers, consumers, and general interest groups that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States. (A)

ANSI key-encrypting key (AKEK)
A 64- or 128-bit key used exclusively in ANSI X9.17 key management applications to protect data keys exchanged between systems.

ANSI X9.17
An ANSI standard that specifies algorithms and messages for DES key distribution.

4. UNIX is a trademark of UNIX Systems Laboratories, Incorporated.

ANSI X9.19
An ANSI standard that specifies an optional double-MAC procedure which requires a double-length MAC key.

application program
A program written for or by a user that applies to the user's work, such as a program that does inventory control or payroll.

A program used to connect and communicate with stations in a network, enabling users to perform application-oriented activities. (D)

application program interface (API)
A functional interface supplied by the operating system or by a separately deliverable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program. (D)

Application System/400 system (AS/400)
AS/400 was one of a family of general purpose mid-range systems with a single operating system, Operating System/400®, that provides application portability across all models. AS/400 is now referred to as IBM System i®.

assembler language
A source language that includes symbolic machine language statements in which there is a one-to-one correspondence between the instruction formats and the data formats of the computer.

asymmetric cryptography
Synonym for public key cryptography. (D)

authentication
A process used to verify the integrity of transmitted data, especially a message (I). In computer security, a process used to verify the user of an information system or protected resources.

authentication pattern
An 8-byte pattern that is calculated from the master key when initializing the cryptographic key data set. The value of the authentication pattern is placed in the header record of the cryptographic key data set.

authorize
To permit or give authority to a user to communicate with or make use of an object, resource, or function.

authorization
The right granted to a user to communicate with or make use of a computer system (I). The process of granting a user either complete or restricted access to an object, resource, or function.

bus In a processor, a physical facility along which data is transferred.

byte A binary character operated on as a unit and usually shorter than a computer word. (A) A string that consists of a number of bits, treated as a unit, and representing a character. A group of eight adjacent binary digits that represents one EBCDIC character.

C

Card-Verification Code (CVC)
See *Card-Verification Value*.

Card-Verification Value (CVV)
A cryptographic method, defined by VISA, for detecting forged magnetic-striped cards. This method cryptographically checks the contents of a magnetic stripe. This process is functionally the same as MasterCard's Card-Verification Code (CVC) process.

Central Processor Assist for Cryptographic Functions (CPACF)
Implemented on all z890, z990, z9 EC, z9 BC, z10™ EC and z10 BC processors to provide SHA-1 secure hashing.

channel
A path along which signals can be sent; for example, a data channel or an output channel. (A)

checksum
The sum of a group of data associated with the group and used for checking purposes. (T)

Chinese Remainder Theorem (CRT)
A mathematical theorem that defines a format for the RSA private key that improves performance.

Cipher Block Chaining (CBC)
A mode of encryption that uses the data encryption algorithm and requires an initial chaining vector. For encipher, it

exclusively ORs the initial block of data with the initial control vector and then enciphers it. This process results in the encryption both of the input block and of the initial control vector that it uses on the next input block as the process repeats. A comparable chaining process works for decipher.

ciphertext
Text that results from the encipherment of plain text. Synonym for enciphered data. (D). See also *plain text*.

clear data
Data that is not enciphered.

clear key
Any type of encryption key not protected by encryption under another key.

clear text
Text that has not been altered by a cryptographic process. Synonym for plain text. See also *ciphertext*.

Common Cryptographic Architecture (CCA)
The CCA API is the programming interface described in this document.

Common Cryptographic Architecture: Cryptographic Application Programming Interface
Defines a set of cryptographic functions, external interfaces, and a set of key management rules that provide a consistent, end-to-end cryptographic architecture across different IBM platforms.

concatenation
An operation that joins two characters or strings in the order specified, forming one string whose length is equal to the sum of the lengths of its parts.

configuration
The manner in which the hardware and software of an information processing system are organized and interconnected. (I) The physical and logical arrangement of devices and programs that constitutes a data processing system.

console
A part of a computer used for communication between the operator or maintenance engineer and the computer. (A)

control block
A storage area used by a computer program to hold control information. (I) Synonymous with control area.

The circuitry that performs the control functions such as decoding micro instructions and generating the internal control signals that perform the operations requested. (A)

control vector (CV)
In CCA, a 16-byte string that is exclusive-OR-ed with a master key or a key-encrypting key to create another key that is used to encipher and decipher data or data keys. A control vector determines the type of key and the restrictions on the use of that key.

coprocessor
In this document, the IBM 4765 PCIe and IBM 4764 PCI-X Cryptographic Coprocessors, generally also when using the CCA Support Program.

Crypto Express2 Coprocessor
An asynchronous cryptographic coprocessor available on the z890, z990, z9 EC, z9 BC, z10 EC and z10 BC.

Crypto Express3 Coprocessor
An asynchronous cryptographic coprocessor available on z10 EC and z10 BC.

cryptographic adapter (4755 or 4758)
An expansion board that provides a comprehensive set of cryptographic functions for the network security processor and the workstation in the TSS family of products.

cryptographic coprocessor
A microprocessor that adds cryptographic processing functions to specific z890, z990, z9 EC, z9 BC, z10 EC and z10 BC processors. The Cryptographic Coprocessor Feature is a tamper-resistant chip built into the processor board.

cryptographic key data set (CKDS)
A data set that contains the encrypting keys used by an installation. (D)

cryptography
The transformation of data to conceal its meaning.

In computer security, the principles, means, and methods for encrypting plain text and decrypting ciphertext. (D)

CUSP (Cryptographic Unit Support Program)
The IBM cryptographic offering, program product 5740-XY6, using the channel-attached 3848. CUSP is no longer in service.

data A representation of facts or instructions in a form suitable for communication, interpretation, or processing by human or automatic means. Data includes constants, variables, arrays, and character strings. Any representations such as characters or analog quantities to which meaning is or might be assigned. (A)

data key or data-encrypting key
A key used to encipher, decipher, or authenticate data. Contrast with *key-encrypting key*.

Data Encryption Algorithm (DEA)
A 64-bit block cipher that uses a 64-bit key, of which 56 bits are used to control the cryptographic process and 8 bits are used for parity checking to ensure that the key is transmitted properly.

Data Encryption Standard (DES)
The National Institute of Standards and Technology Data Encryption Standard, adopted by the U.S. government as Federal Information Processing Standard (FIPS) Publication 46. which allows only hardware implementations of the data-encryption algorithm.

data translation key
A 64-bit key that protects data transmitted through intermediate systems when the originator and receiver do not share the same key.

dataset
The major unit of data storage and retrieval, consisting of a collection of data in one of several prescribed arrangements and described by control information to which the system has access.

decipher
To convert enciphered data in order to restore the original data. (T)

In computer security, to convert ciphertext into plaintext by means of a cipher system.

To convert enciphered data into clear data. Synonym for *decrypt*. Contrast with *encipher*. (D)

decode
To convert data by reversing the effect of some previous encoding. (A) (I) In the CCA products, decode and encode relate to the Electronic Code Book mode of the Data Encryption Standard (DES). Contrast with *encode* and *decipher*.

decrypt
To decipher or decode. Synonym for *decipher*. Contrast with *encrypt*.

device driver
A program that contains the code needed to attach and use a device.

device ID
In the IBM 4765 and IBM 4764 CCA implementations, a user-defined field in the configuration data that can be used for any purpose the user specifies. For example, it can be used to identify a particular device, by using a unique ID similar to a serial number.

diagnostic
Pertaining to the detection and isolation of errors in programs, and faults in equipment.

directory server
A server that manages key records in key storage by using an Indexed Sequential Access Method.

digital signature
In public key cryptography, information created by using a private key and verified by using a public key. A digital signature provides data integrity and source nonrepudiation.

Digital Signature Algorithm (DSA)
A public key algorithm for digital signature generation and verification used with the Digital Signature Standard.

Digital Signature Standard (DSS)
A standard describing the use of algorithms for digital signature purposes. One of the algorithms specified is DSA (Digital Signature Algorithm).

domain
That part of a network in which the data processing resources are under common control. (T)

double-length key
A key that is 128 bits long. A key can be either double- or single-length. A single-length key is 64 bits long.

Electronic Code Book (ECB)
A mode of operation used with block cipher cryptographic algorithms in which plain text or ciphertext is placed in the input to the algorithm and the result is contained in the output of the algorithm.

Elliptic Curve Cryptography (ECC)
A public-key process discovered independently in 1985 by Victor Miller (IBM) and Neal Koblitz (University of Washington). ECC is based on discrete logarithms. The algebraic structure of elliptic curves over finite fields makes it much more difficult to challenge at equivalent RSA key lengths.

encipher
To scramble data or to convert data to a secret code that masks the meaning of the data to unauthorized recipients. Synonym for *encrypt*. Contrast with *decipher*. See also *encode*.

enciphered data
Data whose meaning is concealed from unauthorized users or observers. See also *ciphertext*.

encode
To convert data by the use of a code in such a manner that reconversion to the original form is possible. (I) In the CCA implementation, decode and encode relate to the Electronic Code Book mode of the Data Encryption Standard. Contrast with *decode*. See also *encipher*.

encrypt
Synonym for *encipher*. (I) To convert clear text into ciphertext. Contrast with *decrypt*.

erasable programmable read-only memory (EPROM)
A type of memory chip that can retain its contents without electricity. Unlike the programmable read-only memory (PROM), which can be programmed only once, the EPROM can be erased by ultraviolet light and then reprogrammed. (E)

Eurocheque International S.C. (ECI)
A financial institution consortium that has defined three PIN-block formats.

exit routine
In the CCA products, a user-provided routine that acts as an extension to the processing provided with calls to the security API.

exit To execute an instruction within a portion of a computer program in order to terminate the execution of that portion. Such portions of computer programs include loops, subroutines, modules, and so on. (T)

A user-written routine that receives control from the system during a certain point in processing - for example, after an operator issues the START command.

exportable form
A condition a key is in when enciphered under an exporter key-encrypting key. In this form, a key can be sent outside the system to another system. A key in exportable form cannot be used in a cryptographic function.

EXPORTER key
In the CCA implementation, a type of DES key-encrypting key that can encipher a key at a sending node. Contrast with *IMPORTER key*.

exporter key-encrypting key
A 128-bit key used to protect keys sent to another system. A type of transport key.

feature
A part of an IBM product that can be ordered separately.

Federal Information Processing Standard (FIPS)
A standard published by the US National Institute of Science and Technology.

file A named set of records stored or processed as a unit. (T)

financial PIN
A Personal Identification Number used to identify an individual in some financial transactions. To maintain the security of the PIN, processes and data structures have been adopted for creating, communicating, and verifying PINs used in financial transactions. See also *Personal Identification Number*.

Flash-Erasable Programmable Read-Only Memory (flash EPROM)
A memory that has to be erased before new data can be saved into the memory.

German Bank Pool (GBP)
A German financial institution consortium that defines specific methods of PIN calculation.

hashing
An operation that uses a one-way (irreversible) function on data, usually to reduce the length of the data and to provide a verifiable authentication value (checksum) for the hashed data.

header record
A record containing common, constant, or identifying information for a group of records that follows. (D)

host In this publication, same as host computer or host processor. The machine in which the coprocessor resides. In a computer network, the computer that usually performs network-control functions and provides end-users with services such as computation and database access. (I)

importable form
A condition a key is in when it is enciphered under an importer key-encrypting key. A key is received from another system in this form. A key in importable form cannot be used in a cryptographic function.

IMPORTER key
In the CCA implementation, a type of DES key-encrypting key that can decipher a key at a receiving mode. Contrast with *EXPORTER key*.

importer key-encrypting key
A 128-bit key used to protect keys received from another system. A type of transport key.

initialize
In programming languages, to give a value to a data object at the beginning of its lifetime. (I) To set counters, switches, addresses, or contents of storage to zero or other starting values at the beginning of, or at prescribed points in, the operation of a computer routine. (A)

Integrated Cryptographic Service Facility (ICSF)
An IBM licensed program that supports the cryptographic hardware feature for the high-end System/390® processor running in a z/OS environment.

International Organization for Standardization (ISO) An organization of national standards bodies established to promote the development of standards to facilitate the international exchange of goods and services, and develop cooperation in intellectual, scientific, technological, and economic activity.

initial chaining vector (ICV)
A 64-bit random or pseudo-random value used in the cipher block chaining mode of encryption with the data encryption algorithm.

input PIN-encrypting key
A 128-bit key used to protect a PIN block sent to another system or to translate a PIN block from one format to another.

installation exit
See *exit*.

jumper
A wire that joins two unconnected circuits on a printed circuit board.

key In computer security, a sequence of symbols used with a cryptographic algorithm to encrypt or decrypt data.

key agreement
A key establishment procedure where the resultant secret keying material is a function of information contributed by two participants, so that no party can predetermine the value of the secret keying material independently from the contributions from the other parties.

key-encrypting key (KEK)
A key used for the encryption and decryption of other keys. Contrast with *data-encrypting key*. Also called a transport key.

key half
In the CCA implementation, one of the two DES keys that make up a double-length key.

key identifier
In the CCA implementation, a 64-byte variable which is either a key label or a key token.

key label
In the CCA implementation, an identifier of a key-record in key storage.

key storage
In the CCA implementation, a data file that contains cryptographic keys which are accessed by key label.

key token
In the CCA implementation, a data structure that can contain a cryptographic key, a control vector, and other information related to the key.

key output data set
A key generator utility program data set containing information about each key that the key generator utility program generates except an importer key for file encryption.

key part
A 32-digit hexadecimal value that you enter to be combined with other values to create a master key or clear key.

key part register
A register in the key storage unit that stores a key part while you enter the key part.

link The logical connection between nodes including the end-to-end control procedures. The combination of physical media, protocols, and programming that connects devices on a network. In computer programming, the part of a program, in some cases a single instruction or an address, that passes control and parameters between separate portions of the computer program. (A) (I) To interconnect items of data or portions of one or more computer programs. (I) In SNA, the combination of the link connection and link stations joining network nodes.

linkage
The coding that passes control and parameters between two routines.

LPAR mode
The central processor mode that enables the operator to allocate the hardware resources among several logical partitions.

MAC generation key
A 64-bit or 128-bit key used by a message originator to generate a message authentication code sent with the message to the message receiver.

MAC verification key
A 64-bit or 128-bit key used by a message receiver to verify a message authentication code received with a message.

magnetic tape
A tape with a magnetizable layer on which data can be stored. (T)

make file
A composite file that contains either device configuration data or individual user profiles.

master key (MK, KM)
In computer security, the top-level key in a hierarchy of key-encrypting keys.

master key register
A register in the cryptographic coprocessors that stores the master key that is active on the system.

master key variant
A key derived from the master key by use of a control vector. It is used to force separation by type of keys on the system.

MD4 Message Digest 4. A hash algorithm.

MD5 Message Digest 5. A hash algorithm.

Message Authentication Code (MAC)
A number or value derived by processing data with an authentication algorithm, The cryptographic result of block cipher operations on text or data using a Cipher Block Chaining (CBC) mode of operation, A digital signature code.

migrate
To move data from one hierarchy of storage to another. To move to a changed operating environment, usually to a new release or a new version of a system.

Modification Detection Code (MDC)
In cryptography, a number or value that interrelates all bits of a data stream so that, when enciphered, modification of any bit in the data stream results in a new MDC.

multiple encipherment
The method of encrypting a key under a double-length key-encrypting key.

National Institute of Science and Technology (NIST)
The current name for the US National Bureau of Standards.

network
A configuration of data-processing devices and software programs connected for information interchange. An arrangement of nodes and connecting branches. (I)

new master key (NMK) register
A register in the key storage unit that stores a master key before you make it active on the system.

NOCV processing
Process by which the key generator utility program or an application program encrypts a key under a transport key itself rather than a transport key variant.

node In a network, a point at which one-or-more functional units connect channels or data circuits. (I)

nonce
A time-varying value that has at most a negligible chance of repeating, such as a random value that is generated anew for each use, a timestamp, a sequence number, or some combination of these.

nonrepudiation
A method of ensuring that a message was sent by the appropriate individual.

offset The process of exclusively ORing a counter to a key.

old master key (OMK) register
A register in the key storage unit that stores a master key that you replaced with a new master key.

operational form
The condition of a key when it is encrypted under the master key so that it is active on the system.

output PIN-encrypting key
A 128-bit key used to protect a PIN block received from another system or to translate a PIN block from one format to another.

panel The complete set of information shown in a single image on a display station screen.

parameter
In the CCA security API, an address pointer passed to a verb to address a variable exchanged between an application program and the verb.

password
In computer security, a string of characters known to the computer system and a user; the user must specify it to gain full or limited access to a system and to the data stored within it.

PCI Cryptographic Coprocessor
The 4758 model 2 standard PCI-bus card supported on the field upgraded IBM S/390 Parallel Enterprise Server - Generation 5, the IBM S/390 Parallel Enterprise Server - Generation 6 and the IBM eServer zSeries.

PCI X Cryptographic Coprocessor
An asynchronous cryptographic coprocessor available on the IBM eServer zSeries 990 and IBM eServer zSeries 800.

Personal Account Number (PAN)
A Personal Account Number identifies an individual and relates that individual to an account at a financial institution. It consists of an issuer identification number, customer account number, and one check digit.

Personal Identification Number (PIN)
The 4-digit to 12-digit number entered at an automatic teller machine to identify and validate the requester of an automatic teller machine service. Personal identification numbers are always enciphered at the device where they are entered, and are manipulated in a secure fashion.

Personal Security card
An ISO-standard “smart card” with a microprocessor that enables it to perform a variety of functions such as identifying and verifying users, and determining which functions each user can perform.

PIN block
A 64-bit block of data in a certain PIN block format. A PIN block contains both a PIN and other data.

PIN generation key
A 128-bit key used to generate PINs or PIN offsets algorithmically.

PIN key
A 128-bit key used in cryptographic functions to generate, transform, and verify the personal identification numbers.

PIN offset
For 3624, the difference between a customer-selected PIN and an institution-assigned PIN. For German Bank Pool, the difference between an institution PIN (generated with an institution PIN key) and a pool PIN (generated with a pool PIN key).

PIN verification key
A 128-bit key used to verify PINs algorithmically.

plain text
Data that has nor been altered by a cryptographic process. Synonym for *clear text*. See also *ciphertext*.

Power-On Self Test (POST)
A series of diagnostic tests run automatically by a device when the power is turned on.

private key
In computer security, a key that is known only to the owner and used together with a public-key algorithm to decipher data. The data is enciphered using the related public key. Contrast with *public key*. See also *public-key algorithm*.

procedure call
In programming languages, a language construct for invoking execution of a procedure. (I) A procedure call usually includes an entry name and possible parameters.

profile
Data that describes the significant characteristics of a user, a group of users, or one-or-more computer resources.

profile ID
In the CCA implementation, the value used to access a profile within the CCA access-control system.

protocol
A set of semantic and syntactic rules that determines the behavior of functional units in achieving communication. (I) In SNA, the meanings of and the sequencing rules for requests and responses used to manage the network, transfer data, and synchronize the states of network components. A specification for the format and relative timing of information exchanged between communicating parties.

Programmed Cryptographic Facility (PCF)
An IBM licensed program that provides facilities for enciphering and deciphering data and for creating, maintaining, and managing cryptographic keys. (D)

The IBM cryptographic offering, program product 5740-XY5, using software only for encryption and decryption. This product is no longer in service.

public key
In computer security, a key that is widely known, and used with a public-key algorithm to encrypt data. The encrypted data can be decrypted only with the related private key. Contrast with *private key*. See also *public-key algorithm*.

Public Key Algorithm (PKA)
In computer security, an asymmetric cryptographic process that uses a public key to encrypt data and a related private key to decrypt data. Contrast with *Data Encryption Algorithm* and *Data Encryption Standard algorithm*. See also *Rivest-Shamir-Adleman algorithm*.

public key cryptography
In computer security, cryptography in which a public key is used for encryption and a private key is used for decryption. Synonymous with asymmetric cryptography.

Public-Key Cryptography Standards (PKCS)
Specifications produced by RSA Laboratories in cooperation with secure system developers worldwide, for the purpose of accelerating the deployment of public-key cryptography. First published in 1991.

Random access memory (RAM)
A storage device into which data are entered and from which data are retrieved in a non-sequential manner.

reason code
A value that provides a specific result as opposed to a general result. Contrast with *return code*.

record chaining
When there are multiple cipher requests and the output chaining vector (OCV) from the previous encipher request is used as the input chaining vector (ICV) for the next encipher request.

Read-only memory (ROM)
Memory in which stored data cannot be modified by the user except under special conditions.

Resource Access Control Facility (RACF)
An IBM licensed program that enables access control by identifying and verifying the users to the system, authorizing access to protected resources, logging detected unauthorized attempts to enter the system, and logging detected accesses to protected resources.

retained key
A private key that is generated and retained within the secure boundary of the PCI Cryptographic Coprocessor.

return code
A code used to influence the execution of succeeding instructions. (A) A value returned to a program to indicate the results of an operation requested by that program. In the CCA implementation, a value that provides a general result as opposed to a specific result. Contrast with *reason code*.

Rivest-Shamir-Adleman (RSA) algorithm
A process for public key cryptography that was developed by R. Rivest, A. Shamir, and L. Adleman.

RS-232
A specification that defines the interface between data terminal equipment and data circuit-terminating equipment, using serial binary data interchange.

RS-232C
A standard that defines the specific physical, electronic, and functional characteristics of an interface line that uses a 25-pin connector to connect a workstation to a communication device.

Secure Electronic Transaction
A standard created by Visa International and MasterCard for safeguarding payment card purchases made over open networks.

Secure Hash Algorithm (SHA), FIPS 180
A set of related cryptographic hash functions designed by the National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST). The first member of the family, published in 1993, is officially called SHA. However, today, it is often unofficially called *SHA-0* to avoid confusion with its successors. Two years later, SHA-1, the first successor to SHA, was published. Four more variants have since been published with increased output ranges and a slightly different design: SHA-224, SHA-256, SHA-384, and SHA-512 (all are sometimes referred to as *SHA-2*).

secure key
A key that is encrypted under a master key. When using a secure key, it is passed to a cryptographic coprocessor where the coprocessor decrypts the key and performs the function. The secure key never appears in the clear outside of the cryptographic coprocessor.

Secure Sockets Layer
A security protocol that provides communications privacy over the Internet by allowing client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery.

security
The protection of data, system operations, and devices from accidental or intentional ruin, damage, or exposure.

security server
In the CCA implementation, the functions provided through calls made to the security API.

server On a Local Area Network, a data station that provides facilities to other data stations; for example, a file server, a print server, a mail server. (A)

session
In network architecture, for the purpose of data communication between functional units, all the activities that take place during the establishment, maintenance, and release of the connection. (I) The period of time during which a user of a terminal can communicate with an interactive system (usually, the elapsed time between logon and logoff).

SHA-1 (Secure Hash Algorithm 1, FIPS 180)
A hash algorithm required for use with the Digital Signature Standard.

SHA-2 (Secure Hash Algorithm 2, FIPS 180)
Four additional variants to the SHA family, with increased output ranges and a slightly different design: SHA-224, SHA-256, SHA-384, and SHA-512 (all are sometimes referred to as SHA-2).

SHA-224
One of the SHA-2 algorithms.

SHA-256
One of the SHA-2 algorithms.

SHA-384
One of the SHA-2 algorithms.

SHA-512
One of the SHA-2 algorithms.

single-length key
A key that is 64 bits long. A key can be single- or double-length. A double-length key is 128 bits long.

smart card
A plastic card that has a microchip capable of storing data or process information.

special secure mode
An alternative form of security that allows you to enter clear keys with the key generator utility program or generate clear PINs.

string A sequence of elements of the same nature, such as characters, considered as a whole. (I)

subsystem
A secondary or subordinate system, usually capable of operating independently of, or asynchronously with, a controlling system. (I)

supervisor state
A state during which a processing unit can execute input/output and other privileged instructions. (D)

system administrator
The person at a computer installation who designs, controls, and manages the use of the computer system.

Systems Network Architecture (SNA)
An architecture that describes logical structure, formats, protocols, and operational sequences for transmitting information units through, and controlling the configuration and operation of, networks. **Note:** The layered structure of SNA allows the ultimate origins and destinations of information, that is, the end users, to be independent of and unaffected by the specific SNA network services and facilities used for information exchange.

throughput
A measure of the amount of work performed by a computer system over a given period of time; for example, number of jobs per day. (A) (I) A measure of the amount of information transmitted over a network in a given period of time; for example, a network's data-transfer-rate is usually measured in bits per second.

TLV A widely used construct, Tag, Length, Value, to render data self-identifying. For example, such constructs are used with EMV smart cards.

token In a Local Area Network, the symbol of authority passed successively from one data station to another to indicate the station is temporarily in control of the transmission medium. (I) A string of characters treated as a single entity.

Transaction Security System (TSS)
An IBM product offering including both hardware and supporting software that provides access control and basic cryptographic key-management functions in a network environment. In the workstation environment, this includes the 4755 Cryptographic Adapter, the Personal Security Card, the 4754 Security Interface Unit, the Signature Verification feature, the Workstation Security Services Program, and the AIX Security Services Program/6000. In the host environment, this includes the 4753 Network Security Processor and the 4753 Network Security Processor MVS™ Support Program.

transport key
A 128-bit key used to protect keys distributed from one system to another. A transport key can either be an exporter key-encrypting key, an importer key-encrypting key, or an ANSI key-encrypting key.

transport key variant
A key derived from a transport key by

use of a control vector. It is used to force separation by type for keys sent between systems.

Unique key per transaction (UKPT)
A cryptographic process that can be used to decipher PIN blocks in a transaction.

user-exit routine
A user-written routine that receives control at predefined user-exit points.

user ID
User identification. A string of characters that uniquely identifies a user to the system.

utility program
A computer program in general support of computer processes. (I)

verb A function that has an entry-point-name and a fixed-length parameter list. The procedure call for a verb uses the standard syntax of a programming language.

verification pattern
An 8-byte pattern calculated from the key parts you enter when you enter a master key or clear key. You can use the verification pattern to verify that you have entered the key parts correctly and specified a certain type of key.

virtual machine (VM)
A functional simulation of a computer and its associated devices. Each virtual machine is controlled by a suitable operating system. VM controls concurrent execution of multiple virtual machines on one host computer.

VISA A financial institution consortium that has defined four PIN block formats and a method for PIN verification.

VISA PIN Verification Value (VISA PVV)
An input to the VISA PIN verification process that, in practice, works similarly to a PIN offset.

3621 A model of an IBM Automatic Teller Machine that has a defined PIN block format.

3624 A model of an IBM Automatic Teller Machine that has a defined PIN block format and methods of PIN calculation.

4753 The Network Security processor. The IBM 4753 is a processor that uses the Data Encryption Algorithm and the RSA algorithm to provide cryptographic support for systems requiring secure transaction processing (and other cryptographic services) at the host computer. The NSP includes a 4755 cryptographic adapter in a workstation which is channel attached to a S/390 host computer.

4758 The IBM PCI Cryptographic processor provides a secure programming and hardware environment where DES and RSA processes are performed.

4764 IBM 4764 PCI-X Cryptographic Coprocessor.

4765 IBM 4765 PCIe Cryptographic Coprocessor.

Index

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Readers' Comments — We'd Like to Hear from You

Linux on z Systems
Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide
Version 5.0

Publication No. SC33-8294-05

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:
- Send your comments to the address on the reverse side of this form.
- Send your comments via email to: eservdoc@de.ibm.com

If you would like a response from IBM, please fill in the following information:

Name	Address
Company or Organization	
Phone No.	Email address

Readers' Comments — We'd Like to Hear from You
SC33-8294-05

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Deutschland Research & Development GmbH
Information Development
Department 3282
Schoenaicher Strasse 220
71032 Boeblingen
Germany

Fold and Tape | **Please do not staple** | Fold and Tape

SC33-8294-05

Cut or Fold
Along Line

SC33-8294-05