Linux on Z and LinuxONE

Exploiting Enterprise PKCS #11 using openCryptoki 3.15

SC34-2713-02

Note

Before using this document, be sure to read the information in “Notices” on page 81.

Edition notice

This edition applies to the EP11 host library version 3.0 which covers the latest features of openCryptoki version 3.15, and to all subsequent versions and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

Summary of changes

This edition reflects changes to the Development stream for tokens of type EP11 to be plugged-in into openCryptoki.

You can find the software package of the EP11 enablement at:

```
https://www.ibm.com/security/cryptocards/pciecc4/lonzsoftware
```

Updates for the EP11 token for openCryptoki versions 3.11 up to 3.15

The following enhancements are implemented for the EP11 enablement for openCryptoki version 3.15. For complete exploitation of the listed enhancements, the EP11 host library version 3.0 is required.

- The performance of the EP11 token is improved by the following features:
  - For single part sign- and verify-operations, as well as for single part encrypt- or decrypt-operations, the `init` call is not passed through the EP11 host library as long as there is no corresponding multi-part operation. You must explicitly enable this feature with the new option OPTIMIZE_SINGLE_PART_OPERATIONS in the EP11 token configuration file.
  - You can request to increase the performance of hash operations. Setting the new DIGEST_LIBICA option in the EP11 token configuration file causes the EP11 token to load the default libica library on initialization. For required hash operations during processing, the EP11 token then uses the libica SHA-based hash functions. These hash functions perform on the CPACF, thus avoiding hash processing on a cryptographic coprocessor and therefore avoiding I/O operations to the coprocessor.
- You can set the new USE_PRANDOM option in the EP11 token configuration file to control from where the EP11 token reads random data. When you specify USE_PRANDOM, then the token does not read random data from the random number generator of the EP11 cryptographic coprocessor. Instead, random data is read from `/dev/prandom`, or `/dev/urandom` if `/dev/prandom` is not available, .
- With GA2 of the IBM z14®, the EP11 token provides the following enhancements:
  - Support of the bit coin curve *secp256k1* is added to the EP11 token.
  - Support of the RSA OAEP mechanism CKM_RSA_PKCS_OAEP for encrypt and decrypt, as well as for wrap and unwrap operations.
  - Support of a new domain control point (access control point) related to a new BSICC2017 compliance mode. When enabled, this compliance mode disables the RSAPKCS #11 v1.5 mechanisms.
- With the availability of the IBM z15 in September 2019, the EP11 token provides the following new features:
  - SHA3 support via a vendor-specific mechanisms.
  - Support of CMAC via standard and vendor-specific mechanisms.
  - Support of the CKM_ECDH1_DERIVE mechanism according to PKCS #11 v2.4 semantics.

  With these enhancements provided as a prerequisite in the EP11 host library version 3.0, you can additionally exploit the following features of the EP11 token with the availability of the IBM z15 in November 2019:

  - Support of the RSA OAEP mechanism with SHA2 and SHA3 as hashing algorithms and mask generation function (MGF) algorithm is available.
  - New IBM®-specific mechanisms are provided for the support of elliptic curve cryptography (ECC). With these, you can use Edwards Curves *ed25519* and *ed448* for EdDSA and Montgomery curves *curve25519* and *curve448* for ECDH.

  - New domain (access) control points are implemented to control elliptic curve cryptography, to allow data key generation and import for protected keys, and to enable the use of the post-quantum Dilithium signature algorithm.
- Function `C_DigestKey` now always returns CKR_FUNCTION_NOT_SUPPORTED since the EP11 library does no longer support it.
- The EP11 token is getting ready for post-quantum cryptography:
  - You can use the quantum safe CRYSTALS-Dilithium Digital Signature Algorithm for generating keys and for signing and verifying digital signatures.
  - You can import and transport externally generated Dilithium keys.
- openCryptoki now implements the *PKCS #11 version 3.0 Baseline Provider* specification. A library implementing PKCS #11 according to the **Baseline Provider Clause** as described in PKCS #11 Cryptographic Token Interface Profiles Version 3.0 is called a *PKCS #11 version 3.0 Baseline Provider*. Such a library can be exploited by an application conforming to the **Baseline Consumer Clause** described in the same document. Such applications are in turn called *PKCS #11 version 3.0 Baseline Consumers*.
- A new vendor-specific function called `C_IBM_ReencryptSingle` is introduced into openCryptoki and is supported by all tokens. Data that is already encrypted with a specific key and mechanism can be re-encrypted with this function, using a different key and mechanism. For secure key encryption with an EP11 token or a CCA token, the data is never visible in the clear anywhere outside the cryptographic coprocessor.
- You can use the **`pkcstok_migrate`** utility to transform an EP11 token, a CCA token, an ICA token, or a Soft Token created with any version of openCryptoki into a data format that was generated by FIPS compliant operations. This new data format can be used with openCryptoki version 3.12 or later. However, also for version 3.12 or later, the old non-compliant format is the default. Being FIPS compliant, the token data is stored in a format that is better protected against attacks than the previously used data format without FIPS compliance.

Updates for the openCryptoki version 3.10 EP11 token type

The following enhancements are implemented for the EP11 enablement for openCryptoki version 3.10:

- All mechanisms provided by the EP11 token type that are defined in PKCS #11 v2.40 are now supported. See Table 2 on page 48 for a complete list of supported mechanisms for the EP11 token.

  However, the EP11 token still supports the CKM_ECDH1_DERIVE mechanism according to PKCS #11 v2.20 (without key derivation function (KDF) and shared data).
- You can activate one or two session modes to limit the access to cryptographic objects in order to improve security. The available session modes are the *strict session mode* or the *virtual HSM (VHSM) mode*.

  In *strict session mode*, for each new session, a unique EP11 session ID is generated. This prevents that a session key (if copied from a session) will be handled as a valid key by the EP11 crypto adapter even after the PKCS#11 session that generated the key has ended.

  In *virtual HSM (VHSM) mode*, you can restrict keys to only that token that was used to generate it.

  You can configure an EP11 token to use either one of the available modes, or both.
- You can now set an option that makes CK_TRUE the default value for the CKA_SENSITIVE attribute when generating, unwrapping, or building secret keys with `C_CreateObject`. This eliminates a restriction with using EP11 library functions from earlier versions.
- The list of mechanisms returned by the `C_GetMechanismList` function is now filtered according to the (domain) control points configured in the cryptographic coprocessor.
- You can now import keys of type CKK_DH, CKK_DSA, and CKK_EC. See "Importing keys" on page 52 for more information.
- The error handling when using EP11 library functions is enhanced:

  - User-friendly messages are issued into the SYSLOG during token initialization, when no CK_SESSION_INFO structure is available for providing meaningful reason codes.
  - The use of return codes is adapted to better comply to the PKCS #11 standard.
- openCryptoki now supports multiple token instances of the same token type. This edition documents what to do to exploit this support for multiple tokens of type EP11.
- Starting with EP11 library version 2.0, as the default, the TKE uses the *ep11TKEd* daemon to authenticate with a Linux® user who is member of a new *ep11tke group*, which is created during EP11 package installation.
- New tools are described in Chapter 7, "Tools and utilities," on page 63:
  - **`ep11info`** provides information about EP11 cryptographic coprocessors and about configured domains.
  - **`pkcsep11_session`** allows to delete an EP11 session from EP11 cryptographic coprocessors left over by programs that did not terminate normally.

About this document

Linux on Z applications that are using a PKCS #11 API can take advantage of the Enterprise PKCS #11 (EP11) coprocessor mode of an IBM cryptographic adapter (Crypto Express), starting with a CEX4P coprocessor.

EP11 is a stack architecture for using a library of standard cryptographic functions to write applications on IBM Z® mainframes with cryptographic hardware. In this document, the term CEX*P either stands for CEX4P, CEX5P, or CEX6P.

This publication describes how to work with the EP11 token type. With the support of multiple token instances of a certain token type, this publication uses the term EP11 token or EP11 token instance (or just token or token instance) for tokens of type EP11.

You can find the latest version of this document on the developerWorks® website at:

www.ibm.com/developerworks/linux/linux390/documentation_dev.html

and on the IBM Knowledge Center at:

www.ibm.com/support/knowledgecenter/linuxonibm/liaaf/lnz_r_lib.html

How this document is organized

The information is divided into topics that describe installing, configuring, and using the EP11 library.

Chapter 1, "Introduction," on page 1 contains general information about the Linux on Z EP11 enablement.

Chapter 2, "The EP11 crypto stack," on page 3 describes how the components of the Linux on Z EP11 enablement are positioned within the different layers between applications and hardware.

Chapter 3, "Building the EP11 crypto stack," on page 9 describes how to prepare or install the EP11 components within the stack.

Chapter 4, "Configuring openCryptoki for EP11 support," on page 37 describes the configuration and customization tasks for enabling the exploitation of the EP11 library functions from applications.

Chapter 5, "Using an EP11 token," on page 47 describes the APIs for invoking the EP11 library functions.

Chapter 6, "Troubleshooting EP11," on page 61 provides information how to resolve problems when using the Linux on Z EP11 enablement.

Chapter 7, "Tools and utilities," on page 63 documents various tools and utilities which provide general information about EP11, which support you in migrating EP11 master keys, and which help you to manage EP11 sessions.

Chapter 8, "Programming examples for openCryptoki," on page 69 is a set of programming samples that use the EP11 library.

Terminology

The following terms for cryptographic coprocessors are used in this document:

CEX4P
An IBM 4765 Crypto Express4 feature (CEX4S), configured in EP11 coprocessor mode.

CEX5P
An IBM 4767 Crypto Express5 feature (CEX5S), configured in EP11 coprocessor mode.

CEX6P
An IBM 4768 Crypto Express6 feature (CEX6S), configured in EP11 coprocessor mode.

CEX7P
An IBM 4769 Crypto Express7 feature (CEX7S), configured in EP11 coprocessor mode.

CEX*P
Designates either one or more or all of CEX4P, CEX5P, CEX6P, or CEX7P. Also, CEX*S designates one or more or all of CEX4S, CEX5S, CEX6S, or CEX7S.

Who should read this document

This document is intended for C programmers who want to access IBM Z hardware support for cryptographic methods. It is also intended for system administrators who need to enable and configure the required cryptographic hardware.

Furthermore, this publication addresses users who want to enhance their existing openCryptoki applications with the new features of Enterprise PKCS #11.

Distribution independence

This publication does not provide information that is specific to a particular Linux distribution.

The tools it describes are distribution independent.

Other publications for Linux on Z and LinuxONE

You can find publications for Linux on Z and LinuxONE on IBM Knowledge Center.

These publications are available on IBM Knowledge Center at www.ibm.com/support/knowledgecenter/linuxonibm/liaaf/lnz_r_lib.html

- *Device Drivers, Features, and Commands*
- *Using the Dump Tools*
- *KVM Virtual Server Quick Start*, SC34-2753
- *KVM Virtual Server Management*, SC34-2752
- *How to use FC-attached SCSI devices with Linux on z Systems®*, SC33-8413
- *Introducing IBM Secure Execution for Linux*, SC34-7721
- *libica Programmer's Reference*, SC34-2602
- *Exploiting Enterprise PKCS #11 using openCryptoki*, SC34-2713
- *Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide*, SC33-8294
- *Pervasive Encryption for Data Volumes*, SC34-2782
- *How to set an AES master key*, SC34-7712
- *Troubleshooting*, SC34-2612
- *Kernel Messages*, SC34-2599
- *How to Improve Performance with PAV*, SC33-8414
- *How to Set up a Terminal Server Environment on z/VM*, SC34-2596

Chapter 1. Introduction

The Linux on Z Enterprise PKCS #11 (EP11) enablement allows applications to use a PKCS #11 API to run secure key cryptographic operations on an IBM Crypto Express adapter that is configured as an Crypto Express EP11 coprocessor. The CEX4S adapter card is the first Crypto Express adapter which can be configured as an EP11 coprocessor.

The Linux on Z EP11 enablement comprises several components that need to be installed and configured within certain locations of the EP11 stack as described in Chapter 2, “The EP11 crypto stack,” on page 3.

An application's request is first submitted to a PKCS #11 API, implemented by the openCryptoki library and the EP11 token. From this token, the request is propagated to the Crypto Express EP11 coprocessor. The request is then processed on this coprocessor. The resulting output is finally returned to the application across the involved interfaces.

The EP11 cryptography architecture offers a secure key infrastructure.

This introduction provides information about the standard software that is used in this implementation and about the used Crypto Express EP11 coprocessor (shortly referred to as CEX*P, which stands for any type of a Crypto Express EP11 coprocessor).

What is PKCS #11?

The Public-Key Cryptography Standards (PKCS) comprise a group of cryptographic standards that provide guidelines and application programming interfaces (APIs) for the usage of cryptographic methods. As the name PKCS suggests, these standards put an emphasis on the usage of public key (that is, asymmetric) cryptography.

PKCS #11 is a cryptographic token interface standard, which specifies an API, called *Cryptoki*. With this API, applications can address cryptographic devices as tokens and can perform cryptographic functions as implemented by these tokens. This standard, first developed by the RSA Laboratories in cooperation with representatives from industry, science, and governments, is now an open standard lead-managed by the *OASIS PKCS 11 Technical Committee*.

It follows an object-based approach, addressing the goals of technology independence (any kind of HW device) and resource sharing. It also presents to applications a common, logical view of the device that is called a cryptographic *token*. PKCS #11 assigns a slot ID to each token. An application identifies the token that it wants to access by specifying the appropriate slot ID.

For more information about PKCS #11, refer to this URL:

`PKCS #11 Cryptographic Token Interface Standard`

What is openCryptoki?

openCryptoki is an open source implementation of the *Cryptoki* API defined by the PKCS #11 Cryptographic Token Interface Standard. Thus, openCryptoki provides support for several cryptographic algorithms according to the industry-wide PKCS #11 standards. The openCryptoki library loads the so called tokens that provide hardware or software specific support for cryptographic functions.

The EP11 token extends the openCryptoki token library. It uses special hardware cryptographic functions that are provided by an IBM Crypto Express adapter (starting with CEX4S), which is configured by a certain firmware (see “Enabling a cryptographic coprocessor for EP11 firmware exploitation” on page 11).

openCryptoki can be used directly through the openCryptoki shared library (C API).

For more information about the openCryptoki services, or about the interfaces between the openCryptoki main module and its tokens, see

```
https://github.com/opencryptoki/opencryptoki.
```

You can also read topic “openCryptoki overview” on page 5 for an introduction of the openCryptoki main features.

What is a Crypto Express EP11 coprocessor ?

An IBM Crypto Express adapter, which is configured with the Enterprise PKCS #11 (EP11) firmware, is called a Crypto Express EP11 coprocessor (shortly referred to as CEX*P). The Crypto Express4 adapter is the first adapter that can be configured as an EP11 coprocessor (CEX4P). You can also use CEX5Ps or CEX6Ps on the appropriate IBM Z systems.

The CEX*P adapters provide hardware-accelerated support for crypto operations that are based on the PKCS #11 Cryptographic Token Interface Standard. Access from applications to the functions of a CEX*P adapter is enabled through the EP11 stack. This EP11 stack consists of certain EP11 user space libraries and an EP11 extension in the Linux AP device driver. Using several layers of interfaces, the PKCS #11 standard requests are propagated to and returned from the CEX*P adapter by the device driver.

A CEX*P adapter is a *hardware security module (HSM)* that maintains and protects secrets (for example, master keys) such that these secrets cannot be revealed from outside the adapter: No operating system service or application can retrieve these secrets and any trial to physically break into the card destroys its data due to its tamper proof design.

A CEX*P adapter supports cryptographic operations with secure keys. A *secure key* is a key that is encrypted (wrapped) by a master key that is stored in the adapter. So sometimes, a master key is also referred to with the more general term *wrapping key*, as for example in document Enterprise PKCS#11 (EP11) Library structure.

Therefore, on the CEX*P adapter, applications can decrypt (unwrap) a secure key and use it for cryptographic operations inside the adapter. Outside the adapter (for example, inside an operating system), a secure key is only available as a binary large object (blob) wrapped by the master key, and cannot be used for cryptographic operations. To use a secure key, an application must call functions on the CEX*P adapter. It is therefore safe to keep a secure key in memory or to store it in a file system.

Cryptographic keys that are not encrypted are called *clear keys*. If a clear key is stored in memory or in a file, unauthorized access to that memory or file must carefully be prevented. Otherwise, the key can be stolen and used to decrypt protected information. The CEX*P adapters do not support clear key cryptography.

The maximum number of supported domains depends on the mainframe model and is the same for all Crypto Express EP11 coprocessors in that mainframe. For example, an IBM z14 (z14) supports up to 85 domains (with hexadecimal domain IDs 0000 to 0054). Each domain acts like an EP11 coprocessor, but maintains its own master key. That means, that the master key of one domain cannot be accessed by another domain. Different domains of a crypto adapter may be assigned to different LPARs or z/VM guests, such that multiple LPARs or guests can share one Crypto Express EP11 coprocessor without sharing their master keys.

Chapter 2. The EP11 crypto stack

The EP11 crypto stack for Linux on Z consists of various components within the different layers: An application sends a request down to the hardware (cryptographic adapter), via the device driver and the firmware. The request is routed all the layers down and back again, and the request result is returned to the application. The stack thus provides an end-to-end solution for cryptographic operations.

For example, an application sends an encryption request to the crypto adapter. Through various interfaces, such a request is propagated from the application layer down to the target crypto adapter hardware. On its way down, the request passes through the involved layers: the standard crypto interfaces, the IBM Z crypto libraries, and the operating system kernel. The zcrypt device driver finally sends the request to the Crypto Express EP11 coprocessor. The resulting request output is sent back to the application just the other way round through the layer interfaces.

Figure 1 on page 4 illustrates the EP11 crypto stack within the Linux on Z environment. The components that make up the Linux on Z EP11 enablement are highlighted:

- the EP11 token within openCryptoki
- the host part of the EP11 library (located in user space, which is named `libep11.so`). In version 2.0, `libep11.so` is a symbolic link to the versioned library `libep11.so.2` and this in turn is a symbolic link to `libep11.so.2.0.0`.
- the EP11 extension of the zcrypt device driver. This extension was included with kernel level 3.14 on https://www.kernel.org/. Note that distributions sometimes back-port features from newer *kernel.org* kernels into their current kernel versions. Therefore check with your distribution partner, whether your distribution release supports the EP11 enablement, if its kernel version is older than 3.14.
- the module part of the EP11 library, that is, the EP11 firmware that is installed on the Crypto Express EP11 coprocessor adapter hardware.

openCryptoki can be used directly through the openCryptoki shared library (C API).

openCryptoki supports several token types, which can offer different functionality for different hardware devices or software solutions. Tokens of type EP11 (aka EP11 tokens) interact with the host part of the EP11 library. EP11 can operate with the Crypto Express adapter (CEX*P) with EP11 firmware load for processing cryptographic functions.

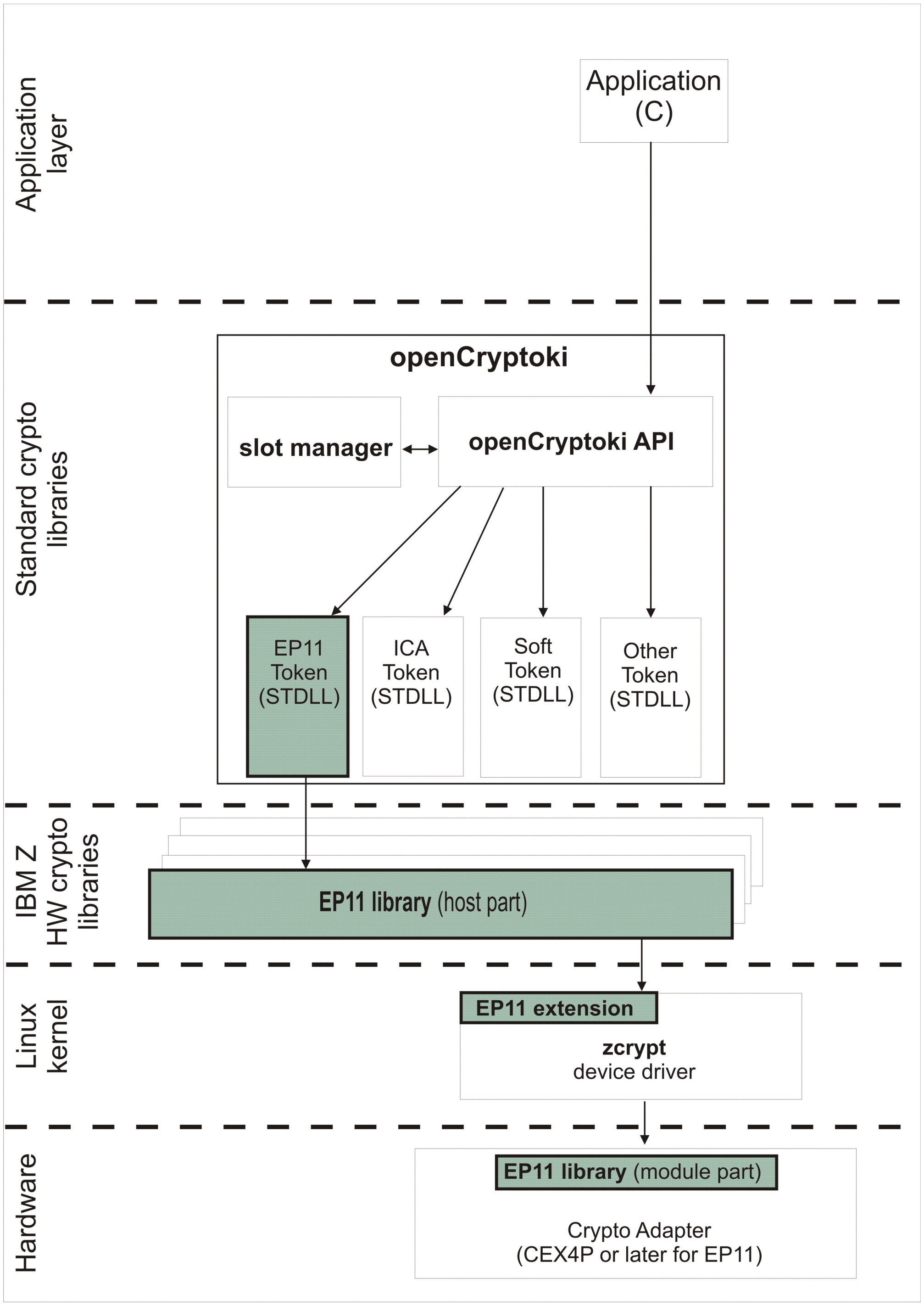


Figure 1. Stack and process flow with a configured EP11 token

The EP11 token itself does not implement PKCS #11 but provides services for accessing EP11 functions to openCryptoki. For a description of these services or the interface between the common part of openCryptoki and its tokens, see the openCryptoki documentation. Once the EP11 token is configured, cryptographic functions from the EP11 token are available to an application through the PKCS #11 API provided by the common openCryptoki code. The EP11 token itself accesses the EP11 library. The EP11 library is divided into the host part and the module part, which runs in the Crypto Express EP11 coprocessor. An installed EP11 library is a prerequisite for enabling openCryptoki to use the EP11 token. The EP11 library passes requests to the CEX*P EP11 coprocessor through the zcrypt device driver of Linux on Z.

The host part of the EP11 library creates cryptographic requests from the EP11 token in Abstract Syntax Notation One (ASN.1). These requests are sent to and understood by the CEX*P adapter. The host part also converts response buffers that are received from the adapter into data structures that are expected by the EP11 token. The EP11 token makes these APIs accessible to openCryptoki and thus the applications, but does not implement any cryptographic mechanism. The mechanisms available and their parameters depend on the EP11 implementation (EP11 library and CEX*P card) and its configuration. The PKCS #11 Cryptographic Token Interface Standard defines methods for inquiring available mechanisms. You can obtain an inquiry of all available mechanisms and their parameters using the PKCS #11 functions `C_GetMechanismList` and `C_GetMechanismInfo`.

Besides the CEX*P adapter that is loaded with the EP11 firmware (EP11 module part), the EP11 token furthermore requires a zcrypt device driver within the kernel, extended with the Linux on Z EP11 enablement support (see "Installing and loading the cryptographic device driver" on page 16). In addition, the EP11 token requires the availability of the host part of the EP11 library.

Therefore, check the following dependencies:

- **Dependencies on distributors:** Distributors build the openCryptoki RPM and DEB packages that comprise the EP11 support (EP11 token) for delivering them to customers. Generally the distributors provide two packages, one library package and one development package. See also "Installing openCryptoki " on page 34.
- **Dependencies on hardware:** The EP11 library functions run on the IBM zEnterprise EC12 (zEC12) processor family (processor types 2827-H20, -H43, -H66, -H89, -HA1) or follow-on processors with an IBM Crypto Express4S (CEX4S) or follow-on adapter.

Note: In the remainder of this publication, the terms EP11 or Linux on Z EP11 enablement stand for the entirety of the implementation components that consists of the EP11 token, the EP11 extension of the *zcrypt* device driver, and the EP11 library (host part and module part) as shown in Figure 1 on page 4.

openCryptoki overview

openCryptoki consists of an implementation of the PKCS #11 API, a slot manager, an API for slot token dynamic link libraries (STDLLs), and a set of STDLLs (or tokens). The EP11 token is a new STDLL introduced with openCryptoki version 3.1.

The openCryptoki base library (`libopencryptoki.so`) provides the generic API as outlined in the PKCS #11 specification (version 2.20). This library also loads token-specific modules (STDLLs) that provide the token specific implementation of the PKCS #11 API and cryptographic functions (for example, session management, object management, and crypto algorithms). For a description of the PKCS #11 version 2.20 standard, refer to the following URL: PKCS #11 Cryptographic Token Interface Standard

A global configuration file (`/etc/opencryptoki/opencryptoki.conf`) is provided which describes the available tokens. This configuration file can be customized for the individual tokens. The openCryptoki package contains man pages that describe the format of the configuration files. For more information, see "Adjusting the openCryptoki configuration file" on page 37.

The EP11 token is a plug-in into the openCryptoki token library, providing support for several cryptographic algorithms.

Slot manager

The slot manager (**`pkcsslotd`**) runs as a daemon. Upon startup, it creates a shared memory segment and reads the openCryptoki configuration file to acquire the available token and slot information. The openCryptoki API attaches to this memory segment to retrieve token information. Thus, the slot manager provides the openCryptoki API with the token information when required. An application in turn links to or loads the openCryptoki API.

Slot token dynamic link libraries (STDLLs)

The EP11 token is an example of an STDLL within openCryptoki. STDLLs are plug-in modules to the openCryptoki (main) API. They provide token-specific functions that implement the interfaces. Specific devices can be supported by building an appropriate STDLL. Figure 1 on page 4 illustrates the stack and the process flow in an IBM Z environment.

The STDLLs require local disk space to store persistent data, such as token information, personal identification numbers (PINs) and token objects. This information is stored in a separate directory for each token (per default in `/var/lib/opencryptoki/ep11tok` for the EP11 token). Within each of these directories there is a sub-directory `TOK_OBJ` that contains the token objects (token key store). Each private token object is represented by an encrypted file. Most of these directories are created during installation of openCryptoki.

The `pkcsconf` command line program

openCryptoki provides a command line program (`/usr/lib/pkcs11/methods/pkcsconf`) to configure and administer tokens that are supported within the system. The `pkcsconf` capabilities include token initialization, and security officer (SO) PIN and user PIN initialization and maintenance (see also "Initializing EP11 tokens" on page 45).

`pkcsconf` operations that address a specific token must specify the slot that contains the token with the **`-c`** option. You can view the list of tokens present within the system by specifying the **`-t`** option (without **`-c`** option). For example, the following code shows the options for the `pkcsconf` command and displays slot information for the system:

```
# pkcsconf ?
usage: pkcsconf [-itsmlIupPh] [-c slotnumber -U user-PIN -S SO-PIN -n new PIN]
```

The available options have the following meanings:

-i
: display PKCS11 info

-t
: display token info

-s
: display slot info

-m
: display mechanism list

-l
: display slot description

-I
: initialize token

-u
: initialize user PIN

-p
: set the user PIN

-P
: set the SO PIN

-h | --help | ?
: show this help

-c
: specify the token slot for the operation

-U
: the current user PIN (for use when changing the user pin with -u and -p options); if not specified, user will be prompted

-S
: the current Security Officer (SO) pin (for use when changing the SO pin with -P option); if not specified, user will be prompted

-n

the new pin (for use when changing either the user pin or the SO pin with -u, -p or -P options); if not specified, user will be prompted

For more information about the `pkcsconf` command, see the `pkcsconf` man page.

Chapter 3. Building the EP11 crypto stack

The components of the Linux on Z EP11 enablement must be embedded into an infrastructure of hardware and software cryptographic components. In this environment, applications can start the provided functions by using the PKCS #11 openCryptoki API. This infrastructure is referred to as EP11 stack.

To enable the EP11 hardware cryptographic function support on IBM Z mainframes, you must prepare some hardware components. You must also install and load specific driver modules and libraries, configure and start daemons, and set up your system environment.

Building this EP11 stack comprises several subtasks that are described in the following topics:

- "Preparing the Crypto Express EP11 coprocessor" on page 9
- "Installing and loading the cryptographic device driver" on page 16
- "Installing the host part of the EP11 library" on page 17
- "Setting a master key on the Crypto Express EP11 coprocessor" on page 19
- "Installing openCryptoki " on page 34
- "Adding EP11 tokens to openCryptoki" on page 39

Preparing the Crypto Express EP11 coprocessor

To take advantage of the hardware-accelerated support for crypto operations from a CEX*P adapter, you must switch the CEX*S adapter into the CEX*P mode. This modification enables the installed and required EP11 firmware (module part of the EP11 library) to run on this adapter.

The required information is presented in the following subtopics:

- "Purpose of domains" on page 9
- "Assigning adapters and domains to LPARs" on page 11
- "Enabling a cryptographic coprocessor for EP11 firmware exploitation" on page 11
- "Assigning EP11 adapters as dedicated adapters to z/VM guests" on page 15
- "Restriction to extended evaluations" on page 59

Purpose of domains

When you configure your system on the Support Element (SE), you can specify how a logical partition (LPAR) uses coprocessors and accelerators. In this context, the Crypto Express cards support a concept of cryptographic domains. Each domain is protected by a master key, thus preventing access across domains and effectively separating the contained keys.

For information on how to configure domains, refer to *zEnterprise System Support Element Operations Guide*, which you can download from the IBM Resource Link.

There are two types of access to a cryptographic domain:

- for usage of cryptographic functions
- for management (control) of the domain, which includes the management of the master keys

A domain, which is assigned to an LPAR for usage access is called a usage domain of that LPAR. A domain, which is assigned to an LPAR for management (control) access is called a control domain of that LPAR. Every domain, which is a usage domain of an LPAR must also be a control domain of that LPAR, but not the other way round.

Usage domains

A logical partition's *usage domains* are domains in the coprocessors that can be used for cryptographic functions.

In Linux, you can use the **`lszcrypt -b`** command to find out which usage domain is configured for that Linux system:

```
$ lszcrypt -b

ap_domain=0x1a
ap_max_domain_id=0x54
ap_interrupts are enabled
config_time=30 (seconds)
poll_thread is disabled
poll_timeout=250000 (nanoseconds)
```

Control domains

A logical partition's *control domains* are those cryptographic domains for which remote secure administration functions can be established and administered from this logical partition.

This logical partition's control domains must include its usage domains. So for each index that is selected in the *Usage domain index* list, you must select the same index in the *Control domain index* list.

But a logical partition's control domains can also include the control domains of other logical partitions. Assigning multiple logical partitions' control domains as control domains of a single logical partition allows using the partition to perform administrative functions from the TKE .

If you are using the Integrated Cryptographic Service Facility (ICSF) from z/OS, select at least one control domain with its matching usage domain. Refer to the ICSF documentation for information about ICSF basic operations.

If you are using a Trusted Key Entry (TKE) workstation to manage cryptographic keys, you can define your TKE host and the control domains for a logical partition. See "Setting a master key on the Crypto Express EP11 coprocessor" on page 19 for more information.

Control domain exposure

For configuration and management purposes the TKE needs to know which control domains are configured on the system.

In Linux, use a `sysfs` attribute called `ap_control_domain_mask` in `/sys/bus/ap/` to display the configured control domains. This information is set automatically from the device driver.

The attribute `ap_control_domain_mask` is read-only and contains a 32-byte field in hexadecimal notation, representing the installed control domain facilities. Each bit position represents a dedicated control domain. Thus, a maximum number of 256 domains could be addressed.

Example:

cat /sys/bus/ap/ap_control_domain_mask
0x0004000000000000000000000000000000000000000000000000000000000000

Byte
    Meaning

1
    domain 0-7

2
    domain 8-15

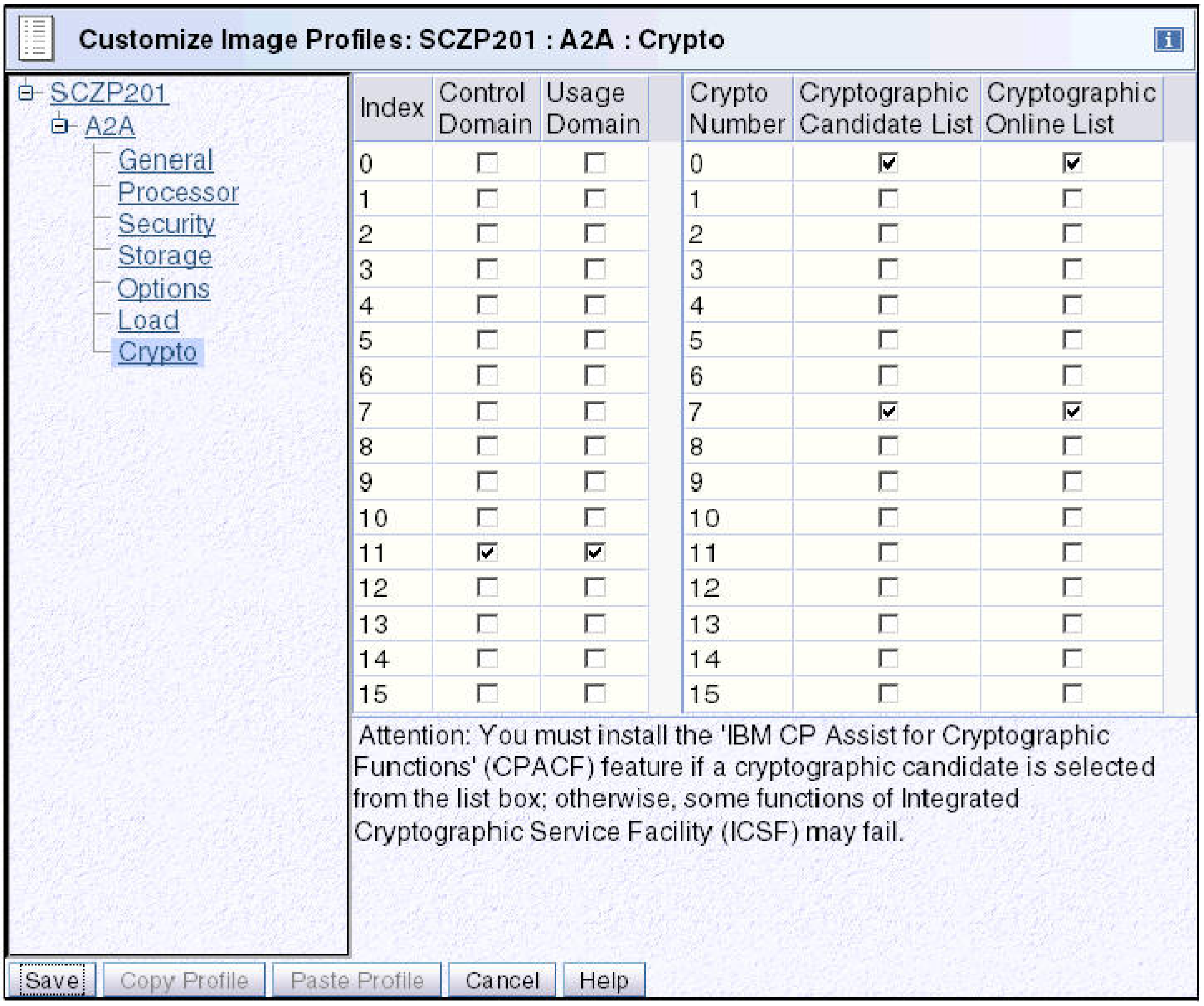
...
    ...

In this example, the control domain 13 was configured.

Assigning adapters and domains to LPARs

After you set up the Crypto Express adapter in the Support Element, you must allow access to it from your LPAR. You achieve this by using the Hardware Management Console (HMC) or the Support Element (SE).

You can define a certain LPAR to use a domain (or multiple domains) as a usage domain and as a control domain, or as a control domain only. You can retrieve this information from the Support Element. Each adapter supports 16 domains (see Figure 2 on page 11). The selected domains apply to all selected adapters. For a more detailed information about planning the cryptographic configuration, see *IBM System z10 Enterprise Class Configuration Setup, SG24-7571*.



Customize Image Profiles: SCZP201 : A2A : Crypto

Index	Control Domain	Usage Domain	Crypto Number	Cryptographic Candidate List	Cryptographic Online List
0	□	□	0	☑	☑
1	□	□	1	□	□
2	□	□	2	□	□
3	□	□	3	□	□
4	□	□	4	□	□
5	□	□	5	□	□
6	□	□	6	□	□
7	□	□	7	☑	☑
8	□	□	8	□	□
9	□	□	9	□	□
10	□	□	10	□	□
11	☑	☑	11	□	□
12	□	□	12	□	□
13	□	□	13	□	□
14	□	□	14	□	□
15	□	□	15	□	□

Attention: You must install the 'IBM CP Assist for Cryptographic Functions' (CPACF) feature if a cryptographic candidate is selected from the list box; otherwise, some functions of Integrated Cryptographic Service Facility (ICSF) may fail.

Figure 2. Cryptographic configuration for LPAR A2A

In Figure 2 on page 11, LPAR A2A is defined to use and control the cryptographic domain number 11. It is also allowed to access the crypto adapters numbers 0 and 7. They are brought online if they are present in the system, if the LPAR is activated, and if the **zcrypt** device driver is loaded.

Linux kernels earlier than version 4.9 can only use one crypto domain at a given time. In that case, if the LPAR contains multiple domains, the kernel selects the default domain. Also, if for these kernel versions you want to use a different default domain, you need to specify this domain as a parameter when loading the *ap* main module of the zcrypt device driver.

Enabling a cryptographic coprocessor for EP11 firmware exploitation

You must have an IBM 4765 Crypto Express4 feature or higher that is configured as an EP11 coprocessor, and that is initialized and personalized in your z/VM guest or LPAR. Read this topic to learn how to check for the existence of a suitably configured CEX*P adapter (starting with CEX4P, or higher), and how to configure this adapter if it is missing yet.

About this task

A CEX*S Crypto Express card configured in the Enterprise PKCS #11 coprocessor mode (or shortly EP11 coprocessor mode) is also called a Crypto Express EP11 coprocessor (CEX*P). Such a coprocessor, which is installed in your z/VM guest or LPAR, is a prerequisite for using the functions of the EP11 library. This procedure shows you how to configure a CEX*S Crypto Express adapter into a CEX*P adapter by enabling the installed EP11 firmware from the Support Element.

Procedure

1. Check whether you have already plugged in and enabled your CEX*S Crypto Express card, and validate your model and type configuration (accelerator or coprocessor).

   To check, enter the **`lszcrypt`** command and check the output:

   ```
   # lszcrypt
   CARD.DOMAIN TYPE  MODE        STATUS  REQUESTS
   ----------------------------------------------
   00          CEX5A Accelerator online         0
   00.001a     CEX5A Accelerator online         0
   01          CEX5C CCA-Coproc  online        50
   01.001a     CEX5C CCA-Coproc  offline       50
   02          CEX6C CCA-Coproc  online        55
   02.001a     CEX6C CCA-Coproc  offline       55
   03          CEX6P EP11-Coproc online         8
   03.001a     CEX6P EP11-Coproc online         8
   05          CEX7P EP11-Coproc online       104
   05.001a     CEX7P EP11-Coproc online       104
   ```

   If you see the output as shown, with an output line similar to

   ```
   xx.xxxx     CEX6P EP11-Coproc online
   ```

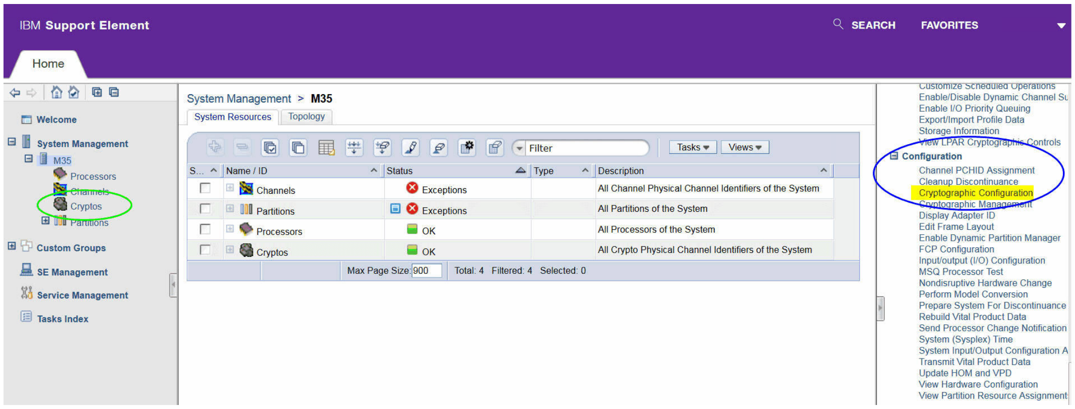
   then an CEX6P adapter is available and ready for use with EP11 and the task is completed.
2. If the following error message is displayed, the zcrypt device driver module must be installed.

   ```
   error - cryptographic device driver zcrypt is not loaded!
   ```

   For installation information, refer to "Installing and loading the cryptographic device driver" on page 16.
3. If the output from the **`lszcrypt`** command in step 1 does not show one of **`CEX<n>P`**, (where **`<n>`** can be 4, or higher), then check the reason why this happened. If a CEX*S card is correctly assigned to the LPAR or z/VM guest, where the Linux is running in, but none of **`CEX<n>P`** is shown, then you must activate the EP11 firmware on the CEX*S adapter.

   For this purpose, log on to the Support Element with a user ID granted the appropriate access rights. You can either go directly to the Support Element, or you can use its web interface.
4. In the **System Management** window, select the CPC that holds the CEX*S adapter that you want to configure.

   In the sample screen from Figure 3 on page 13, the selected CPC is *M35*.

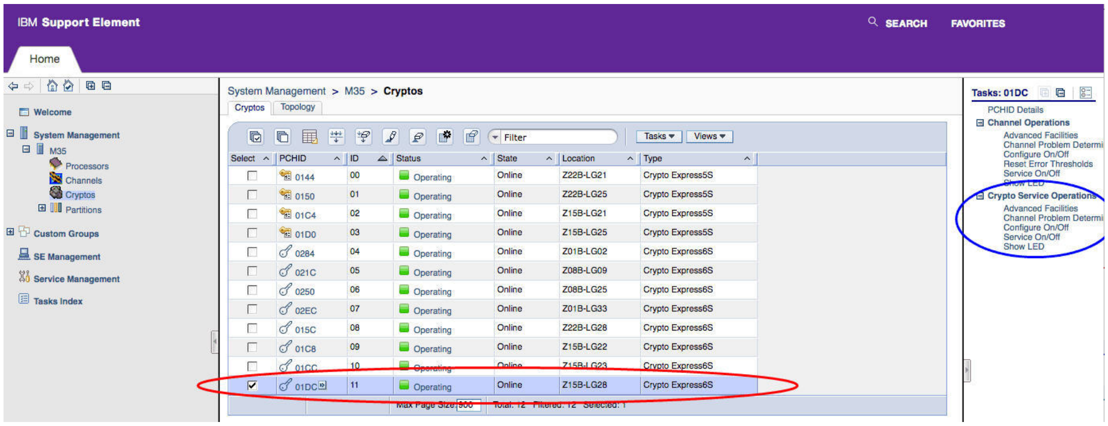

IBM Support Element
SEARCH
FAVORITES
Home
Welcome
System Management
M35
Processors
Channels
Cryptos
Partitions
Custom Groups
SE Management
Service Management
Tasks Index
System Management > M35
System Resources
Topology
Filter
Tasks
Views
Customize Scheduled Operations
Enable/Disable Dynamic Channel Su
Enable I/O Priority Queuing
Export/Import Profile Data
Storage Information
View LPAR Cryptographic Controls
Configuration
Channel PCHID Assignment
Cleanup Discontinuance
Cryptographic Configuration
Cryptographic Management
Display Adapter ID
Edit Frame Layout
Enable Dynamic Partition Manager
FCP Configuration
Input/output (I/O) Configuration
MSQ Processor Test
Nondisruptive Hardware Change
Perform Model Conversion
Prepare System For Discontinuance
Rebuild Vital Product Data
Send Processor Change Notification
System (Sysplex) Time
System Input/Output Configuration A
Transmit Vital Product Data
Update HOM and VPD
View Hardware Configuration
View Partition Resource Assignment

S...	Name / ID	Status	Type	Description
□	Channels	Exceptions		All Channel Physical Channel Identifiers of the System
□	Partitions	Exceptions		All Partitions of the System
□	Processors	OK		All Processors of the System
□	Cryptos	OK		All Crypto Physical Channel Identifiers of the System

Max Page Size: 900 Total: 4 Filtered: 4 Selected: 0

Figure 3. System Management in the Support Element

5. Select **Cryptos** from the navigation area on the left of the dialog to get a list of installed adapters as shown in Figure 4 on page 13.

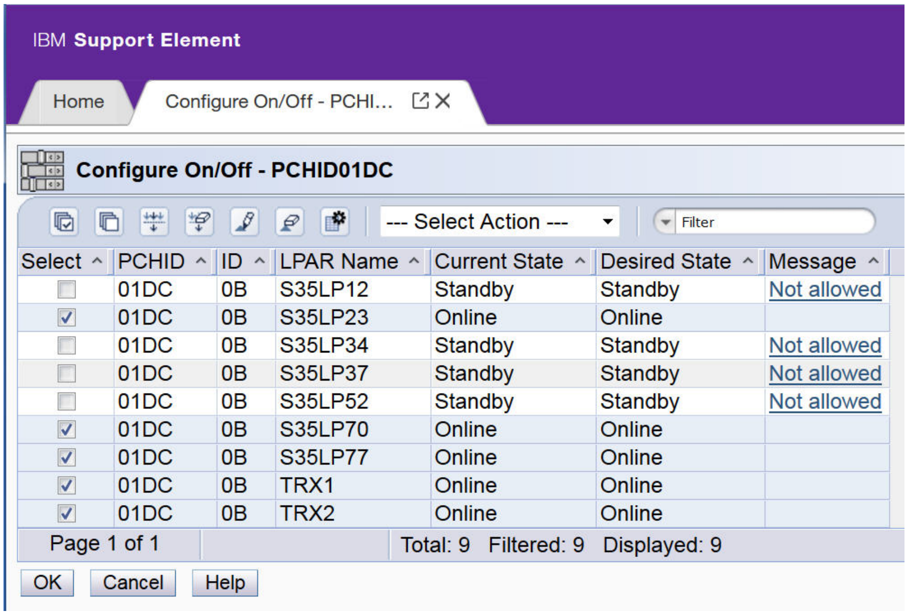


Select	PCHID	ID	Status	State	Location	Type
□	0144	00	Operating	Online	Z22B-LG21	Crypto Express5S
□	0150	01	Operating	Online	Z22B-LG25	Crypto Express5S
□	01C4	02	Operating	Online	Z15B-LG21	Crypto Express5S
□	01D0	03	Operating	Online	Z15B-LG25	Crypto Express5S
□	0284	04	Operating	Online	Z01B-LG02	Crypto Express6S
□	021C	05	Operating	Online	Z08B-LG09	Crypto Express6S
□	0250	06	Operating	Online	Z08B-LG25	Crypto Express6S
□	02EC	07	Operating	Online	Z01B-LG33	Crypto Express6S
□	015C	08	Operating	Online	Z22B-LG28	Crypto Express6S
□	01C8	09	Operating	Online	Z15B-LG22	Crypto Express6S
□	01CC	10	Operating	Online	Z15B-LG23	Crypto Express6S
☑	01DC	11	Operating	Online	Z15B-LG28	Crypto Express6S

Max Page Size: 900 Total: 12 Filtered: 12 Selected: 1

Figure 4. System Management - installed crypto adapters

6. Select the crypto card to be changed - in our scenario, a CEX6 coprocessor with PCHID 01DC and ID 11 - and also select **Configure On/Off** from the **Crypto Service Operations** to reach the view shown in Figure 5 on page 13.



Configure On/Off - PCHID01DC

Select	PCHID	ID	LPAR Name	Current State	Desired State	Message
□	01DC	0B	S35LP12	Standby	Standby	Not allowed
☑	01DC	0B	S35LP23	Online	Online	
□	01DC	0B	S35LP34	Standby	Standby	Not allowed
□	01DC	0B	S35LP37	Standby	Standby	Not allowed
□	01DC	0B	S35LP52	Standby	Standby	Not allowed
☑	01DC	0B	S35LP70	Online	Online	
☑	01DC	0B	S35LP77	Online	Online	
☑	01DC	0B	TRX1	Online	Online	
☑	01DC	0B	TRX2	Online	Online	

Page 1 of 1 Total: 9 Filtered: 9 Displayed: 9

Figure 5. System Management - configure LPARs off

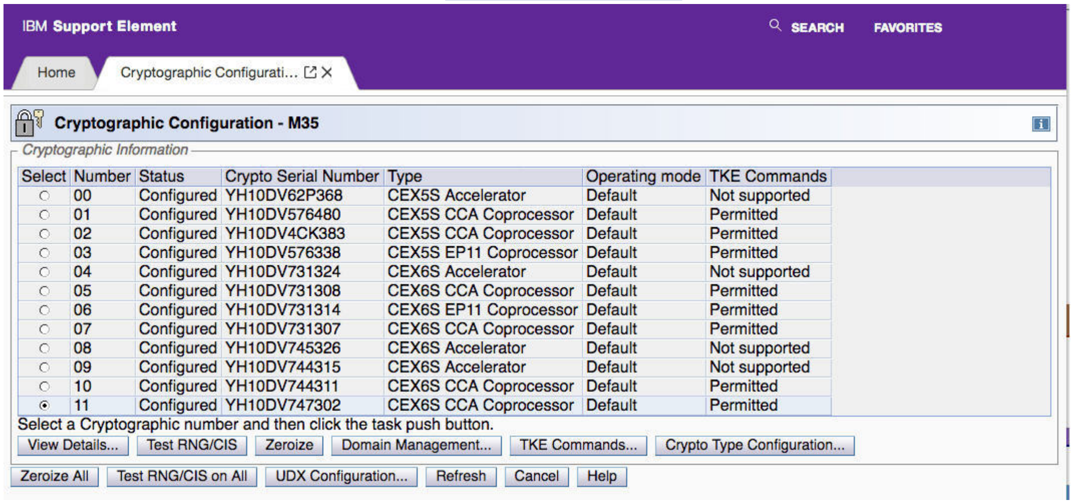
7. Select all LPARs, where this adapter is configured online (if any), as shown in Figure 5 on page 13.

The Crypto Express adapter must be configured offline in all LPARs, before you can change the configuration type. For this purpose, specify **`Toggle`** from the **Select Action** pull-down to toggle to the desired state and then press **OK** to apply the change. In the next dialog, you need to confirm your intended action, because this could be disruptive for processes from the affected LPARs.

Finally, you return to the view shown in Figure 4 on page 13. You see the selected adapter stopped now.

8. Navigate back to the **System Management** window (Figure 3 on page 13). Now scroll down and select **Cryptographic Configuration** from the **Configuration** menu on the right hand side.

   This leads you to the figure shown in Figure 6 on page 14.



Cryptographic Configuration - M35

Cryptographic Information

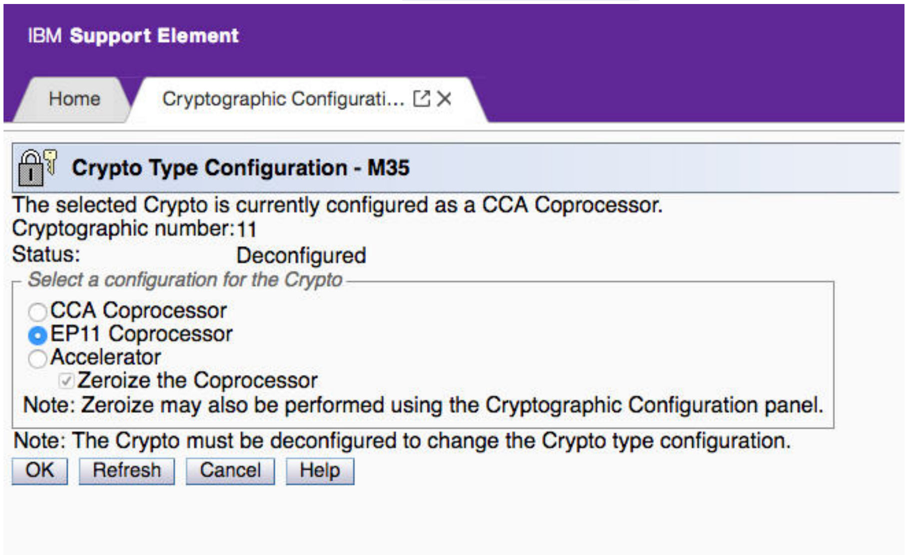
Select	Number	Status	Crypto Serial Number	Type	Operating mode	TKE Commands
○	00	Configured	YH10DV62P368	CEX5S Accelerator	Default	Not supported
○	01	Configured	YH10DV576480	CEX5S CCA Coprocessor	Default	Permitted
○	02	Configured	YH10DV4CK383	CEX5S CCA Coprocessor	Default	Permitted
○	03	Configured	YH10DV576338	CEX5S EP11 Coprocessor	Default	Permitted
○	04	Configured	YH10DV731324	CEX6S Accelerator	Default	Not supported
○	05	Configured	YH10DV731308	CEX6S CCA Coprocessor	Default	Permitted
○	06	Configured	YH10DV731314	CEX6S EP11 Coprocessor	Default	Permitted
○	07	Configured	YH10DV731307	CEX6S CCA Coprocessor	Default	Permitted
○	08	Configured	YH10DV745326	CEX6S Accelerator	Default	Not supported
○	09	Configured	YH10DV744315	CEX6S Accelerator	Default	Not supported
○	10	Configured	YH10DV744311	CEX6S CCA Coprocessor	Default	Permitted
●	11	Configured	YH10DV747302	CEX6S CCA Coprocessor	Default	Permitted

Select a Cryptographic number and then click the task push button.

Figure 6. System Management - Cryptographic Configuration

9. Select the desired adapter again (see step 4).

   Now press **Crypto Type Configuration** from the dialog shown in Figure 6 on page 14. This selection brings you to the dialog shown in Figure 7 on page 14.



Crypto Type Configuration - M35

The selected Crypto is currently configured as a CCA Coprocessor.
Cryptographic number: 11
Status: Deconfigured

Select a configuration for the Crypto

○ CCA Coprocessor
● EP11 Coprocessor
○ Accelerator
☑ Zeroize the Coprocessor

Note: Zeroize may also be performed using the Cryptographic Configuration panel.

Note: The Crypto must be deconfigured to change the Crypto type configuration.

Figure 7. System Management - Cryptographic Configuration

10. Select **EP11 Coprocessor** and press **OK**.

    This action makes the adapter to become a CEX*P adapter that is upgraded with the EP11 firmware. Also note, that TKE commands are always permitted for a CEX*P adapter, so that it can communicate with the TKE daemon *ep11TKEd*.

11. You must now select those LPARs that you want to allow to access and use the reconfigured adapter. For these LPARs, you need to configure back online the reconfigured adapter.

Therefore, go to the dialog shown in Figure 5 on page 13, now toggling the status of the adapter for the LPAR back to online.

12. A restart of z/VM or the LPAR is required to activate the reconfiguration.

    For z/VM, check before, that the correct definitions have been applied to the EP11 coprocessor card. Also for the LPARs, on z/VM and on Linux, you must add the reconfigured adapter to the activation profile. Now deactivate and activate the LPAR. Then perform an IPL of Linux on that LPAR, respectively perform an IPL of z/VM and then start the guests using the reconfigured adapter.
13. Optionally, you can use the **chzcrypt** command to enable (online state) and disable (offline state) the IBM crypto adapter:

    ```
    $ chzcrypt -e 0x06    // set card06 online
    $ chzcrypt -d 0x06    // set card06 offline
    ```

    For more information about the IBM crypto adapter, see *Device Drivers, Features, and Commands*, SC33-8411 available at

    ```
    www.ibm.com/developerworks/linux/linux390/documentation_dev.html
    ```

Results

Now that the EP11 firmware has been enabled on your cryptographic coprocessor, this card turned into a so called CEX*P coprocessor which can take advantage of the Linux on Z EP11 enablement. To check the capability of a configured adapter, you can use the following **lszcrypt -c <card-number>** command:

```
$ lszcrypt -c 03
card03 provides capability for:
EP11 Secure Key
```

Notes:

1. If you work with the available session modes (strict session mode or virtual HSM mode) as described in "Controlling access to cryptographic objects" on page 56, a unique EP11 session ID is generated for each session and is stored as a pin-blob (binary large object) on the coprocessor domain. A Crypto Express EP11 coprocessor offers storage for up to 1024 nonces or pin-blobs (binary large objects), shared among all defined domains on the coprocessor.
2. If multiple EP11 cryptographic coprocessors in your environment are configured with different levels of the EP11 firmware (module part of the EP11 library), then the EP11 token only provides those features that the lowest CEX*P EP11 coprocessor provides.

Assigning EP11 adapters as dedicated adapters to z/VM guests

On a z/VM guest, you can authorize the user to define virtual cryptographic facilities and provide the guest access to the AP queues on the PCI cryptographic cards. You achieve this with the help of the CRYPTO directory statement using the **DOMAIN** and **APDEDicated** operands.

The **DOMAIN** operand specifies up to 16 domains the virtual machine may use. The **APDEDicated** operand specifies up to 64 APs the virtual machine may use for dedicated access to the Adjunct Processor (AP) cryptographic facility. You can specify as many CRYPTO statements as you need to assign domains or APs to the virtual machine.

You can use the z/VM CP command **QUERY CRYPto DOMains** to request the display of the status of the cryptographic hardware and of installed AP domains.

Note: The **CRYPTO APVIRTual** directory statement cannot be used with the EP11 enablement.

For more information, see also *z/VM CP Planning and Administration* and *z/VM CP Commands and Utilities Reference* from the IBM Knowledge Center.

Installing and loading the cryptographic device driver

You need an installed Linux kernel that includes the cryptographic device driver. This cryptographic device driver is normally included in the regular kernel package shipped with your Linux distribution. Loading the cryptographic device driver is only required for earlier installations as described in this topic.

About this task

In earlier Linux distributions, the cryptographic device driver 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.

There are distributions using kernel levels starting with 4.10, that have basic cryptographic device driver support as part of the kernel (that is, the **ap** module is already compiled in the kernel). In this case, loading the **ap** main module with the **modprobe** command is no longer needed. In addition, the **domain** and **poll_thread** parameters are no longer module parameters, but kernel parameters. In this case, you can change the values directly via sysfs, or change as kernel parameters. Refer to the *Device Drivers, Features, and Commands* for kernel 4.12 or later on the developerWorks website for further information.

Procedure

1. For installations with a loadable cryptographic device driver, use the **lsmod** command to find out if either the **z90crypt** or the **ap** module is already loaded.
2. 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=

   Use an integer that identifies the default cryptographic domain for the Linux instance. You define cryptographic domains in the LPAR activation profile on the HMC or SE. The default value (**domain=autoselect**) causes the device driver to choose one of the available domains automatically.

   Important: Be sure to enter an existing domain. The Trusted Key Entry workstation does not find the cryptographic adapters if a non-existing domain is entered here.

   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.

   If the cryptographic device driver is part of the kernel, you cannot unload it. In this case, you can directly edit domain settings via sysfs.

   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 earlier than z10.

   For Linux instances that run in LPAR mode on a 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.

Results

The zcrypt device driver that contains the EP11 extension is loaded and **lszcrypt** displays the cryptographic adapters available to the Linux system.

Installing the host part of the EP11 library

Read the contained information about how to install the host part of the EP11 library as a component of the EP11 stack.

About this task

As a part of the EP11 stack, you need to install the host part of the EP11 library on your IBM Z, as shown in Figure 1 on page 4.

Also, to use the EP11 functionality, the TKE daemon (*ep11TKEd*) must be available and running to perform certain communication tasks. This communication path is necessary, for example, for the initial key personalization or for key updates (see also "Setting a master key on the Crypto Express EP11 coprocessor" on page 19).

Procedure

1. Obtain the appropriate EP11 software package for use on IBM Z mainframe servers, that contains the Linux on Z EP11 enablement from the software package selection page:

   ```
   https://www.ibm.com/security/cryptocards/pciecc3/lonzsoftware.shtml
   ```

   RPM is the installation package format for Red Hat Enterprise Linux and SUSE Linux Enterprise Server distributions. DEB is the package format for the Ubuntu distribution. The names of the packages are as follows:

   - `ep11-host-2.0.0-2.s390x.rpm` or later is the standard RPM package that provides libraries (`libep11.so`) and tools (for example, the **`ep11info`** tool) to configure and use a CEX*P EP11 coprocessor.
   - `libep11_2.0.0-2_s390x.deb` or later is the equivalent Ubuntu package.
   - `ep11-host-devel-2.0.0-2.s390x.rpm` or later is the development RPM package which is required if you want to develop programs that link to the EP11 library.
   - `libep11-dev_2.0.0-2_s390x.deb` or later is the equivalent Ubuntu package.

   To see a complete list of files contained in the packages, you can download the associated `RELEASE.txt` file from the software-package selection page.

   Note: The host part of the EP11 library is developed and maintained by IBM and therefore not part of any commercial Linux distribution.

2. Install the RPM or DEB by issuing one of the following commands:

   ```
   rpm -Uvh <rpm_packet>   /* for RPM new installation or updates*/
   dpkg -i <deb_packet>    /* for DEB new installation or updates*/
   ```

3. The EP11 TKE daemon (ep11TKEd), which comes along with the standard RPM or DEB packages obtained in step "1" on page 17 is also installed during the installation. It is required and must be running for handling administrative commands and for managing communication between the TKE workstation and the CEX*P EP11 coprocessor.

What to do next

Starting with EP11 library version 2.0, the TKE can use the *ep11TKEd* daemon to authenticate with a Linux user who is member of the *ep11tke group* which is defined in `/etc/group` of the system. This is the default, and it is recommended not to change this.

However, you could disable the authentication in the *ep11TKEd* configuration file as described hereafter.

The *ep11TKEd* daemon uses the Linux pluggable authentication modules (PAM) subsystem to authenticate the user. The interaction with PAM can also be configured in a *ep11TKEd*-specific PAM configuration file.

Software requirements: As of EP11 software package 2.0.0, the EP11 TKE daemon requires the OpenSSL library version 1.0.x for secure authentication with the TKE. It also requires the PAM standard modules for the authentication process. Refer to your Linux distribution documentation for supported versions of OpenSSL.

The *ep11TKEd* daemon uses **`systemd`** for daemonizing and logging. If you do not use **`systemd`**, you need to do the daemonizing and routing of log messages to files yourself.

Only TKE versions equal or greater than 8.0 are supported with this version of *ep11TKEd*.

Security notes: The *ep11TKEd* daemon typically runs as the nobody user. For the authentication process, *ep11TKEd* needs privileges to access the shadow file. For those cases, *ep11TKEd* can be a **`setGID`** program which uses the shadow group, or a **`setUID to root`** program to gain access to the file.

The *ep11TKEd* daemon uses these privileges of the shadow group or the read or search capability only through a small window of a running authentication process. Privileges are permanently dropped, if authentication is disabled in the configuration file.

In the host package installation process, Linux is checked for its capabilities. If the shadow group is found, then the sticky bit for the shadow group is set. If the group is not found, the sticky bit for the root user is set.

If supported by your Linux distribution, **`AppArmor`** rules are installed. If they are not already enforced for the *ep11TKEd* daemon, you can enforce them manually.

Configuration:

Note: It is recommended to use the default settings. The configuration features described hereafter may be used in special environments.

- ***Configuration files:***
  - The EP11 TKE daemon can be configured in file `/etc/ep11/ep11tked.conf`.

    The only allowed option is **`CipherMode`**. The two allowed values are *AES* and *None*. *AES* is the default value.

    AES
    Use the Linux PAM system to authenticate a user.

    None
    Do not use any authentication.

    Note: If possible, *AES* should always be used!
  - The authentication process can be configured in the file `/etc/pam.d/ep11tked`. See the PAM module manuals for help on editing this file. Be careful when changing this file as it involves the risk of rendering the authentication useless.

    The default setting is to allow any user that has a password configured and is member of the *ep11tke group* to gain access through the *ep11TKEd* daemon.
- ***How to control the daemon:***

  The program can be started manually by executing the file `/usr/sbin/ep11TKEd`.

  This starts *ep11TKEd* in the running shell and not as a daemon. Log messages are printed to the console. This is sometimes useful for troubleshooting, but usually *ep11TKEd* should be started through **`systemd`**:

```
systemctl start ep11TKEd
```

  To automatically start the daemon during boot use the following command:

```
systemctl enable ep11TKEd
```

  To disable the automatic start use the following command:

```
systemctl disable ep11TKEd
```

See the **systemd** documentation for help with the service manager. When using **systemd** for controlling the daemon, log messages are written to the **systemd** journal. See the **journald** manual for more information.

Restrictions:

The versions of the *ep11TKEd* daemon delivered with an EP11 host library starting with version 2.0 cannot be used on an IBM zEnterprise EC12 (zEC12) system. On these zEC12 systems, you must use version 1.x of the EP11 package together with the contained *ep11TKEd* daemon.

Setting a master key on the Crypto Express EP11 coprocessor

To generate a secure and secret master key, use the TKE workstation that is connected to the IBM Z mainframe.

This publication outlines a selection of the basic steps for creating and initializing EP11 smart cards and for generating a master key. It does not document the complete process of setting up a comprehensive security concept, nor does it demonstrate all security features available from the TKE workstation. For information about sophisticated features, for example, for a dual control security policy, for the zone concept, or for using TKE domain groups, refer to the *Trusted Key Entry Workstation User's Guide* from the IBM Resource Link.

Note: The EP11 master key set on a CEX*P adapter is referred to as *wrapping key* in document Enterprise PKCS#11 (EP11) Library structure.

Trusted Key Entry (TKE) is a priced optional feature that is used for managing IBM Z cryptographic coprocessors in a customer environment. Cryptographic coprocessors operate with a master key that is located inside the coprocessor itself. These coprocessors use keys that are protected by being encrypted (wrapped) with the master key. These wrapped keys are called secure keys and are only decrypted inside the coprocessor's secure enclosure.

Information is provided in the following topics:

- "Setting up the TKE environment" on page 19
- "Create and initialize an EP11 smart card" on page 20
- "Creating a master key on the TKE workstation" on page 23

For more information about these tasks, refer to topics *Using the Crypto Module Notebook to administer EP11 crypto modules* and *Smart Card Utility Program (SCUP)* in the *z/OS Cryptographic Services ICSF Trusted Key Entry Workstation*.

Setting up the TKE environment

For a Crypto Express EP11 coprocessor, a TKE workstation is required to perform certain key management functions.

A TKE version 7.3 is required to detect EP11 adapters and set and manage wrapping keys (master keys) correctly.

Note: For any master key transactions to the card (key generation or import) and for initialization/ personalization purposes, you need at least two smart card readers. Furthermore, the described outline uses one CA (Certificate Authority) smart card and two smart cards that hold two separate key parts which make up the master key. The smart cards can be initialized from scratch by using the TKE interfaces.

To use the EP11 functions of the TKE, the EP11 library (*libep11.so*) and the TKE daemon (*ep11TKEd*) must be installed. The *ep11TKEd* daemon is used for receiving messages from a Trusted Key Entry workstation and for routing those messages to the specified Crypto Express EP11 coprocessor (CEX*P card).

To start the daemon, use the command

```
service ep11TKEd start
```

This command is redirected and performs the same processing as the command

```
systemctl start ep11TKEd.service
```

The *ep11TKEd* TKE daemon uses the EP11 host library to communicate with a CEX*P EP11 coprocessor and is listening on port 50004 for administrative TKE commands. These commands are translated into **`ioctl`** commands to talk to the zcrypt device driver.

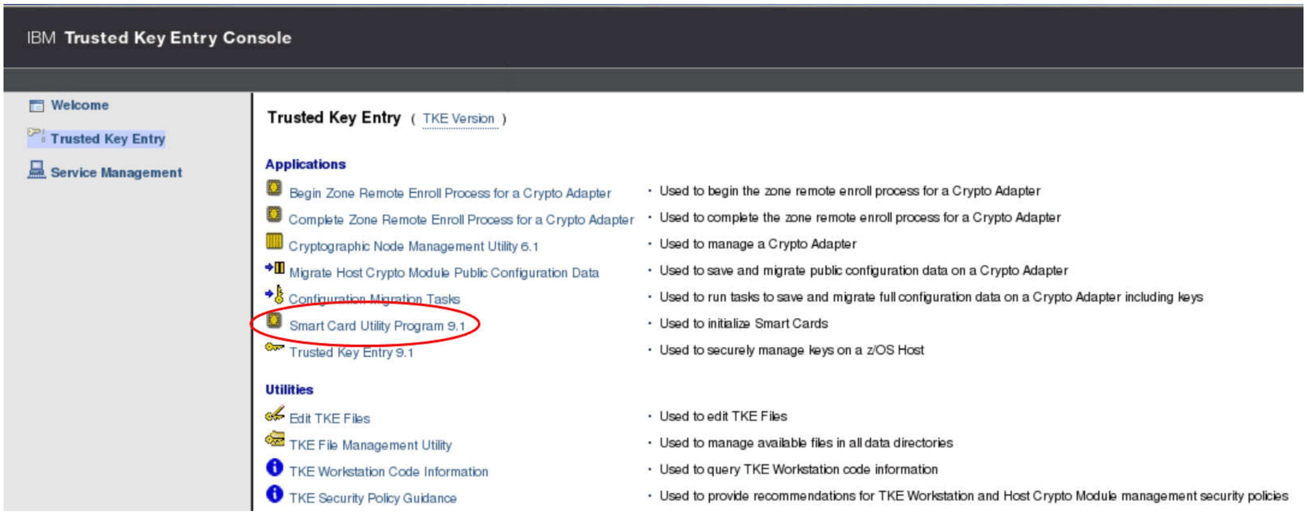
Create and initialize an EP11 smart card

Step 1

As a prerequisite, you need a valid CA (Certificate Authority) smart card to be authorized to create EP11 smart cards (see **Step 4**).

The **Trusted Key Entry** console automatically loads on start-up with a set of commonly used tasks. After the TKE console started, the initial **Trusted Key Entry Console** window appears.

This initial window provides access to applications and utilities available on the TKE workstation.



Trusted Key Entry (TKE Version)

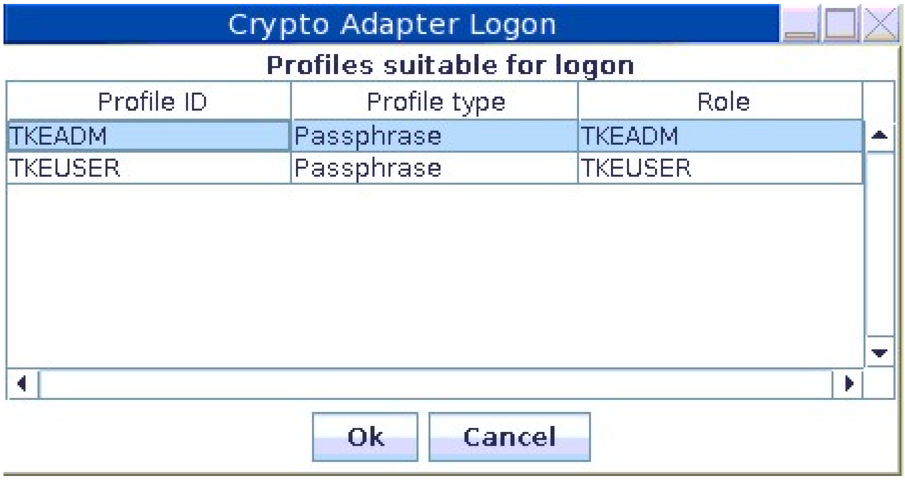
Applications	
Begin Zone Remote Enroll Process for a Crypto Adapter	• Used to begin the zone remote enroll process for a Crypto Adapter
Complete Zone Remote Enroll Process for a Crypto Adapter	• Used to complete the zone remote enroll process for a Crypto Adapter
Cryptographic Node Management Utility 6.1	• Used to manage a Crypto Adapter
Migrate Host Crypto Module Public Configuration Data	• Used to save and migrate public configuration data on a Crypto Adapter
Configuration Migration Tasks	• Used to run tasks to save and migrate full configuration data on a Crypto Adapter including keys
Smart Card Utility Program 9.1	• Used to initialize Smart Cards
Trusted Key Entry 9.1	• Used to securely manage keys on a z/OS Host
Utilities	
Edit TKE Files	• Used to edit TKE Files
TKE File Management Utility	• Used to manage available files in all data directories
TKE Workstation Code Information	• Used to query TKE Workstation code information
TKE Security Policy Guidance	• Used to provide recommendations for TKE Workstation and Host Crypto Module management security policies

Figure 8. TKE Console - initial window

Step 2

Click the **Smart Card Utility Program** application as shown in Figure 8 on page 20.

When you open a TKE application or utility, you must sign on with a profile that is on the TKE workstation crypto adapter. Therefore, depending on how you have initialized your environment, the **Crypto Adapter Logon** window is displayed with profile IDs that represent a single or group passphrase logon. The individual or group profile you choose must have enough authority to do the functions that are performed by the application or utility. The steps described here use the default TKEADM user name.



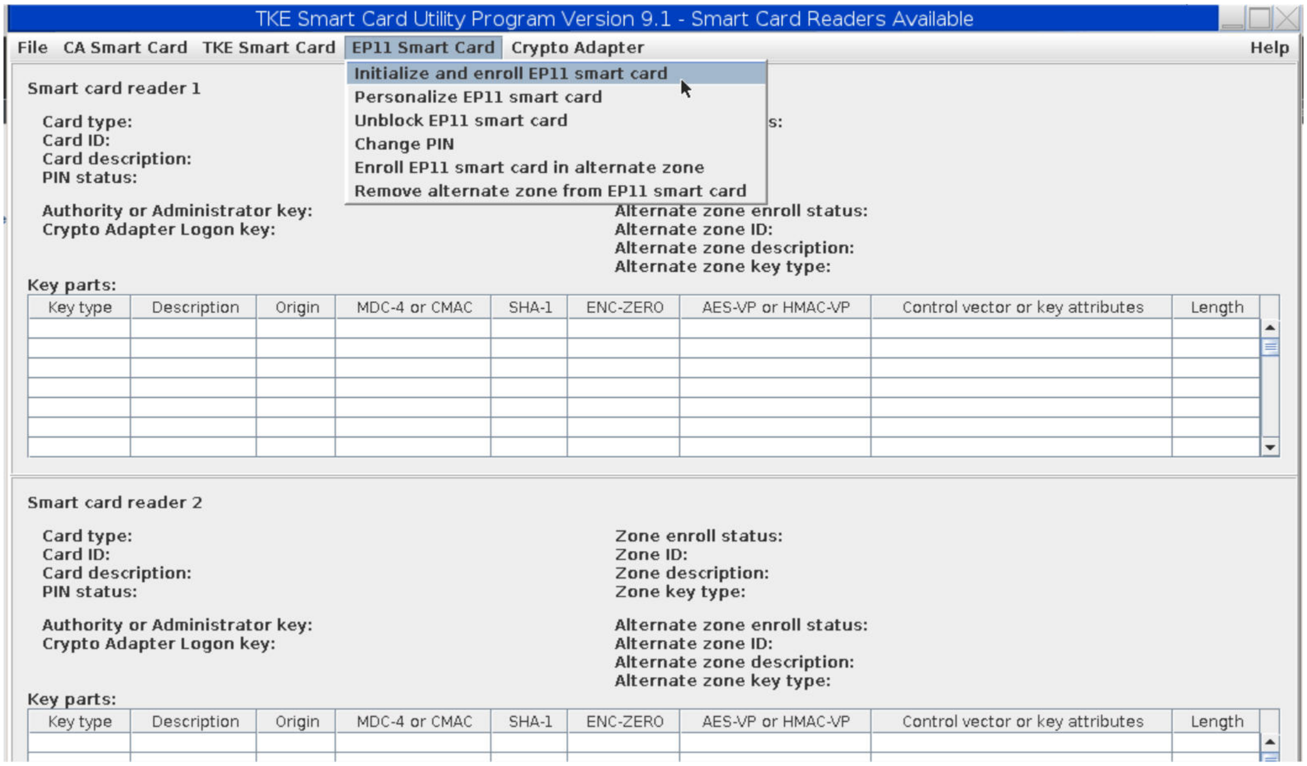
Profile ID	Profile type	Role
TKEADM	Passphrase	TKEADM
TKEUSER	Passphrase	TKEUSER

Figure 9. Crypto Adapter Logon

Step 3

After a successful log-on, the **Smart Card Utility Program** opens and shows a table for each smart card reader for all detected plugged-in smart card types. The tables are still empty at this point in time, because the EP11 smart card is not yet created. The missing information is provided during the process of initializing and personalizing the smart card as described in the remainder of this topic.

To continue, select **Initialize and enroll EP11 smart card** from the **EP11 Smart Card** pulldown choice.


TKE Smart Card Utility Program Version 9.1 - Smart Card Readers Available
File
CA Smart Card
TKE Smart Card
EP11 Smart Card
Crypto Adapter
Help
Initialize and enroll EP11 smart card
Personalize EP11 smart card
Unblock EP11 smart card
Change PIN
Enroll EP11 smart card in alternate zone
Remove alternate zone from EP11 smart card

Smart card reader 1

Card type:
Card ID:
Card description:
PIN status:

Authority or Administrator key:
Crypto Adapter Logon key:

s:

Alternate zone enroll status:
Alternate zone ID:
Alternate zone description:
Alternate zone key type:

Key parts:

Key type	Description	Origin	MDC-4 or CMAC	SHA-1	ENC-ZERO	AES-VP or HMAC-VP	Control vector or key attributes	Length

Smart card reader 2

Card type:
Card ID:
Card description:
PIN status:

Authority or Administrator key:
Crypto Adapter Logon key:

Zone enroll status:
Zone ID:
Zone description:
Zone key type:

Alternate zone enroll status:
Alternate zone ID:
Alternate zone description:
Alternate zone key type:

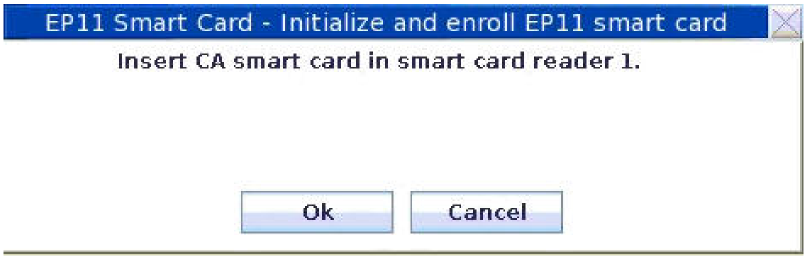
Key parts:

Key type	Description	Origin	MDC-4 or CMAC	SHA-1	ENC-ZERO	AES-VP or HMAC-VP	Control vector or key attributes	Length

Figure 10. Initialize and enroll EP11 smart card

Step 4

The **Smart Card Utility Program** prompts you to insert the CA smart card into the smart card reader 1 and then press the **OK** button. For detailed information, read the TKE documentation.

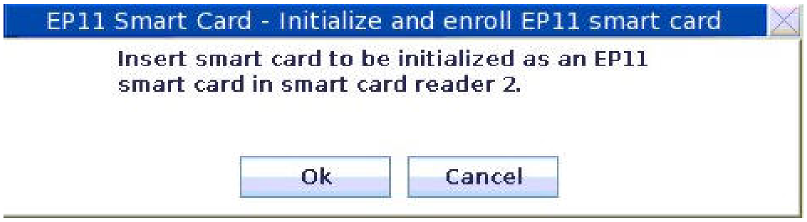

EP11 Smart Card - Initialize and enroll EP11 smart card

Insert CA smart card in smart card reader 1.

Figure 11. Insert CA smart card

Step 5

As next step, the **Smart Card Utility Program** prompts you to insert a smart card to be initialized as an EP11 smart card into smart card reader 2 and then press the **OK** button.

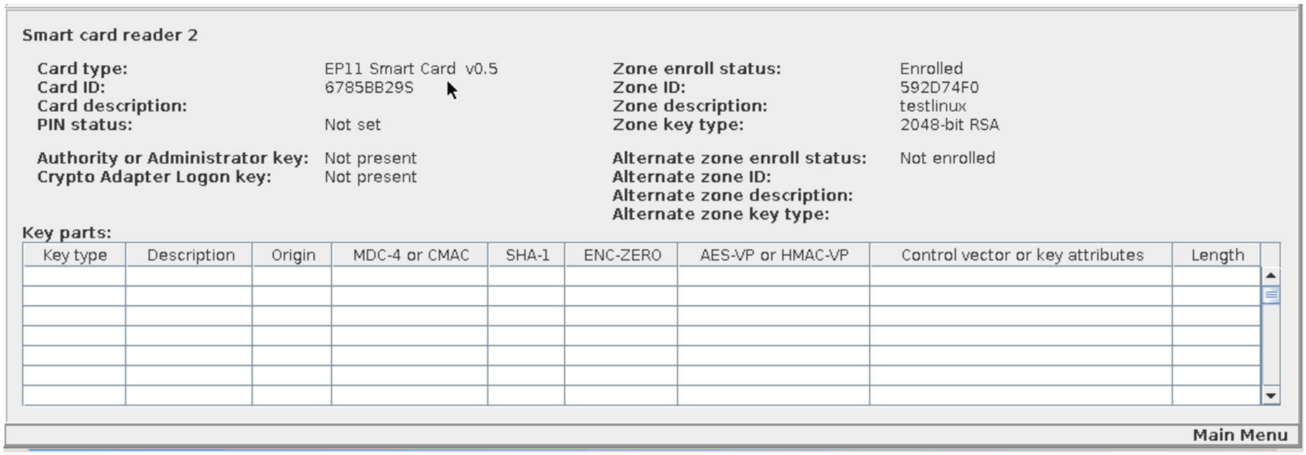

EP11 Smart Card - Initialize and enroll EP11 smart card

Insert smart card to be initialized as an EP11
smart card in smart card reader 2.

Figure 12. Insert smart card to be initialized as an EP11 smart card

Step 6

The **Smart Card Utility Program** initializes and builds the EP11 smart card and displays a message when the creation was successful. This may take some time. When the processing is finished, you see the new EP11 card information in an updated view from Figure 10 on page 21 (lower part) as shown in Figure 13 on page 22.


Smart card reader 2

Card type:	EP11 Smart Card v0.5	**Zone enroll status:**	Enrolled
Card ID:	6785BB29S	**Zone ID:**	592D74F0
Card description:		**Zone description:**	testlinux
PIN status:	Not set	**Zone key type:**	2048-bit RSA
Authority or Administrator key:	Not present	**Alternate zone enroll status:**	Not enrolled
Crypto Adapter Logon key:	Not present	**Alternate zone ID:**	
		Alternate zone description:	
		Alternate zone key type:	

Key parts:

Key type	Description	Origin	MDC-4 or CMAC	SHA-1	ENC-ZERO	AES-VP or HMAC-VP	Control vector or key attributes	Length

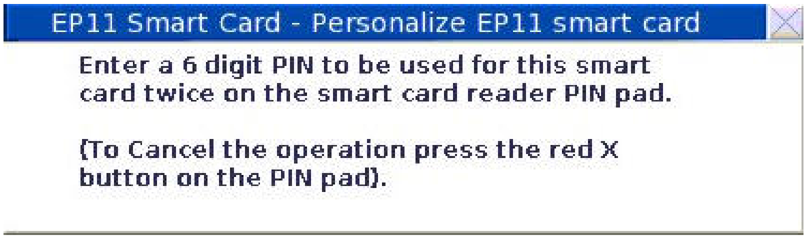
Figure 13. EP11 smart card successfully created

Step 7

The **Smart Card Utility Program** now goes back to the window shown in Figure 10 on page 21, where you now select item **Personalize EP11 smart card** from the **EP11 Smart Card** pull-down choice.

To personalize the EP11 smart card, the **Smart Card Utility Program** prompts you to enter a PIN to be used for this smart card on the smart card reader PIN pad. The PIN must be entered twice for confirmation.

The TKE also prompts you for an optional description for the smart card.



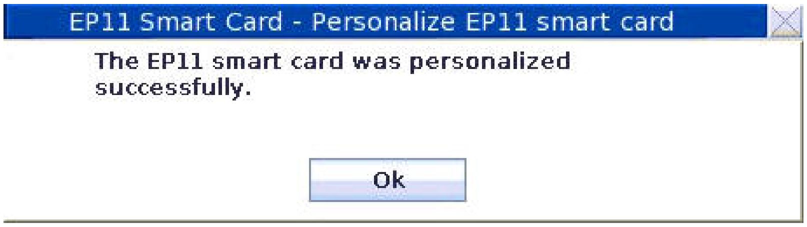
EP11 Smart Card - Personalize EP11 smart card

Enter a 6 digit PIN to be used for this smart card twice on the smart card reader PIN pad.

(To Cancel the operation press the red X button on the PIN pad).

Figure 14. Entering a PIN for the EP11 smart card

The **Smart Card Utility Program** informs you of a successful personalization of the EP11 smart card. This smart card now contains a certificate signed by the CA authority, and a PIN to access the smart card.



EP11 Smart Card - Personalize EP11 smart card

The EP11 smart card was personalized successfully.

Figure 15. EP11 smart card successfully personalized

The EP11 smart card is needed whenever you want to set a new master key on the adapter.

Step 8

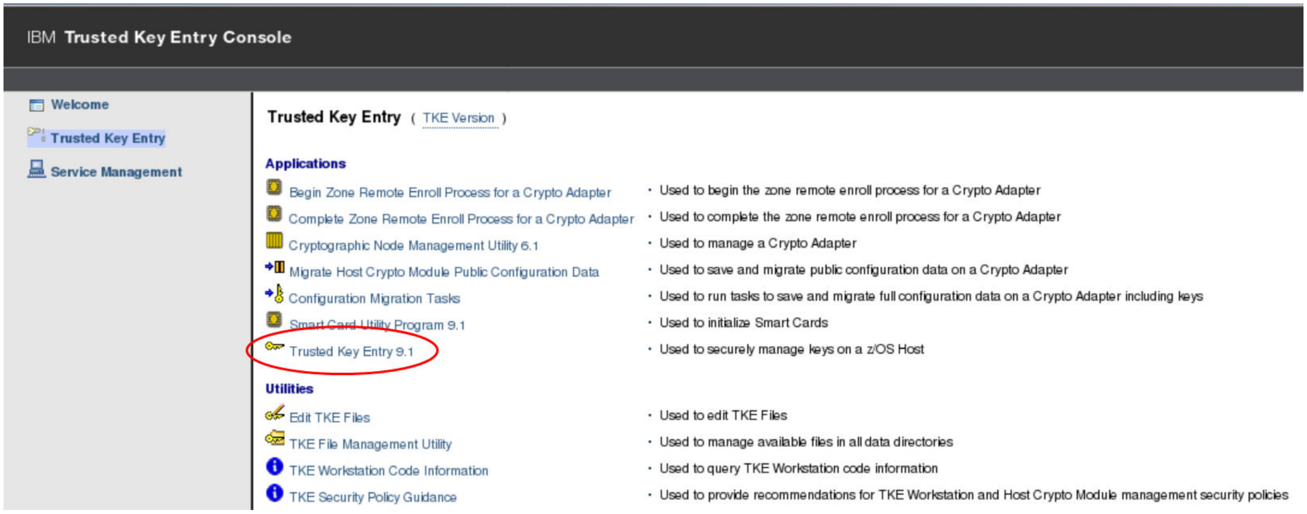
Repeat **Step 3** through **Step 7** to create the second EP11 smart card.

Creating a master key on the TKE workstation

Read an outline of the required steps for creating a master key and installing it on the CEX*P adapter. For detailed information about how to use the TKE workstation, refer to *z/OS Cryptographic Services ICSF Trusted Key Entry Workstation User's Guide*.

Step 1

Go to the **Trusted Key Entry** console as described in Step 1 of "Create and initialize an EP11 smart card" on page 20.



IBM Trusted Key Entry Console

Trusted Key Entry (TKE Version)

Applications

- Begin Zone Remote Enroll Process for a Crypto Adapter · Used to begin the zone remote enroll process for a Crypto Adapter
- Complete Zone Remote Enroll Process for a Crypto Adapter · Used to complete the zone remote enroll process for a Crypto Adapter
- Cryptographic Node Management Utility 6.1 · Used to manage a Crypto Adapter
- Migrate Host Crypto Module Public Configuration Data · Used to save and migrate public configuration data on a Crypto Adapter
- Configuration Migration Tasks · Used to run tasks to save and migrate full configuration data on a Crypto Adapter including keys
- Smart Card Utility Program 9.1 · Used to initialize Smart Cards
- Trusted Key Entry 9.1 · Used to securely manage keys on a z/OS Host

Utilities

- Edit TKE Files · Used to edit TKE Files
- TKE File Management Utility · Used to manage available files in all data directories
- TKE Workstation Code Information · Used to query TKE Workstation code information
- TKE Security Policy Guidance · Used to provide recommendations for TKE Workstation and Host Crypto Module management security policies

Figure 16. TKE Console - initial window

Step 2

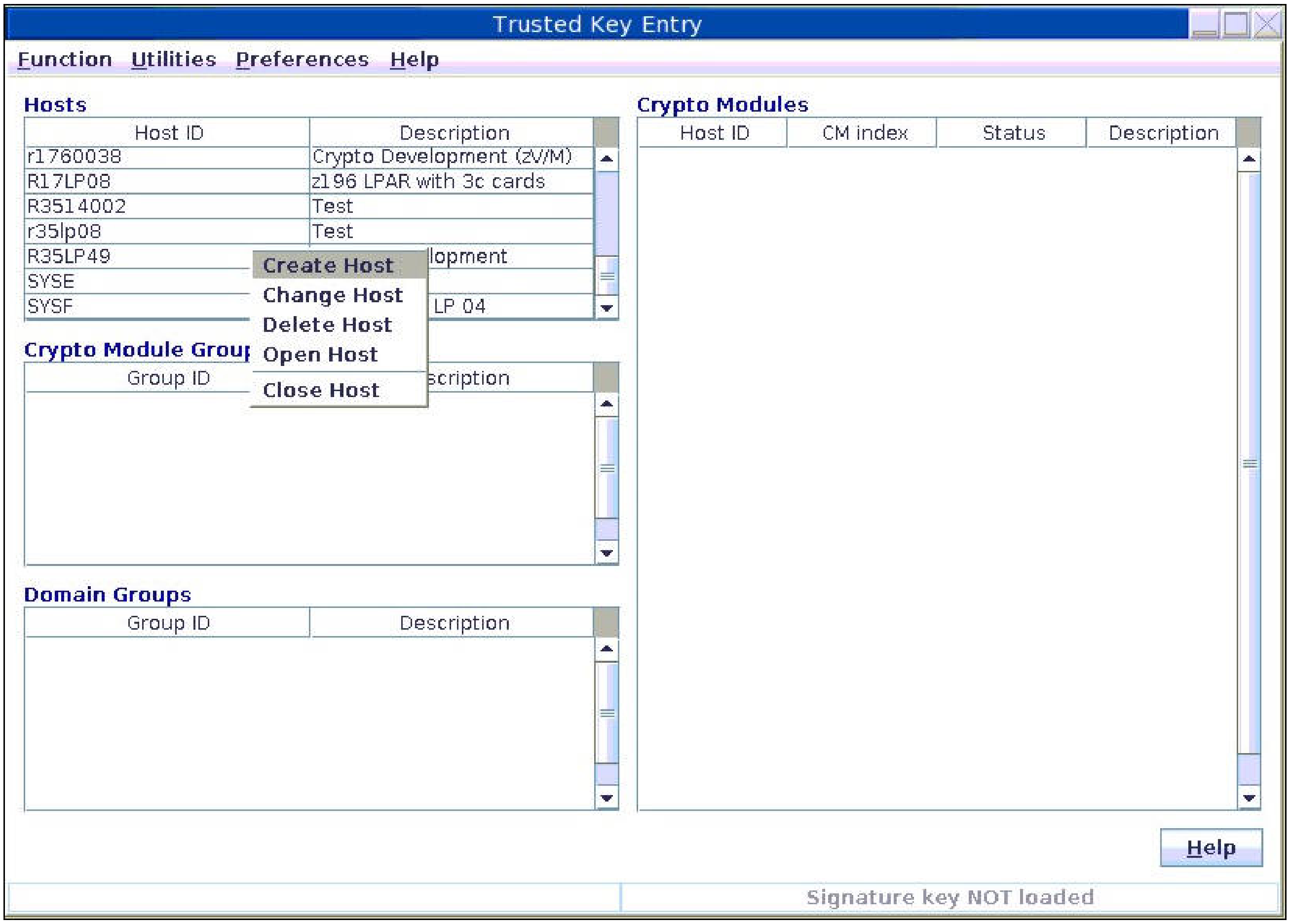
Click the **Trusted Key Entry** application as shown in Figure 16 on page 23. Then proceed with the logon procedure as described in Step 2 of "Create and initialize an EP11 smart card" on page 20.

Step 3

Select the default profile ID TKEUSER, click **OK**, and in the upcoming **Passphrae Logon** dialog for this profile, logon with the passphrase associated to TKEUSER.

Step 4

The **Trusted Key Entry** main window is displayed (Figure 17 on page 24). Open the context menu for hosts and select action **Create Host**.

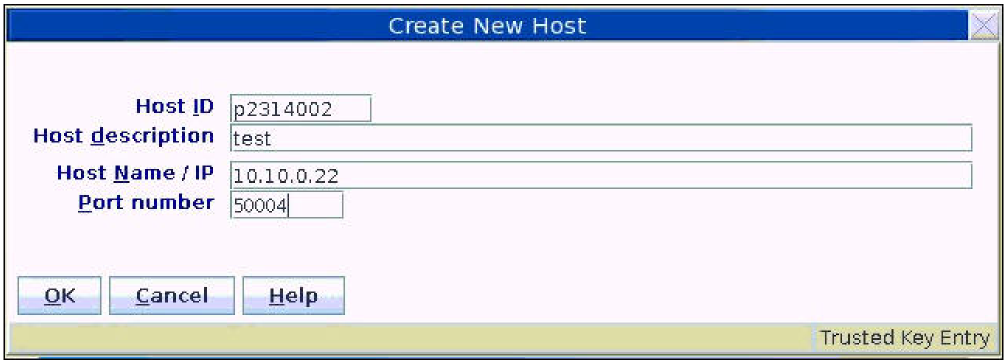


Host ID	Description
r1760038	Crypto Development (zV/M)
R17LP08	z196 LPAR with 3c cards
R3514002	Test
r35lp08	Test
R35LP49	lopment
SYSE	
SYSF	LP 04

Figure 17. Trusted Key Entry - main window

Step 5

In the **Create New Host** dialog, enter the required values of the host for which you want to create the master key. It is assumed that this host is a Linux on Z system running the ep11TKEd TKE daemon. Press **OK** to return to the **Trusted Key Entry** main window.



Host ID: p2314002

Host description: test

Host Name / IP: 10.10.0.22

Port number: 50004

Figure 18. TKE - Create new Host

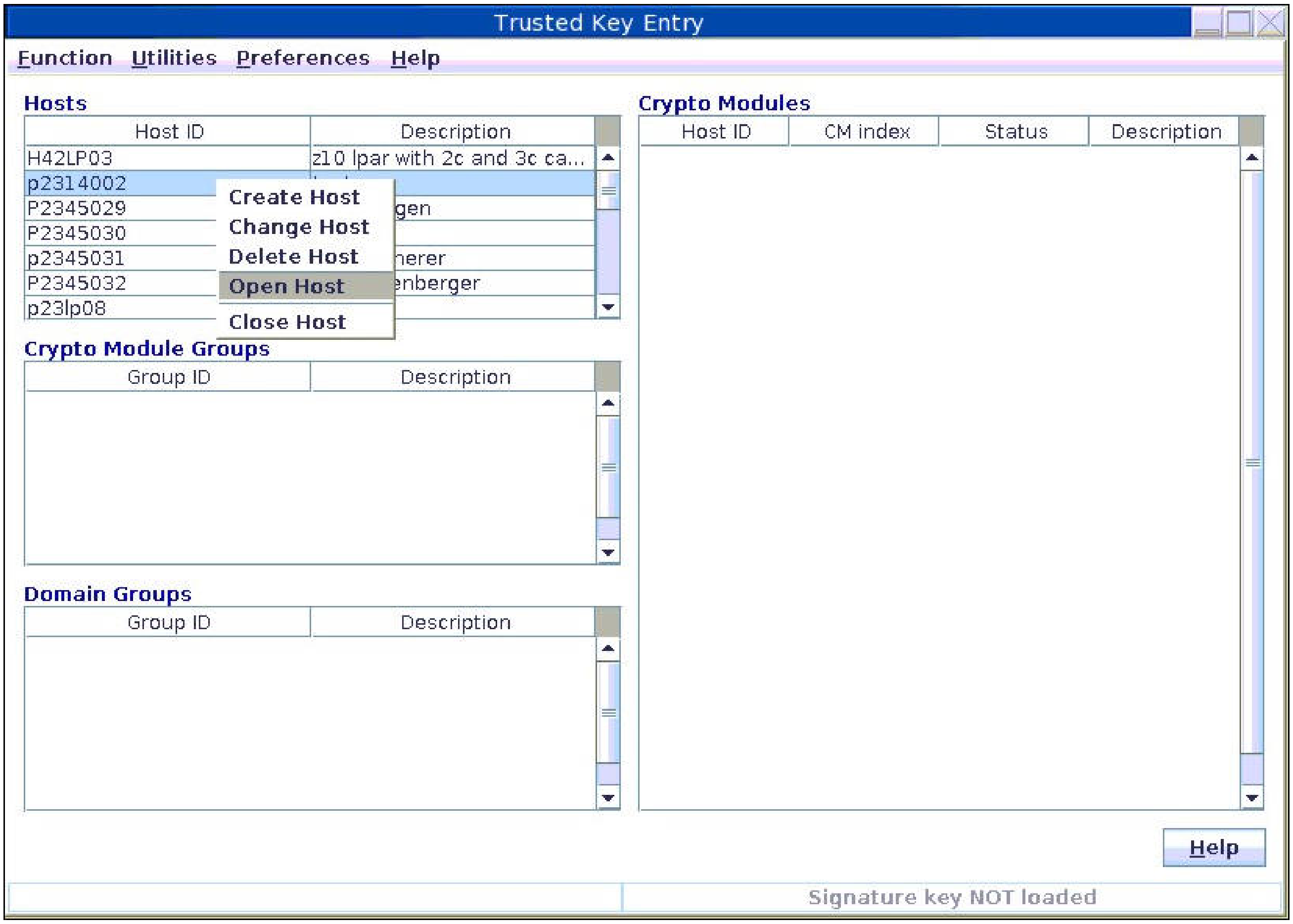
Step 6

The new host is visible now within the list of host IDs.

Before you continue to work on the new host, ensure the following:

- The ep11TKEd TKE daemon is started on the host.
- The TKE has connectivity to the host.

Then open the new host's context menu and select action **Open Host**.



Hosts

Host ID	Description
H42LP03	z10 lpar with 2c and 3c ca...
p2314002	
P2345029	gen
P2345030	
p2345031	herer
P2345032	enberger
p23lp08	

Crypto Module Groups

Group ID	Description

Domain Groups

Group ID	Description

Crypto Modules

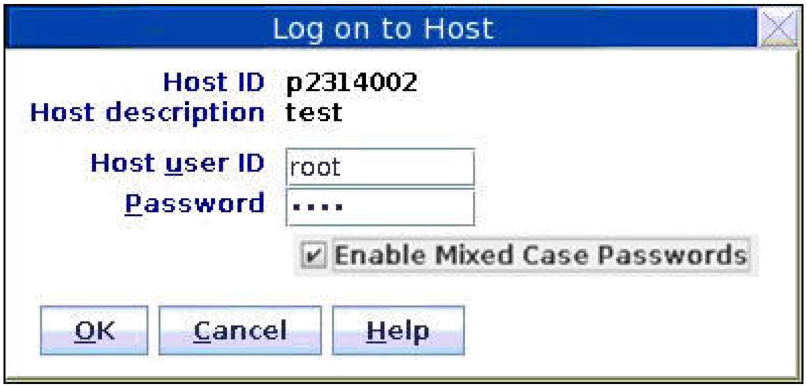
Host ID	CM index	Status	Description

Signature key NOT loaded

Figure 19. Trusted Key Entry - main window with new created host

Step 7

If you accepted the default settings when installing and configuring the ep11TKEd TKE daemon as recommended, you are prompted by the TKE workstation to log on to the selected host with the appropriate Linux user credentials. Do not forget to activate the **Enable Mixed Case Passwords** check box (see Figure 20 on page 26). This user must be a member in the *ep11tke group* defined in `/etc/group` of the system.



Log on to Host

Host ID p2314002

Host description test

Host user ID root

Password ••••

☑ Enable Mixed Case Passwords

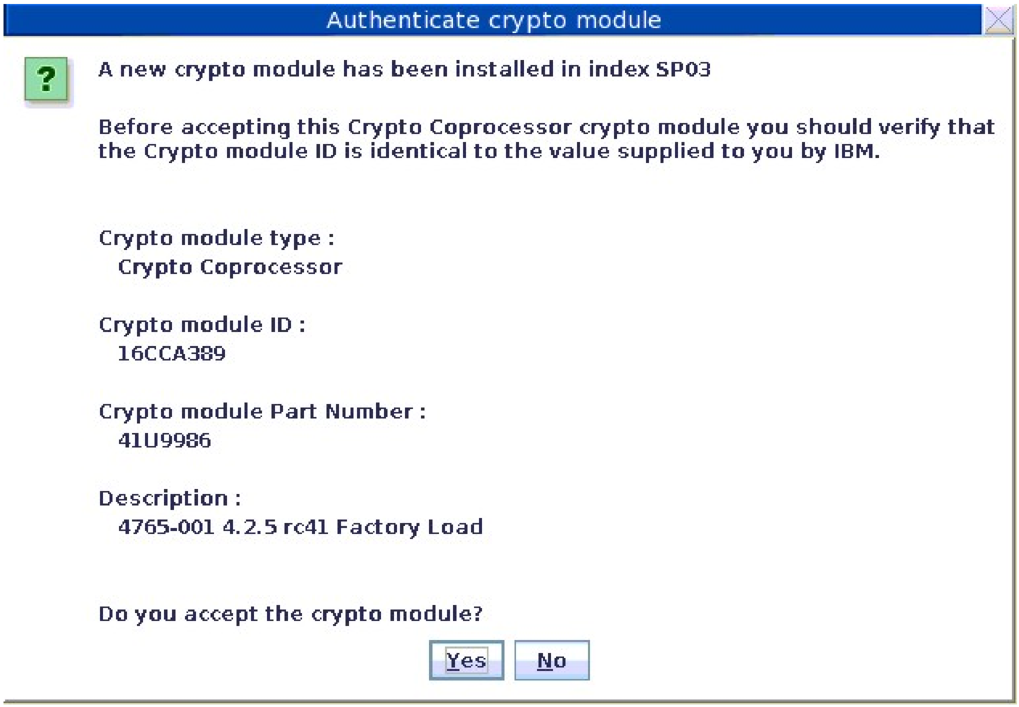
Figure 20. Log on to new host

If you specified `CipherMode=None` in the TKE daemon configuration file `/etc/ep11/ep11tked.conf`, the values that you enter as the user ID and password are not relevant, because they are not validated. You just need to press the **OK** button.

For details about configuring the ep11TKEd TKE daemon, read "Installing the host part of the EP11 library" on page 17 or refer to file `README_TKED.md` coming with the EP11 installation package.

Step 8

The TKE now requests a verification of any new crypto adapter. Press the **Yes** button to continue.



Authenticate crypto module

A new crypto module has been installed in index SP03

Before accepting this Crypto Coprocessor crypto module you should verify that the Crypto module ID is identical to the value supplied to you by IBM.

Crypto module type :
Crypto Coprocessor

Crypto module ID :
16CCA389

Crypto module Part Number :
41U9986

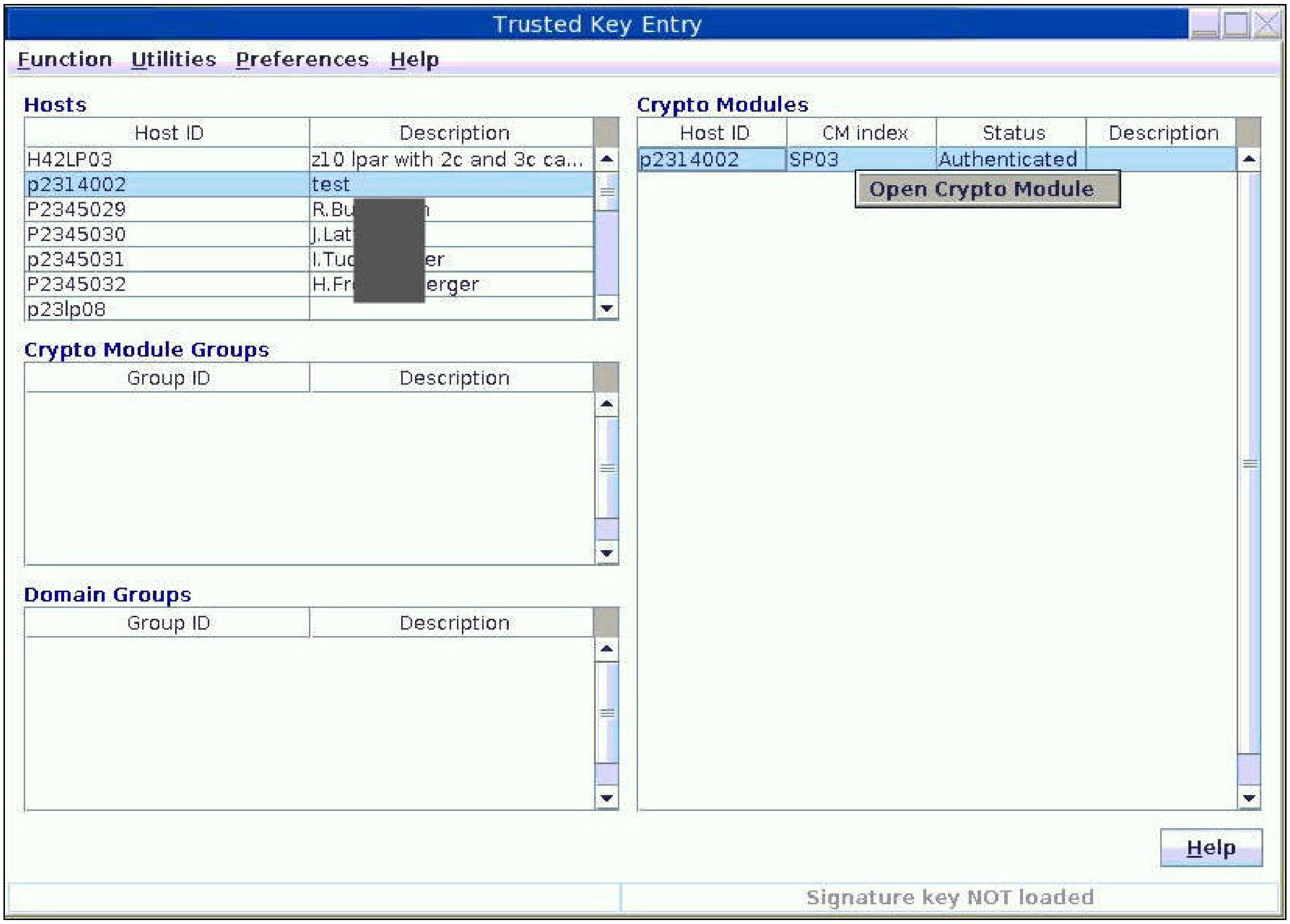
Description :
4765-001 4.2.5 rc41 Factory Load

Do you accept the crypto module?

Figure 21. Authenticate crypto module

Step 9

The **Crypto Modules** list now displays the available adapters. In the sample from Figure 22 on page 27, there is just one adapter available with host ID *p2314002*. Select a crypto adapter of your choice and trigger action **Open Crypto Module** from its context menu.



Hosts

Host ID	Description
H42LP03	z10 lpar with 2c and 3c ca...
p2314002	test
P2345029	R.Bu[redacted]n
P2345030	J.Lat[redacted]
p2345031	I.Tuc[redacted]er
P2345032	H.Fr[redacted]erger
p23lp08	

Crypto Module Groups

Group ID	Description

Domain Groups

Group ID	Description

Crypto Modules

Host ID	CM index	Status	Description
p2314002	SP03	Authenticated	

Figure 22. Crypto Modules list

Step 10

The **Crypto Module Administration** window for the selected crypto adapter opens. Now you can start to configure the domains. Click on the **Domains** tab at the top. On the right side, the window now shows an **Index** tab for each available domain. Choose one of these indexes and select the **Domain Administrators** tab at the bottom of the window to add a new administrator role. In this documentation, the configuration is outlined for the domain with index *13*. For detailed information on domain configuration, refer to the TKE documentation.

Step 11

Now create a user ID with administrator role in the **Crypto Module Administration** window for the selected crypto adapter. Open the context menu by right-clicking into the white space of the window. Select action **Add Administrator**.

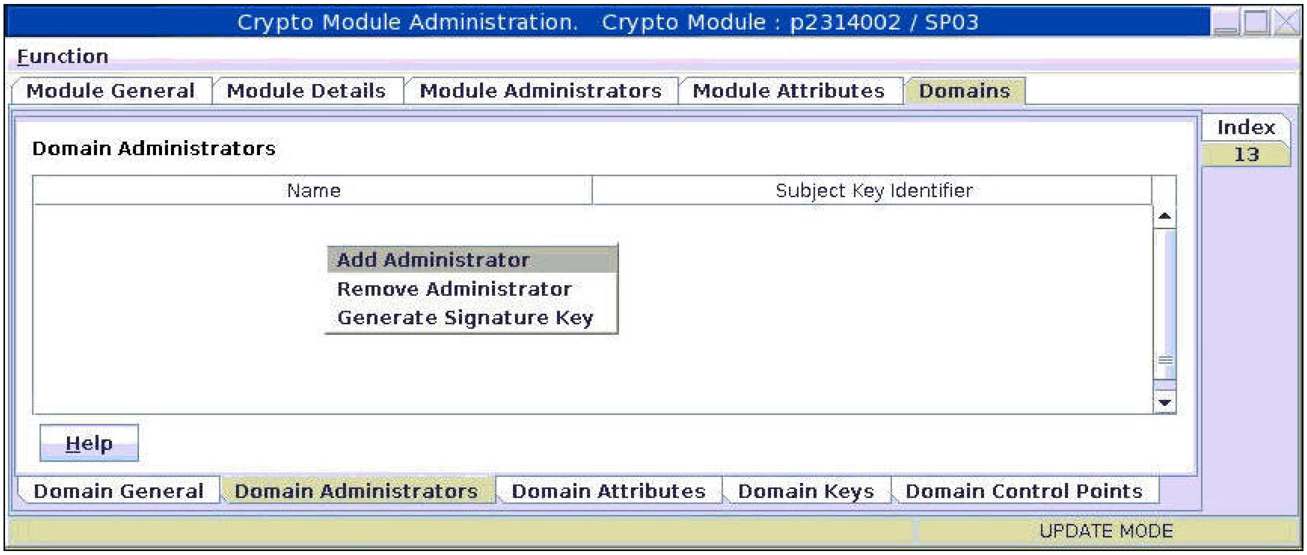
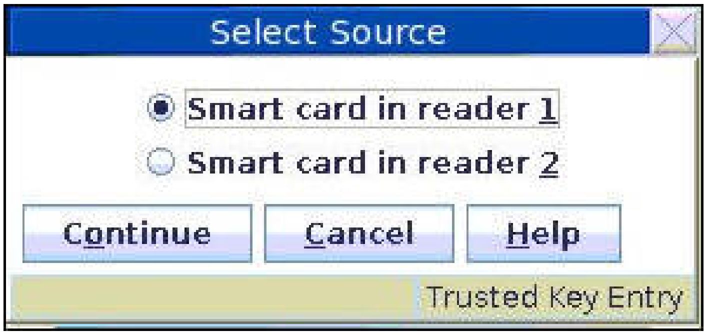


Figure 23. Crypto Module Administration - with context menu

From the opening **Select Source** window, TKE requests certain information from the previously CA prepared smart card that contains the administrator key and certificate.

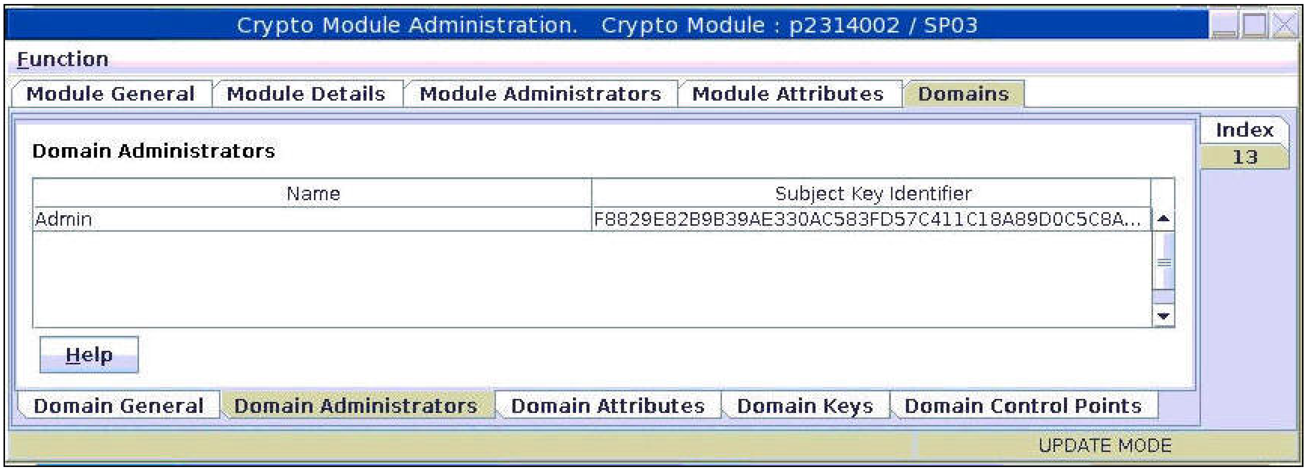


● Smart card in reader 1

○ Smart card in reader 2

Figure 24. Select Source

After a successful authentication on the smart card reader, the TKE workstation imports the administrator key and certificate and creates an administrator profile.



Domain Administrators

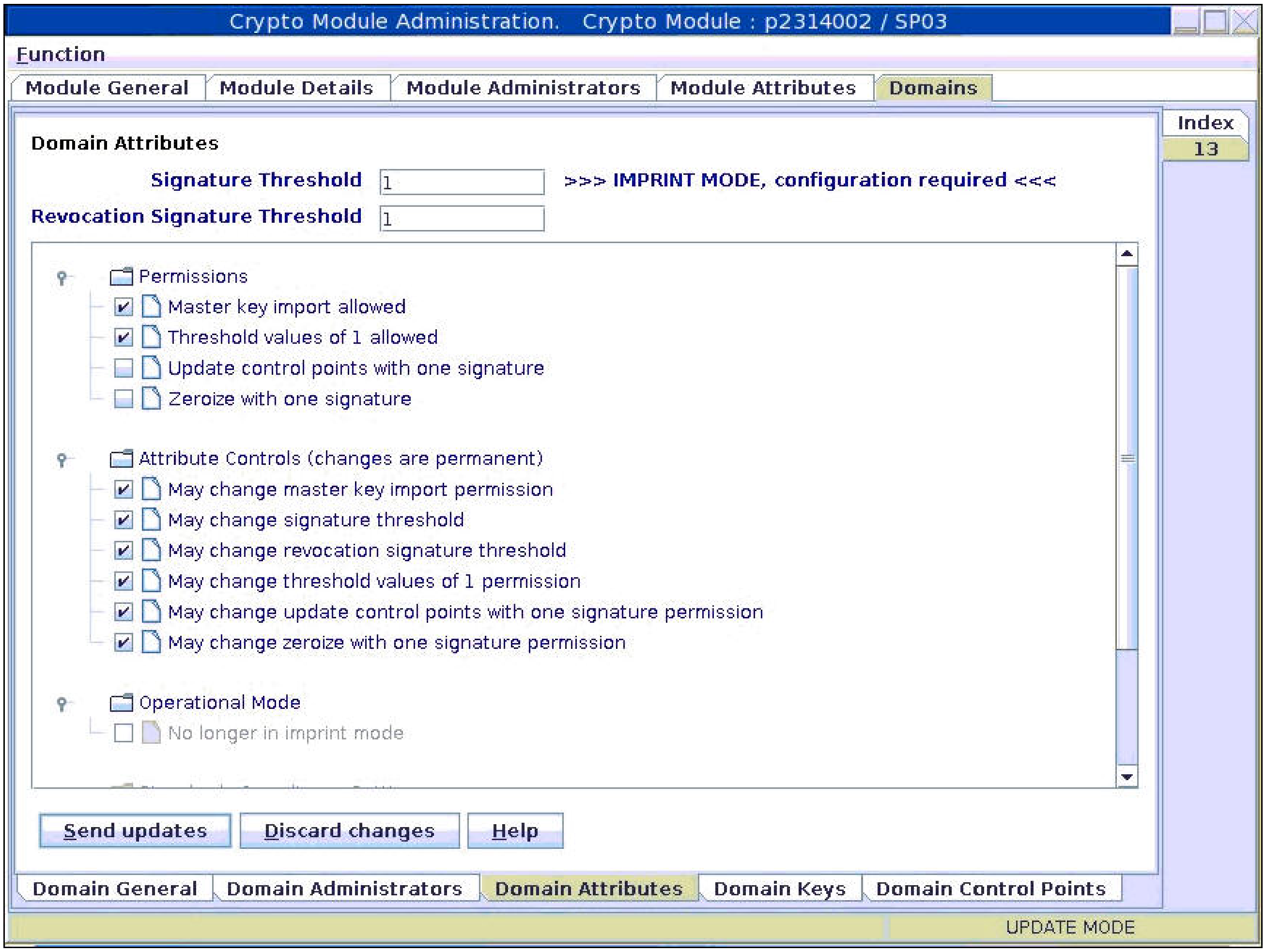
Name	Subject Key Identifier
Admin	F8829E82B9B39AE330AC583FD57C411C18A89D0C5C8A...

Figure 25. Crypto Module Administration - Subject Key Identifier

Step 12

Now select the **Domain Attributes** tab at the bottom of the window. This selection opens the window that is shown in Figure 26 on page 29 where you can specify the required permissions and attribute controls for the current domain.

Per default, the *Signature Threshold* and the *Revocation Signature Threshold* are set to 0. Both values must be changed at least to 1 to release the card from the IMPRINT mode. For more information, see the TKE documentation. Press **Send updates** to apply your settings.



Domain Attributes

Signature Threshold: 1 >>> IMPRINT MODE, configuration required <<<

Revocation Signature Threshold: 1

Permissions
- ☑ Master key import allowed
- ☑ Threshold values of 1 allowed
- □ Update control points with one signature
- □ Zeroize with one signature

Attribute Controls (changes are permanent)
- ☑ May change master key import permission
- ☑ May change signature threshold
- ☑ May change revocation signature threshold
- ☑ May change threshold values of 1 permission
- ☑ May change update control points with one signature permission
- ☑ May change zeroize with one signature permission

Operational Mode
- □ No longer in imprint mode

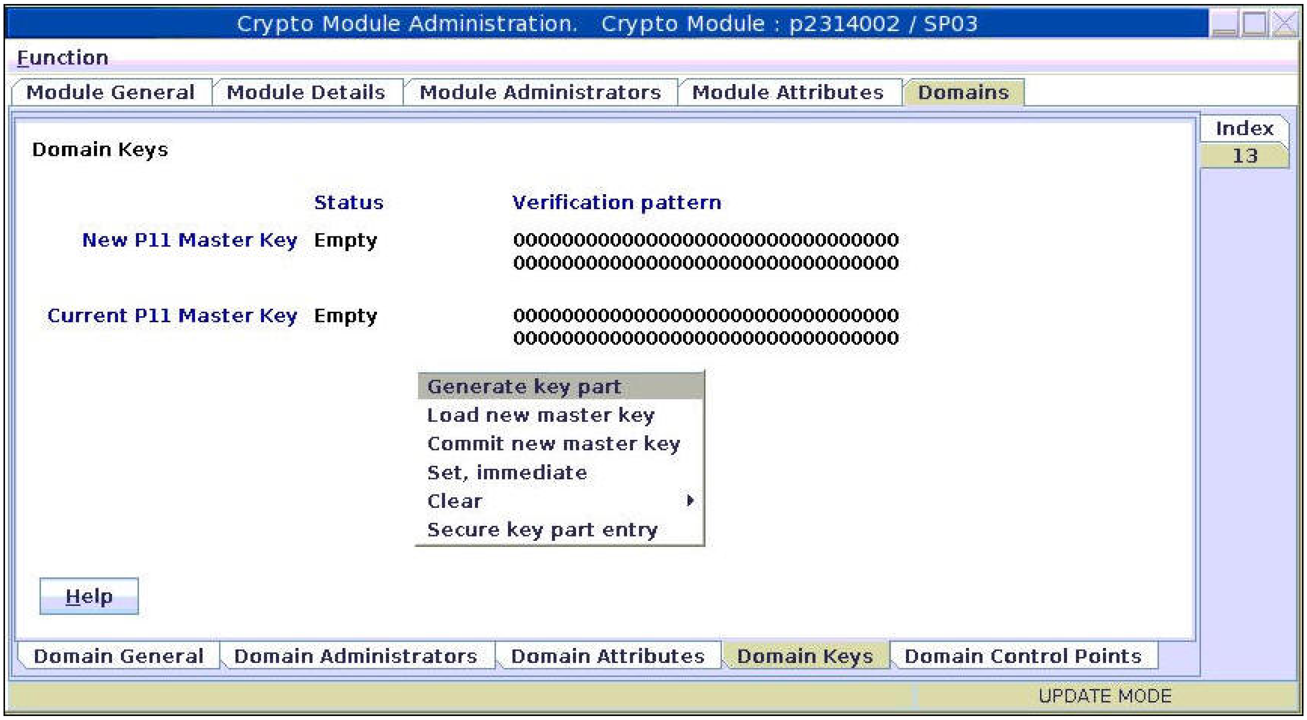
Figure 26. Crypto Module Administration - Setting permissions and attribute controls

Step 13

Now select the **Domain Keys** tab from the bottom of the **Crypto Module Administration** window.

The new **Crypto Module Administration** window with verification patterns for the new and current master key is displayed. The patterns are all set to 0, because the current and new master keys are empty yet.

Open the context menu by right-clicking in the white space, and select action **Generate key part**.



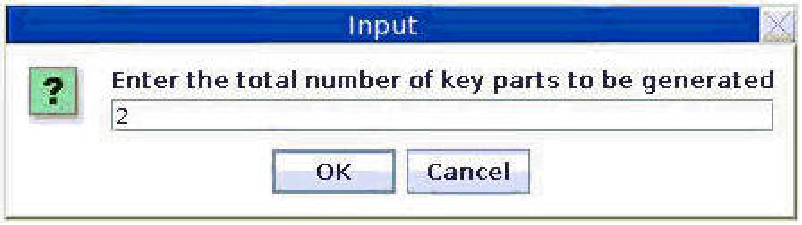
Domain Keys

	Status	Verification pattern
New P11 Master Key	Empty	00000000000000000000000000000000 00000000000000000000000000000000
Current P11 Master Key	Empty	00000000000000000000000000000000 00000000000000000000000000000000

Figure 27. Crypto Module Administration - Generate key part

Step 14

The TKE workstation now prompts you to enter the total number of key parts to be generated. You must at least generate two parts. Enter your input and press the **OK** button.



Enter the total number of key parts to be generated
2

Figure 28. Crypto Module Administration - Input for total number of key parts to be generated

Step 15

In a similar way as in the previous step, you are now prompted to insert an EP11 smart card and to enter a name and description for each generated key part. The TKE workstation informs you about a successful storage of all generated key parts and descriptions. The new master key can now be generated by the TKE component.

Step 16

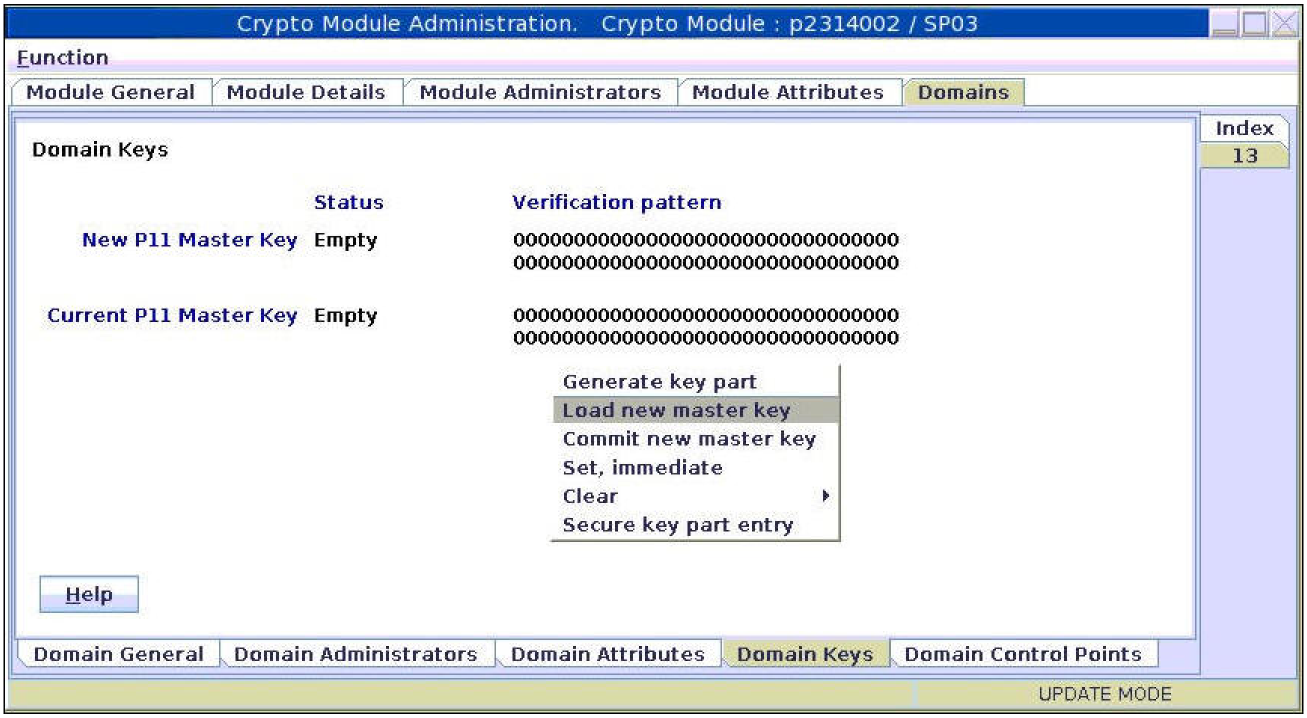
During the current process, the new master key now needs to go through three stages:

Load
The key is just stored on the adapter, but not active.

Commit
The key is activated and is now present on the adapter as the new master key. In this state, the existing objects encrypted under the current master key can be re-encrypted by using this new master key.

Set
The new master key is now switched to become the current key to be used.

Start with the **Load** step: Load the new generated master key parts from the cards to the target crypto adapter. For this purpose, open the context menu from the **Crypto Module Administration** window and select action **Load new master key**. TKE now prompts you for the total number of key parts to be loaded. Type the number of previously generated key parts. TKE then prompts you to load each key part separately.


Crypto Module Administration. Crypto Module : p2314002 / SP03

Domain Keys

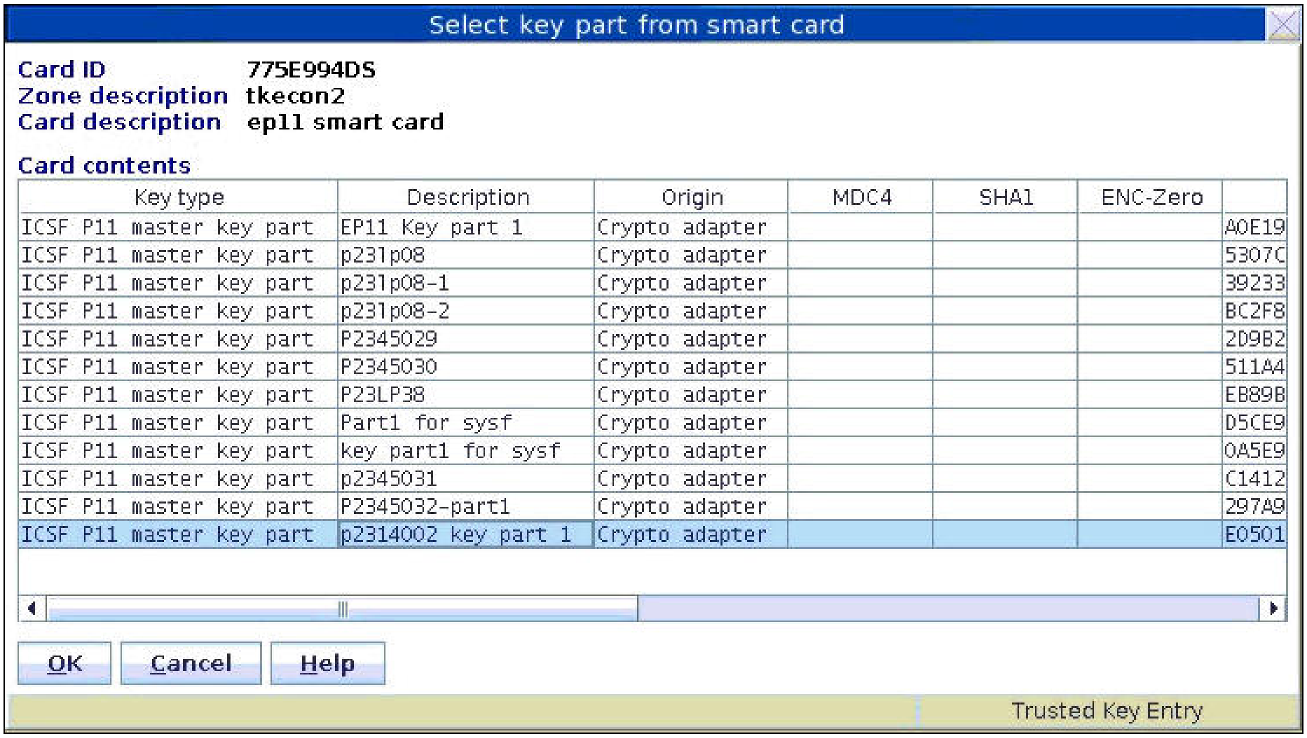
	Status	Verification pattern
New P11 Master Key	Empty	0000000000000000000000000000000000 0000000000000000000000000000000000
Current P11 Master Key	Empty	0000000000000000000000000000000000 0000000000000000000000000000000000

Index 13

UPDATE MODE

Figure 29. Crypto Module Administration - Load new master key

The TKE workstation opens the window **Select key part from smart card** as shown in Figure 30 on page 32. From this window, you can commit the single parts of your key. From the list of shown key parts, select that part that you now want to commit and press **OK**.



Card ID 775E994DS
Zone description tkecon2
Card description ep11 smart card

Card contents

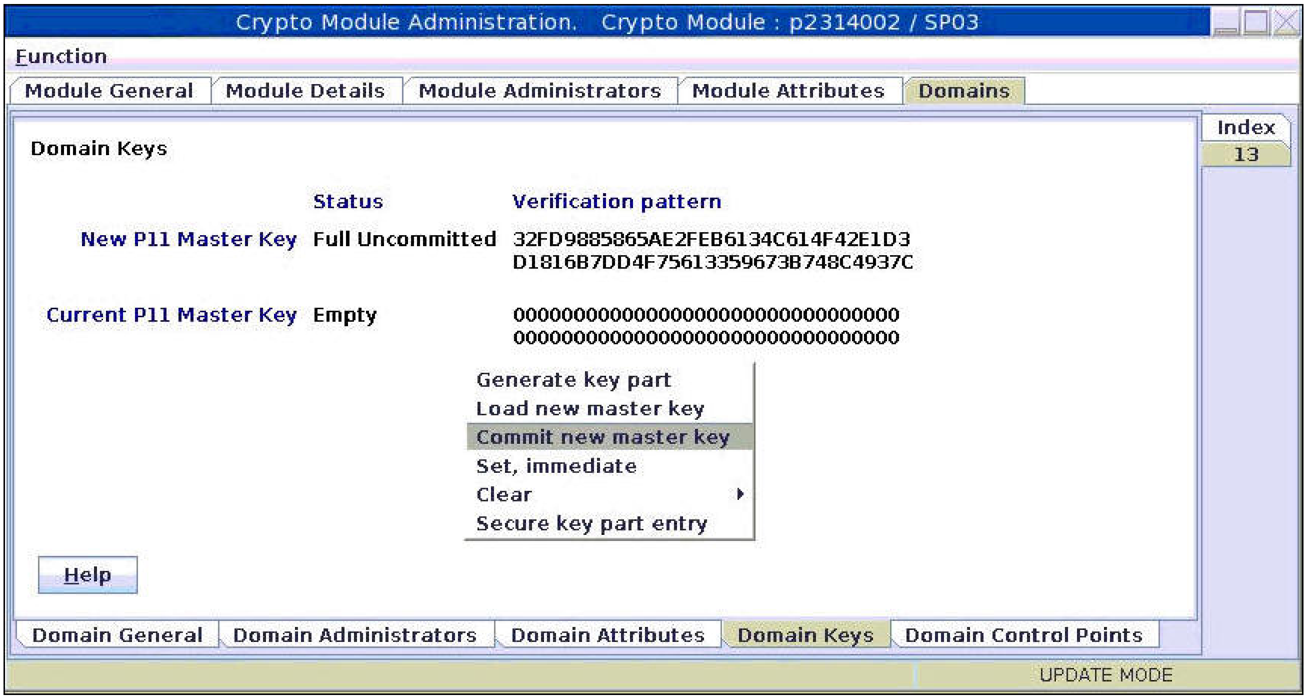
Key type	Description	Origin	MDC4	SHA1	ENC-Zero	
ICSF P11 master key part	EP11 Key part 1	Crypto adapter				A0E19
ICSF P11 master key part	p231p08	Crypto adapter				5307C
ICSF P11 master key part	p231p08-1	Crypto adapter				39233
ICSF P11 master key part	p231p08-2	Crypto adapter				BC2F8
ICSF P11 master key part	P2345029	Crypto adapter				2D9B2
ICSF P11 master key part	P2345030	Crypto adapter				511A4
ICSF P11 master key part	P23LP38	Crypto adapter				EB89B
ICSF P11 master key part	Part1 for sysf	Crypto adapter				D5CE9
ICSF P11 master key part	key part1 for sysf	Crypto adapter				0A5E9
ICSF P11 master key part	p2345031	Crypto adapter				C1412
ICSF P11 master key part	P2345032-part1	Crypto adapter				297A9
ICSF P11 master key part	p2314002 key part 1	Crypto adapter				E0501

Figure 30. Select key part from smart card

Step 17

After you loaded all single master key parts, the complete master key is successfully loaded onto the CEX*P adapter.

The TKE workstation switches back to the **Crypto Module Administration** window. You can see that the new master key is full/complete, but yet uncommitted. To commit the new master key, invoke the context menu and select action **Commit new master key**. The status switches to *Full Committed*, as shown in Figure 32 on page 33.



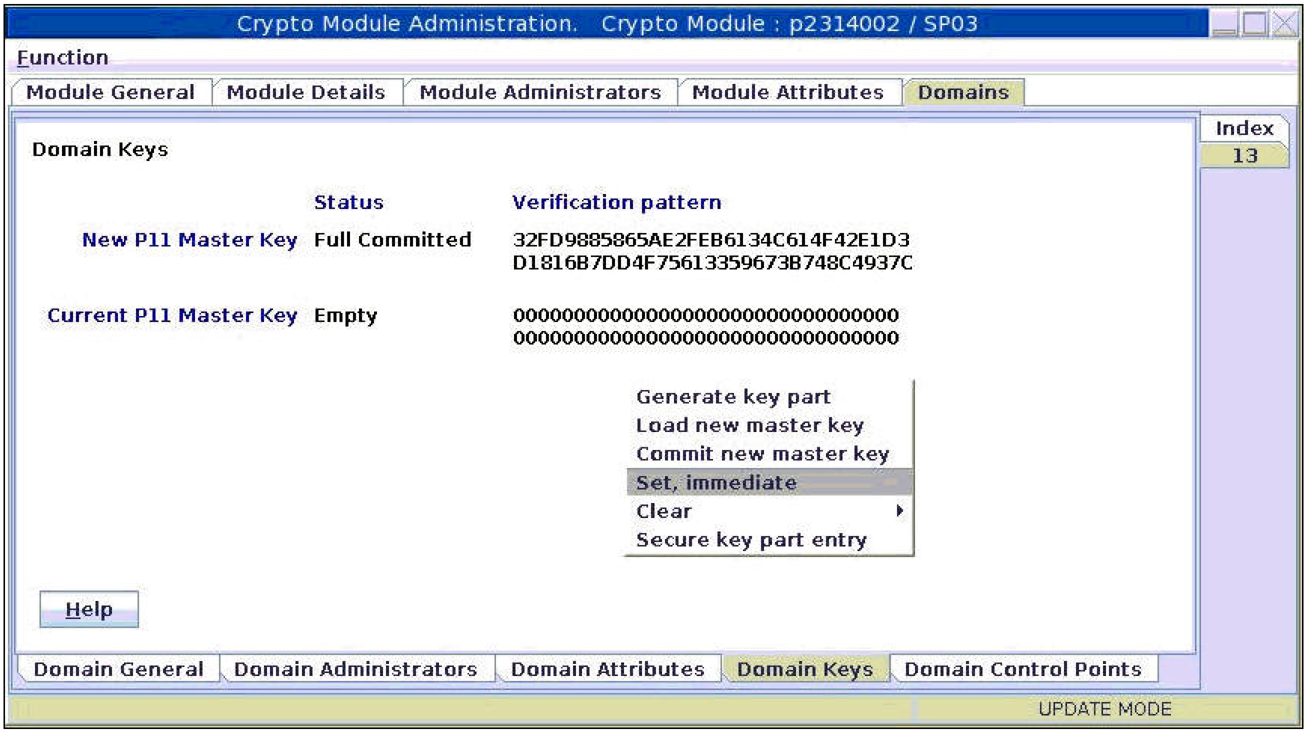
Domain Keys

	Status	Verification pattern
New P11 Master Key	Full Uncommitted	32FD9885865AE2FEB6134C614F42E1D3 D1816B7DD4F75613359673B748C4937C
Current P11 Master Key	Empty	00000000000000000000000000000000 00000000000000000000000000000000

Figure 31. Crypto Module Administration - Commit new master key

Step 18

You can now immediately set the new master key. From the context menu, start action **Set, immediate**.



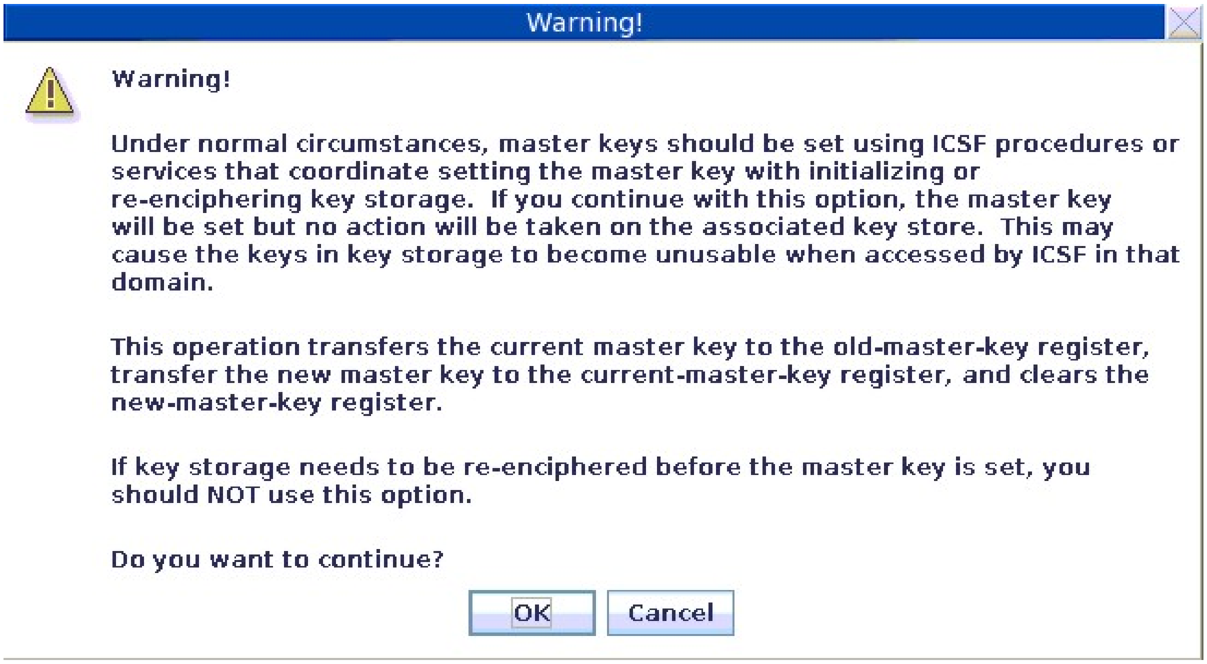
Domain Keys

	Status	Verification pattern
New P11 Master Key	Full Committed	32FD9885865AE2FEB6134C614F42E1D3 D1816B7DD4F75613359673B748C4937C
Current P11 Master Key	Empty	00000000000000000000000000000000 00000000000000000000000000000000

Figure 32. Crypto Module Administration - Set, immediate

Before you actually perform the action, the TKE comes up with a warning. If this is the first time you generated a master key, or if there are no keys stored on your host that are wrapped by the current master key, you can ignore the warning by pressing the **OK** button.

If there are keys wrapped by the current master key on your host, then you should not generate a new master key, but follow the procedure described in "Migrating master keys with the pkcsep11_migrate tool" on page 63.



Warning!

Under normal circumstances, master keys should be set using ICSF procedures or services that coordinate setting the master key with initializing or re-enciphering key storage. If you continue with this option, the master key will be set but no action will be taken on the associated key store. This may cause the keys in key storage to become unusable when accessed by ICSF in that domain.

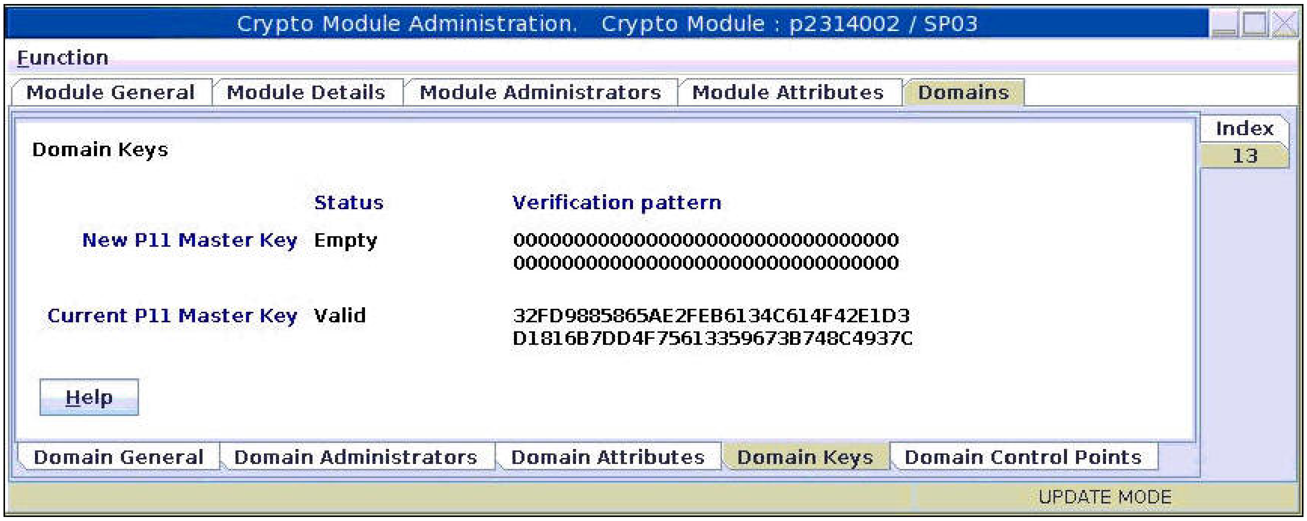
This operation transfers the current master key to the old-master-key register, transfer the new master key to the current-master-key register, and clears the new-master-key register.

If key storage needs to be re-enciphered before the master key is set, you should NOT use this option.

Do you want to continue?

Figure 33. Warning before setting the master key

See the result in Figure 34 on page 34: The new master key is now switched to the *Current Master Key*, and its status is *Valid*.



Domain Keys

	Status	Verification pattern
New P11 Master Key	Empty	00000000000000000000000000000000 00000000000000000000000000000000
Current P11 Master Key	Valid	32FD9885865AE2FEB6134C614F42E1D3 D1816B7DD4F75613359673B748C4937C

Figure 34. Crypto Module Administration - valid current master key

If you need to change the master key, see "Migrating master keys with the pkcsep11_migrate tool" on page 63.

Installing openCryptoki

The EP11 token is part of the openCryptoki package starting with version 3.1. openCryptoki in turn is shipped with the Linux on Z distributions.

Check whether you already installed openCryptoki in your current environment, for example:

```
$ rpm -qa | grep -i opencryptoki /* for RPM */
$ dpkg -l | grep -i opencryptoki /* for DEB */
```

Note: The command examples are distribution dependent. `opencryptoki` must in certain distributions be specified as `openCryptoki` (case-sensitive).

You should see all installed openCryptoki packages. If required packages are missing, use the installation tool of your Linux distribution to install the appropriate openCryptoki RPM or DEB.

Note: You must remove any previous package of openCryptoki, before you can install the new package version 3.10.

Installing from the RPM or DEB package

The openCryptoki version 3.10 or higher packages, that comprise the EP11 support (EP11 token) are delivered by the distributors. Distributors build these packages as RPM or DEB packages for delivering them to customers.

Customers can install these openCryptoki packages by using the installation tool of their selected distribution.

If you received openCryptoki as an *RPM* package, follow the *RPM* installation process that is described in the *RPM* man page. If you received an openCryptoki *DEB* package, you can use the dpkg - package manager for Debian described in the *dpkg* man page.

The installation from either an *RPM* or *DEB* package is the preferred installation method.

Installing from the source package

As an alternative, for example for development purposes, you can get the latest openCryptoki version (inclusive latest patches) from the GitHub repository and build it yourself. But this version is not serviced. It is suitable for non-production systems and early feature testing, but you should not use it for production.

In this case, refer to the `INSTALL` file in the top level of the source tree. You can start from the instructions that are provided with the subtopics of this `INSTALL` file and select from the described alternatives. If you use this installation method parallel to the installation of a package from your distributor, then you should keep both installations isolated from each other.

1. Download the latest version of the openCryptoki sources from:

   ```
   https://github.com/opencryptoki/opencryptoki/releases/tag/v3.10.0
   ```

2. Decompress and extract the compressed tape archive (tar.gz - file). There is a new directory named like `opencryptoki-3.10.x`.
3. Change to that directory and issue the following scripts and commands:

   ```
   $ ./bootstrap
   $ ./configure
   $ make
   $ make install
   ```

   The scripts or commands perform the following functions:

   bootstrap
   Initial setup, basic configurations

   configure
   Check configurations and build the makefile

   make
   Compile and link

   make install
   Install the libraries

Note: When installing openCryptoki from the source package, the location of some installed files will differ from the location of files installed from an RPM or DEB package.

Chapter 4. Configuring openCryptoki for EP11 support

After a successful installation of openCryptoki, you need to perform certain configuration and customization tasks to enable the exploitation of the EP11 library functions from applications. Especially, you need to set up tokens and daemons and then initialize the tokens.

openCryptoki, and in particular the slot manager, can handle several tokens, which can have different support for different hardware devices or software solutions. As shown in Figure 1 on page 4, the EP11 token interacts with the host part of the EP11 library. EP11 can operate with the Crypto Express adapter with EP11 firmware load for symmetric and asymmetric cryptographic functions.

For a complete configuration of the Linux on Z EP11 enablement, finish the tasks as described in the contained subtopics.

- "Adjusting the openCryptoki configuration file" on page 37
- "Defining an EP11 token-specific configuration file" on page 41
- "Setting environment variables" on page 44
- "Initializing EP11 tokens" on page 45

Finally, to control your configuration results, follow the instructions provided in "How to recognize an EP11 token" on page 45.

Adjusting the openCryptoki configuration file

A preconfigured list of all available tokens that are ready to register to the openCryptoki slot daemon is required before the slot daemon can start. This list is provided by the global configuration file called `opencryptoki.conf`. Read this topic for information on how to adapt this file according to your installation.

Table 1 on page 37 lists the maximum number of available libraries that may be in place after you successfully installed openCryptoki. It may vary for different distributions and is dependent from the installed packages.

Also, Linux on Z does not support the Trusted Platform Module (TPM) token library.

A token is only available, if the token library is installed, and the appropriate software and hardware support pertaining to the stack of the token is also installed. For example, the EP11 token is only available if all parts of the EP11 library software are installed and a Crypto Express EP11 coprocessor is detected.

A token needs not be available, even if the corresponding token library is installed. Display the list of available tokens by using the command:

```
$ pkcsconf -t
```

Table 1. openCryptoki libraries

Library	Explanation
`/usr/lib64/opencryptoki/libopencryptoki.so`	openCryptoki base library
`/usr/lib64/opencryptoki/stdll/libpkcs11_ica.so`	libica token library
`/usr/lib64/opencryptoki/stdll/libpkcs11_sw.so`	software token library
`/usr/lib64/opencryptoki/stdll/libpkcs11_tpm.so`	TPM token library

Table 1. openCryptoki libraries (continued)

Library	Explanation
`/usr/lib64/opencryptoki/stdll/libpkcs11_cca.so`	CCA token library
`/usr/lib64/opencryptoki/stdll/libpkcs11_ep11.so`	EP11 token library
`/usr/lib64/opencryptoki/stdll/libpkcs11_icsf.so`	ICSF token library

The `/etc/opencryptoki/opencryptoki.conf` file must exist and it must contain an entry for each instance of an EP11 token to make these instances available. By default, one such entry is available upon installation (see the `slot  4` entry in the provided sample configuration from Figure 35 on page 38).

```
version opencryptoki-3.15

# 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
}
```

Figure 35. Default opencryptoki.conf

Note:

- The standard path for slot token dynamic link libraries (STDLLs) is: `/usr/lib64/opencryptoki/stdll/`.

- The standard path for the token-specific EP11 token configuration file (in our example, ep11tok.conf) is /etc/opencryptoki/. You can change this path by using the OCK_EP11_TOKEN_DIR environment variable. For more information, read "Defining an EP11 token-specific configuration file" on page 41.
- You can use the default opencryptoki.conf file only for a single EP11 token. If you want to use multiple EP11 tokens, read the information in "Adding EP11 tokens to openCryptoki" on page 39.

Use one of the following commands to start the slot daemon, which reads out the configuration information and sets up the tokens:

```
$ service pkcsslotd start
$ systemctl start pkcsslotd.service   /* for Linux distributions providing systemd */
```

For a permanent solution, specify:

```
$ chkconfig pkcsslotd on
```

Adding EP11 tokens to openCryptoki

You need to introduce one or multiple tokens of type EP11 (EP11 tokens) into the openCryptoki library. For this purpose, you must define a slot entry for each desired token in the global openCryptoki configuration file called opencryptoki.conf.

If you want to configure multiple EP11 tokens, you can assign dedicated adapters and domains to different tokens respectively. This ensures data isolation between multiple applications.

If you use multiple EP11 tokens, you must specify a unique token directory in the slot entry for each token, using the tokname attribute. This token directory receives the token-individual information (like for example, key objects, user PIN, SO PIN, or hashes). Thus, the information for a certain EP11 token is separated from other EP11 tokens.

The default EP11 token directory is /var/lib/opencryptoki/ep11tok/. You can use the default only for a single EP11 token. Resulting examples for multiple EP11 token directories can be:

```
/var/lib/opencryptoki/ep11token1/
/var/lib/opencryptoki/ep11token2/
```

Note: A certain token configuration applies to all applications that use this EP11 token.

Adding a slot entry for each applicable EP11 token in opencryptoki.conf

As already mentioned, the default openCryptoki configuration file opencryptoki.conf provides a slot entry for the EP11 token. It is preconfigured to slot #4. Each slot entry must set the stdll attribute to libpkcs11_ep11.so. Check this default entry to find out whether you can use it as is.

For each configured EP11 token, you must create a specific EP11 token configuration file. This EP11-specific configuration file defines the target adapters and target adapter domains to which the EP11 token sends its cryptographic requests.

In turn, each slot entry in the global openCryptoki configuration file must specify this EP11 token configuration file. For this purpose, use the confname attribute with the unique name of the respective EP11 token configuration file as value.

The example from Figure 36 on page 40 configures two EP11 tokens in slots 4 and 5. It defines the names of the specific token configuration files to be ep11tok01.conf and ep11tok02.conf. Per default, these files are searched in the directory where openCryptoki searches its global configuration file. Figure 39 on page 44 shows an example of an EP11 token configuration file.

```
slot 4
{
stdll = libpkcs11_ep11.so
confname = ep11tok01.conf
tokname = ep11token01
description = "Ep11 Token"
manufacturer = "IBM"
hwversion = "4.11"
firmwareversion = "2.0"
}

slot 5
{
stdll = libpkcs11_ep11.so
confname = ep11tok02.conf
tokname = ep11token02
}
```

Figure 36. Multiple EP11 token instances

Setting an option for a FIPS compliant token data format

Starting with openCryptoki version 3.12, you can optionally use only FIPS compliant operations for openCryptoki's login password hashing and for encrypting the stored token data. This is valid for EP11 tokens, libica tokens, CCA tokens and Soft Tokens. Being FIPS compliant, for a new token directory, the token data is stored in a format that is better protected against attacks than the previously used data format.

If you want to use the token data format that was generated with FIPS compliant operations, you must explicitly specify the `tokversion` option for the token's slot entry in the openCryptoki configuration file. You must do this before token initialization with the **`pkcsconf`** command, for example:

```
slot 4
{
stdll = libpkcs11_ep11.so
confname = ep11tok01.conf
tokname = ep11token01
tokversion = 3.12
description = "Ep11 Token"
manufacturer = "IBM"
hwversion = "4.11"
firmwareversion = "2.0"
}
```

Figure 37. Slot entry for an EP11 token with FIPS compliant data format in the opencryptoki.conf file

You can use the **`pkcstok_migrate`** utility to transform an EP11 token, a CCA token, an ICA token, or a Soft Token created with any version of openCryptoki into a data format that was generated by FIPS compliant operations.

The **`pkcstok_migrate`** tool converts all token data produced with any openCryptoki version, including PINs, to be encrypted with a FIPS compliant method. Without this tool, switching to the new token format is only possible with an empty repository. The new FIPS compliant format can be used by specifying the `tokversion` keyword in the token’s slot configuration in `opencryptoki.conf` as shown in Figure 37 on page 40. For a value of `3.12` or greater, the new format is used. Values lower than `3.12` are invalid. To ensure compatibility with key objects generated using older versions of openCryptoki, the old format is still the default. The new format is only used when the user explicitly adds the `tokversion` keyword to the `opencryptoki.conf` file.

For information on how to use this tool, see “Migrating to FIPS compliance using the pkcstok_migrate tool” on page 66.

Defining an EP11 token-specific configuration file

One default configuration file for the EP11 token called `ep11tok.conf` is delivered by openCryptoki. You must adapt it according to your installation's system environment. If you use multiple EP11 tokens, you must provide an individual token configuration file for each token. Each slot entry in the global configuration file `opencryptoki.conf` defines these configuration file names.

In the example from Figure 36 on page 40, these names are defined as `ep11tok01.conf` and `ep11tok02.conf`. If the environment variable `OCK_EP11_TOKEN_DIR` is set, then the EP11 token looks for the configuration file or files in the directory specified with this variable. If `OCK_EP11_TOKEN_DIR` is not set, then the EP11 token configuration files are searched in the global openCryptoki directory, for example: `/etc/opencryptoki/ep11tok.conf`.

Example: If a slot entry in `opencryptoki.conf` specifies `confname = ep11tok02.conf`, and you set the environment variable `OCK_EP11_TOKEN_DIR` like:

```
export OCK_EP11_TOKEN_DIR=/home/user/ep11token
```

then your EP11 token configuration file appears here:

```
<root>/home/user/ep11token/ep11tok02.conf
```

You can use the shown example to set your own token directory for test purposes.

Note: The setting of this environment variable is ignored, if a program trying to access the designated EP11 token is marked with file permission `setuid`.

The following is a list of available options for an EP11 token configuration file.

APQN_WHITELIST

Because different EP11 hardware security modules (HSM) can use different wrapping keys (referred to as master keys in the TKE environment), users need to specify which HSM, in practice an adapter/domain pair, can be used by the EP11 token as a target for cryptographic requests. Therefore, an EP11 token configuration file contains a list of adapter/domain pairs to be used.

You start this list of adapter/domain pairs starting with a line containing the keyword `APQN_WHITELIST`. Next follows the list which can specify up to 512 adapter/domain pairs, denoted by decimal numbers in the range 0 - 255. Each pair designates an adapter (first number) and a domain (second number) accessible to the EP11 token. Close the list using the keyword END.

Alternatively, you can use the keyword `APQN_ANY` to define that all adapter/domain pairs with EP11 firmware, that are available to the system, can be used as target adapters. This is the default.

Notes:

- The term *APQN* stands for adjunct processor queue number. It designates the combination of a cryptographic coprocessor (adapter) and a domain, a so-called adapter/domain pair. At least one adapter/domain pair must be specified.
- If more than one APQN is used by a token, then these APQNs must be configured with the same master key.

An adapter/domain pair is displayed by the **`lszcrypt`** tool or in the `sys` file system (for example, in `/sys/bus/ap/devices`) in the form *card*.*domain*, where both numbers are displayed in hexadecimal format.

There are two ways to specify the cryptographic adapter:

- either as an explicit list of adapter/domain pairs:

```
APQN_WHITELIST
 8 13
 10 13
END
```

The adapter and domain can be given in decimal, octal (with leading 0), or hexadecimal (with leading 0x) notation:

```
APQN_WHITELIST
 8 0x0d
 0x0a 13
END
```

Valid adapter and domain values are in the range 0 to 255.

- or as any available cryptographic adapters:

```
APQN_ANY
```

In the example from Figure 39 on page 44, adapter 0 with domains 0 and 1, and adapter 2 with domain 84 are specified as target for requests from the EP11 token. In Figure 38 on page 42, these adapter/domain pairs are shown in hexadecimal notation as APQNs (00,0000), (00,0001), and (02,0054).

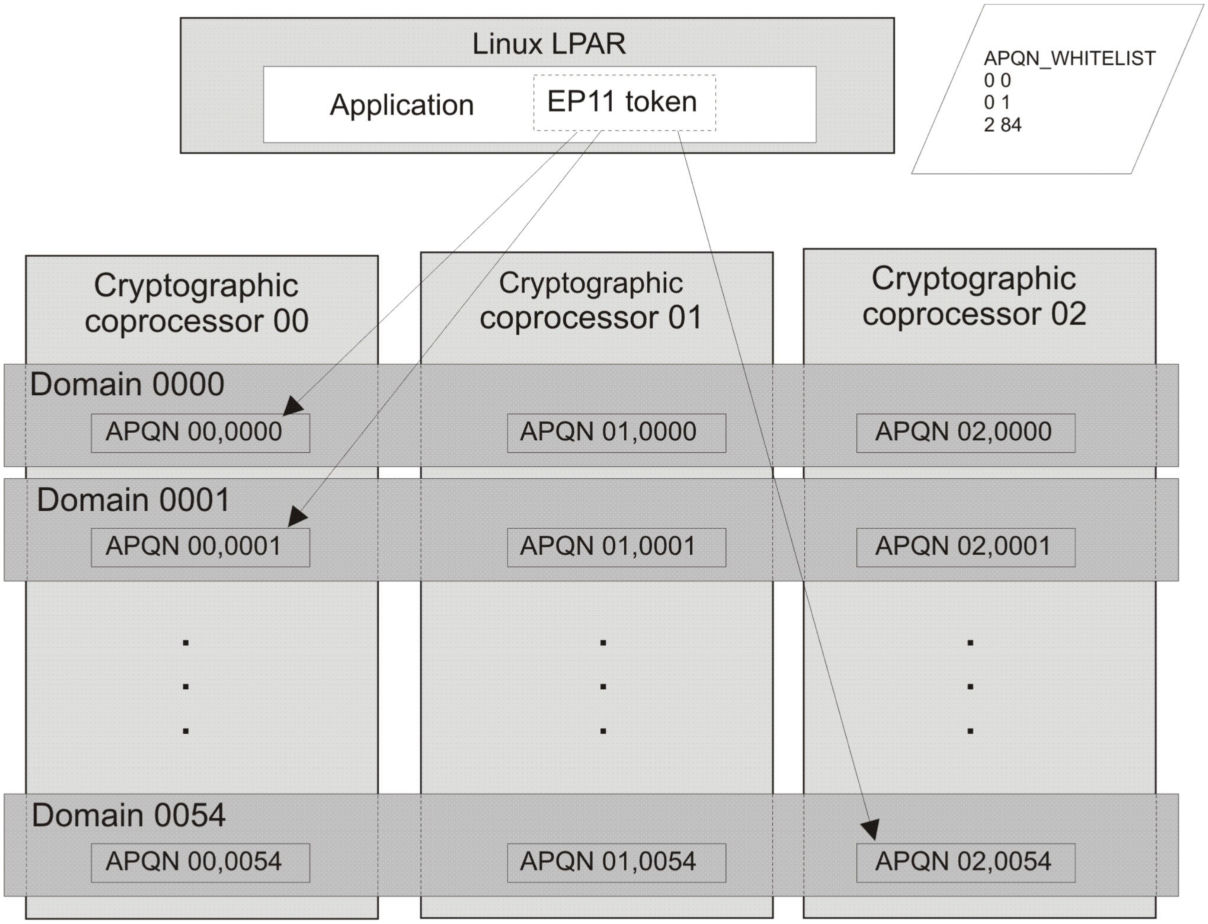


Figure 38. Cryptographic configuration for an LPAR

FORCE_SENSITIVE
Specify this option to force that the default for CKA_SENSITIVE is CK_TRUE for secret keys. For more information, see “Restrictions with using the EP11 library functions” on page 58.

STRICT_MODE
In strict-mode, all session-keys strictly belong to the PKCS #11 session that created it. When the PKCS #11 session ends, all session keys created for this session can no longer be used.

For more information, read “Controlling access to cryptographic objects” on page 56.

VHSM_MODE

In VHSM-mode (virtual-HSM), all keys generated by the EP11 token strictly belong to the EP11 token that created it. Every EP11 token running in this mode requires a VHSM card-PIN which must be set using the **pkcsep11_session** tool.

For more information, read “Controlling access to cryptographic objects” on page 56 and “Managing EP11 sessions with the pkcsep11_session tool” on page 65.

CPFILTER

The list of mechanisms returned by `C_GetMechanismList` is filtered using the domain or access control point (ACP) settings of the used cryptographic coprocessors. The EP11 access control point filter configuration file (ACP-filter configuration file) is used to associate certain access (domain) control points with mechanisms that are dependent on these access control points. The default ACP-filter configuration file is `ep11cpfilter.conf` located in the same directory as this EP11 token configuration file. You can optionally specify the name or location, or both, of the ACP-filter file:

```
CPFILTER /etc/opencryptoki/ep11cpfilter.conf
```

For more information, read “Filtering mechanisms” on page 51.

OPTIMIZE_SINGLE_PART_OPERATIONS

Set this option to optimize the performance of single part sign- and verify-operations, as well as of single part encrypt- or decrypt-operations. Then the `init` call is not passed through the EP11 library as long as there is no corresponding multi-part operation.

When this option is enabled, error handling can be slightly different, when errors from the deferred `init` call are presented during the first update call or during the calls to `C_Sign`, `C_Verify`, `C_Encrypt`, or `C_Decrypt` for a single part operation. That is, the first update call on a multi part operation or the mentioned calls for a single part operation may return errors, which are usually not returned by the update call. Such errors may be for example:

CKR_OBJECT_HANDLE_INVALID
CKR_ATTRIBUTE_VALUE_INVALID
CKR_KEY_HANDLE_INVALID
CKR_KEY_SIZE_RANGE
CKR_KEY_TYPE_INCONSISTENT
CKR_MECHANISM_INVALID
CKR_MECHANISM_PARAM_INVALID

DIGEST_LIBICA *<libica-path>* | DEFAULT | OFF

To improve the performance of required hash functions, the EP11 token on initialization loads the default libica library. If required, the EP11 token invokes the libica SHA-based hash functions, because the libica library performs these hash functions on the CPACF, thus avoiding hash processing on a cryptographic coprocessor which results in I/O operations to the adapter.

libica provides an OpenSSL based software fall-back, in case CPACF or a certain hashing function of CPACF is not available. In case a libica operation fails, because neither the hardware nor the software support is available, or if libica is not available at all, then the request is passed to the EP11 library instead.

With the DIGEST_LIBICA option, you can control which libica library is loaded:

DEFAULT

The default libica library is loaded. If libica could not be found, a message is issued to `syslog`, and all hash based functions will use the EP11 library.

The same behavior is applied if the DIGEST_LIBICA option is not specified at all.

<libica-path>

The specified library is loaded. If it can not be found, a message is issued to `syslog`, and token initialization fails.

OFF

No libica is loaded, and all hash based functions use the EP11 library.

If DIGEST_LIBICA is not specified, then the default libica library is loaded (same behavior as for DIGEST_LIBICA DEFAULT).

USE_PRANDOM

Set this option to control from where the EP11 token reads random data. With USE_PRANDOM specified, the EP11 token reads random data from `/dev/prandom`, or from `/dev/urandom` if `/dev/prandom` is not available. The default is to read the random data using the `m_GenerateRandom` function from the Crypto Express EP11 coprocessor.

Sample of an EP11 token configuration file

```
#
# EP11 token configuration
#
APQN_WHITELIST
0 0
0 1
2 84
END
FORCE_SENSITIVE
STRICT_MODE
VHSM_MODE
CPFILTER /etc/opencryptoki/ep11cpfilter.conf
OPTIMIZE_SINGLE_PART_OPERATIONS
DIGEST_LIBICA DEFAULT
USE_PRANDOM
```

Figure 39. Sample of an EP11 token configuration file

Setting environment variables

To customize your EP11 enablement, you can set environment variables. Setting environment variables overrides any settings from the configuration file.

The following variables are available:

OCK_EP11_TOKEN_DIR

specifies a directory for all available EP11 token configuration files. If multiple configuration files are available, they must all be located below this directory.

The default directory for the EP11 token configuration files is `/etc/opencryptoki/`. This is the same directory where the openCryptoki configuration file `opencryptoki.conf` is stored.

Examples:

```
export OCK_EP11_TOKEN_DIR=/home/user/ep11token
export OCK_EP11_TOKEN_DIR=/var/lib/opencryptoki/
```

Notes:

- Objects belonging to a certain EP11 token are stored in a different directory specified by the **`tokname`** attribute in `opencryptoki.conf`.
- The setting of this environment variable is ignored, if a program trying to access the designated EP11 token is marked with file permission `setuid`.

OPENCRYPTOKI_TRACE_LEVEL

defines the granularity of logging support. Valid values are between 0 and 5. For information about log levels, read topic "Enabling the logging support while running the EP11 token " on page 62.

Example:

```
export OPENCRYPTOKI_TRACE_LEVEL=2
```

Initializing EP11 tokens

Once the openCryptoki configuration file and the configuration files of the EP11 tokens are set up, and the **pkcsslotd** daemon is started, the EP11 token instances must be initialized.

Note: PKCS #11 defines two users for each EP11 token instance: 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 log-in PIN is defined as part of the token initialization.

The following command provides some useful slot information:

```
# pkcsconf -s

Slot #0 Info
        Description: EP11 Token
        Manufacturer: IBM
        Flags: 0x1 (TOKEN_PRESENT)
        Hardware Version: 4.0
        Firmware Version: 2.11
Slot #1 Info
        Description: ICA Token
        Manufacturer: IBM
        Flags: 0x1 (TOKEN_PRESENT)
        Hardware Version: 4.0
        Firmware Version: 2.10
```

Find your preferred token instance 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, because you might need this reference for identification purposes.

pkcsconf -P
For security reasons, openCryptoki requires 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
You must at least once change the user PIN with `pkcsconf -p` option. After you completed the PIN setup, the token is prepared and ready for use.

Note: Define a user PIN that is different from 12345678, because this pattern is checked internally and marked as default PIN. A log-in attempt with this user PIN is recognized as *not initialized*.

How to recognize an EP11 token

You can use the **pkcsconf -t** command to display a table that shows all available tokens. You can check the slot and token information, and the PIN status at any time.

The following information provided by the **pkcsconf -t** command about the EP11 token is returned in the *Token Info* section, where, for example, `Token #1 Info` displays information about the token plugged into slot number 1.

```
$ pkcsconf -t

Token #1 Info:
       Label: ep11
    Manufacturer: IBM Corp.
    Model: IBM EP11Tok
    Serial Number: 123
    Flags: 0x880445
          (RNG|LOGIN_REQUIRED|CLOCK_ON_TOKEN|TOKEN_INITIALIZED|USER_PIN_TO_BE_CHANGED
           |SO_PIN_TO_BE_CHANGED)
    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: 15:29:43
```

The most important information is as follows:

- The token **`Label`** you assigned at the initialization phase (ep11, in the example). You can initialize or change a token label by using the `pkcsconf -I` command.
- The **`Model`** name is unique and designates the token that is in use.
- The **`Flags`** provide information about the token initialization status, the PIN status, and features such as ***Random Number Generator*** (RNG). They also provide information about requirements, such as ***Login required***, which means that there is at least one mechanism that requires a session log-in to use that cryptographic function. For example, the mask for TOKEN_INITIALIZED is 0x00000400 and it is true, if the token has been initialized.

  The flag USER_PIN_TO_BE_CHANGED indicates that the user PIN must be changed before the token can be used. The flag SO_PIN_TO_BE_CHANGED indicates that the SO PIN must be changed before the token can be used.

  For more information about the flags provided in this output, see the description of the TOKEN_INFO structure and the Token Information Flags in the PKCS #11 Cryptographic Token Interface Standard.
- The **`PIN length`** range declared for this token.

Chapter 5. Using an EP11 token

You can take advantage of the EP11 library functions by using 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 are also able to use the Linux on Z EP11 enablement.

An EP11 token plugged into openCryptoki works only on IBM Z hardware, with further prerequisites as described in this publication. You can configure multiple EP11 tokens within the global openCryptoki configuration file. Thus you can assign dedicated adapters and domains to different tokens to ensure data isolation between applications. For more information refer to "Adding EP11 tokens to openCryptoki" on page 39.

openCryptoki implements the *PKCS #11 Baseline Provider* specification. A library implementing PKCS #11 according to the standards of the **Baseline Provider Clause** is called a *PKCS #11 Baseline Provider*. Such a provider has the ability to provide information about its cryptographic services.

A *PKCS #11 Baseline Provider* library can be exploited by an application conforming to the **Baseline Consumer Clause**. Such an application is therefore called a *PKCS #11 Baseline Consumer*. A *Baseline Consumer* calls a *Baseline Provider* implementation of the PKCS #11 API in order to use the cryptographic functionality from that provider. Thus, at run-time, a consumer can query information about a provider, for example, about the offered cryptographic services.

For detailed information about the conformance of a *PKCS #11 Baseline Consumer* and of a *PKCS #11 Baseline Provider* read PKCS #11 Cryptographic Token Interface Profiles Version 3.0.

Supported mechanisms for EP11 tokens

View a list of the supported mechanisms for the EP11 token in the openCryptoki implementation.

Use the **pkcsconf** command with the shown parameters to retrieve a complete list of algorithms (or mechanisms) that are supported by the token:

```
$ pkcsconf -m -c <slot>
Mechanism #2
        Mechanism: 0x131 (CKM_DES3_KEY_GEN)
        Key Size: 24-24
        Flags: 0x8001 (CKF_HW|CKF_GENERATE)
…
Mechanism #10
        Mechanism: 0x132 (CKM_DES3_ECB)
        Key Size: 24-24
        Flags: 0x60301 (CKF_HW|CKF_ENCRYPT|CKF_DECRYPT|CKF_WRAP|CKF_UNWRAP)
Mechanism #11
        Mechanism: 0x133 (CKM_DES3_CBC)
        Key Size: 24-24
        Flags: 0x60301 (CKF_HW|CKF_ENCRYPT|CKF_DECRYPT|CKF_WRAP|CKF_UNWRAP)
...
```

The list displays all mechanisms that are supported by this token. The mechanism ID and name corresponds to the PKCS #11 specification. Each mechanism provides its supported key size and some further properties such as hardware support and mechanism information flags. These flags provide information about the PKCS #11 functions that may use the mechanism. In some cases, the flags also provide further attributes that describe the supported variants of the mechanism. Typical functions are for example, *encrypt*, *decrypt*, *wrap key*, *unwrap key*, *sign*, or *verify*.

On an Crypto Express EP11 coprocessor which is configured to support all applicable PKCS #11 mechanisms from openCryptoki version 3.10, the EP11 token can exploit the mechanisms listed in Table 2 on page 48:

Table 2. PKCS #11 mechanisms supported by the EP11 token

Mechanism	Key sizes (in bits)	Properties
CKM_AES_CBC	16,24,32	ENCRYPT,DECRYPT, WRAP,UNWRAP
CKM_AES_CBC_PAD	16,24,32	ENCRYPT,DECRYPT, WRAP,UNWRAP
CKM_AES_CMAC	16,24,32	SIGN,VERIFY
CKM_AES_CMAC_GENERAL	16,24,32	SIGN,VERIFY
CKM_AES_ECB	16,24,32	ENCRYPT,DECRYPT
CKM_AES_KEY_GEN	16,24,32	GENERATE
CKM_DES2_KEY_GEN	16	GENERATE
CKM_DES3_CBC	16,24	ENCRYPT,DECRYPT, WRAP,UNWRAP
CKM_DES3_CBC_PAD	16,24	ENCRYPT,DECRYPT, WRAP,UNWRAP
CKM_DES3_CMAC	16,24	SIGN,VERIFY
CKM_DES3_CMAC_GENERAL	16,24	SIGN,VERIFY
CKM_DES3_ECB	16,24	ENCRYPT,DECRYPT
CKM_DES3_KEY_GEN	24	GENERATE
CKM_DH_PKCS_DERIVE	1024-3072	DERIVE
CKM_DH_PKCS_KEY_PAIR_GEN	1024-3072	GENERATE_KEY_PAIR
CKM_DH_PKCS_PARAMETER_GEN	1024-3072	GENERATE
CKM_DSA	1024-3072	SIGN,VERIFY
CKM_DSA_KEY_PAIR_GEN	1024-3072	GENERATE_KEY_PAIR
CKM_DSA_PARAMETER_GEN	1024-3072	GENERATE
CKM_DSA_SHA1	1024-3072	SIGN,VERIFY
CKM_EC_KEY_PAIR_GEN	192,521	GENERATE_KEY_PAIR, EC_F_P, EC_NAMEDCURVE, EC_UNCOMPRESS
CKM_ECDH1_DERIVE **[1]**	192,521	DERIVE, EC_F_P, EC_UNCOMPRESS
CKM_ECDSA	192,521	SIGN,VERIFY, EC_F_P, EC_NAMEDCURVE, EC_UNCOMPRESS
CKM_ECDSA_KEY_PAIR_GEN	192,521	GENERATE_KEY_PAIR, EC_F_P, EC_NAMEDCURVE, EC_UNCOMPRESS
CKM_ECDSA_SHA1	192,521	SIGN,VERIFY, EC_F_P, EC_NAMEDCURVE, EC_UNCOMPRESS

Table 2. PKCS #11 mechanisms supported by the EP11 token (continued)

Mechanism	**Key sizes (in bits)**	**Properties**
CKM_ECDSA_SHA224	192-521	SIGN,VERIFY, EC_F_P, EC_NAMEDCURVE, EC_UNCOMPRESS
CKM_ECDSA_SHA256	192-521	SIGN, VERIFY, EC_F_P, EC_NAMEDCURVE, EC_UNCOMPRESS
CKM_ECDSA_SHA384	192-521	SIGN,VERIFY, EC_F_P, EC_NAMEDCURVE, EC_UNCOMPRESS
CKM_ECDSA_SHA512	192-521	SIGN,VERIFY, EC_F_P, EC_NAMEDCURVE, EC_UNCOMPRESS
CKM_IBM_CMAC	16,32	SIGN,VERIFY
CKM_IBM_DILITHIUM **[2]**	256	SIGN,VERIFY, GENERATE_KEY_PAIR
CKM_IBM_EC_C25519	256	DERIVE, EC_F_P, EC_UNCOMPRESS
CKM_IBM_EC_C448	448	DERIVE, EC_F_P, EC_UNCOMPRESS
CKM_IBM_ED25519_SHA512	256	SIGN,VERIFY, EC_F_P, EC_UNCOMPRESS
CKM_IBM_ED448_SHA3	448	SIGN,VERIFY, EC_F_P, EC_UNCOMPRESS
CKM_IBM_EDDSA_SHA512	n/a	SIGN,VERIFY
CKM_IBM_SHA3_224	n/a	DIGEST
CKM_IBM_SHA3_224_HMAC	112-256	SIGN,VERIFY
CKM_IBM_SHA3_256	n/a	DIGEST
CKM_IBM_SHA3_256_HMAC	128-256	SIGN,VERIFY
CKM_IBM_SHA3_384	n/a	DIGEST
CKM_IBM_SHA3_384_HMAC	192-256	SIGN,VERIFY
CKM_IBM_SHA3_512	n/a	DIGEST
CKM_IBM_SHA3_512_HMAC	256	SIGN,VERIFY
CKM_PBE_SHA1_DES3_EDE_CBC	24	GENERATE
CKM_RSA_PKCS	1024-4096	ENCRYPT,DECRYPT, SIGN,VERIFY, WRAP,UNWRAP
CKM_RSA_PKCS_KEY_PAIR_GEN	1024-4096	GENERATE_KEY_PAIR
CKM_RSA_PKCS_OAEP **[3]**	1024-4096	ENCRYPT,DECRYPT, WRAP,UNWRAP
CKM_RSA_PKCS_PSS	1024-4096	SIGN,VERIFY

Table 2. PKCS #11 mechanisms supported by the EP11 token (continued)

Mechanism	Key sizes (in bits)	Properties
CKM_RSA_X9_31	1024-4096	SIGN, VERIFY
CKM_RSA_X9_31_KEY_PAIR_GEN	1024-4096	GENERATE_KEY_PAIR
CKM_SHA_1	n/a	DIGEST
CKM_SHA_1_HMAC	80-256	SIGN,VERIFY
CKM_SHA1_KEY_DERIVATION	n/a	DERIVE
CKM_SHA1_RSA_PKCS	1024-4096	SIGN,VERIFY
CKM_SHA1_RSA_PKCS_PSS	1024-4096	SIGN,VERIFY
CKM_SHA1_RSA_X9_31	1024-4096	SIGN,VERIFY
CKM_SHA224	n/a	DIGEST
CKM_SHA224_HMAC	112-256	SIGN, VERIFY
CKM_SHA224_HMAC_GENERAL	80-2048	SIGN, VERIFY
CKM_SHA224_KEY_DERIVATION	n/a	DERIVE
CKM_SHA224_RSA_PKCS	1024-4096	SIGN,VERIFY
CKM_SHA224_RSA_PKCS_PSS	1024-4096	SIGN,VERIFY
CKM_SHA256	n/a	DIGEST
CKM_SHA256_HMAC	128-256	SIGN,VERIFY
CKM_SHA256_KEY_DERIVATION	n/a	DERIVE
CKM_SHA256_RSA_PKCS	1024-4096	SIGN,VERIFY
CKM_SHA256_RSA_PKCS_PSS	1024-4096	SIGN,VERIFY
CKM_SHA384	n/a	DIGEST
CKM_SHA384_HMAC	192-256	SIGN,VERIFY
CKM_SHA384_KEY_DERIVATION	n/a	DERIVE
CKM_SHA384_RSA_PKCS	1024-4096	SIGN,VERIFY
CKM_SHA384_RSA_PKCS_PSS	1024-4096	SIGN,VERIFY
CKM_SHA512	n/a	DIGEST
CKM_SHA512_224	n/a	DIGEST
CKM_SHA512_224_HMAC	112-256	SIGN,VERIFY
CKM_SHA512_224_HMAC_GENERAL	16,256	SIGN,VERIFY
CKM_SHA512_256	n/a	Digest
CKM_SHA512_256_HMAC	128-256	SIGN,VERIFY
CKM_SHA512_256_HMAC_GENERAL	16,256	SIGN,VERIFY
CKM_SHA512_HMAC	256	SIGN,VERIFY
CKM_SHA512_KEY_DERIVATION	n/a	DERIVE
CKM_SHA512_RSA_PKCS	1024-4096	SIGN,VERIFY

Table 2. PKCS #11 mechanisms supported by the EP11 token (continued)

Mechanism	Key sizes (in bits)	Properties
CKM_SHA512_RSA_PKCS_PSS	1024-4096	SIGN,VERIFY

[1]
With EP11 host library version 3, the CKM_ECDH1_DERIVE mechanism expects the CK_ECDH1_DERIVE_PARAMS structure as mechanism parameter, and thus also supports key derivation functions (KDFs) and shared data.

[2]
Dilithium 6-5 uses key sizes of 1760 bytes for a public key and 3856 bytes for a private key. Refer to https://pq-crystals.org/dilithium/index.shtml for details. These key sizes are comparable to 256 bits of security of classical algorithms.

[3]
Starting with IBM z15, firmware version 7.13 and an EP11 host library of at least 2.1, the CKM_RSA_PKCS_OAEP mechanism supports SHA2 and SHA3 as hashing algorithms and mask generation function (MGF) algorithms.

For explanation about the key object properties see the PKCS #11 Cryptographic Token Interface Standard.

Filtering mechanisms

You can obtain a filtered list of mechanisms, according to the current setting of the access control points (ACPs), while considering different firmware levels on the configured EP11 cryptographic coprocessors. That is, if multiple cryptographic coprocessors are assigned in the EP11 token configuration file, only the mechanisms accessible from the cryptographic coprocessor with the lowest firmware level are considered. That way only those mechanisms are returned which you can really use.

An EP11 cryptographic coprocessor can be configured by means of ACPs, which restrict the use of certain mechanisms within a domain of this coprocessor. Restricted mechanisms are not visible in the mechanism list returned. Some ACPs restrict certain attributes of mechanisms (such as key sizes). Such restrictions are not reflected in the returned list.

You can use functions `C_GetMechanismList`, `C_GetMechanismInfo`, as well as the command **`pkcsconf -m -c n`** to produce a list of accessible mechanisms. Function `C_GetMechanismInfo` returns `CKR_MECHANISM_INVALID` when the examined mechanism is restricted by the current ACP settings.

Since the EP11 token can use multiple EP11 cryptographic coprocessors and also multiple domains on a coprocessor, the access control points of all cryptographic coprocessors and domains are queried. Ideally, all coprocessors and domains should have identical ACP settings. If differences are detected, a message is issued to the SYSLOG, and the minimum ACP setting of all coprocessors and domains is used.

If multiple cryptographic coprocessors with different firmware levels are used, then only the mechanisms allowed by the ACP setting from the cryptographic coprocessor with the lowest firmware level are returned.

When the EP11 token configuration file specifies the APQN_ANY keyword, then the `sysfs` directories under `/sys/devices/ap/` are scanned to find all available EP11 cryptographic coprocessors. If an APQN_WHITELIST is specified, then only those coprocessors specified in the white-list are used.

The associations of access control points with certain mechanisms are read from the so-called access control point filter configuration file. This allows to change the associations easily, when additional mechanisms or ACPs are added.

A default access control point filter configuration file is provided as part of the EP11 token. If the access control point filter configuration file is not found or is empty, then no ACP filter is applied.

The access control point filter configuration file that contains the definitions for the ACP filter is called `ep11cpfilter.conf` per default, and is located at the same place as the EP11 token configuration files. To choose a different name or location of the access control point filter configuration file, specify the CPFILTER keyword followed by the name and path of the access control point filter configuration file in the EP11 token configuration file, for example:

```
CPFILTER /etc/opencryptoki/ep11cpfilter.conf
```

In the access control point filter configuration file, each line specifies the number of an ACP, followed by a colon and a comma-separated list of mechanisms that are not available when this ACP is not set:

```
<cp>: <mech1, mech2, ...>
```

Both, the ACP number and the mechanisms can be specified as name, in decimal, octal (with leading 0), or hexadecimal (with leading 0x), for example:

```

XCP_CPB_SIGN_SYMM: CKM_SHA256_HMAC, CKM_SHA256_HMAC_GENERAL
4: 0x00000251, 0x00000252
```

The shown example filters out the mechanisms CKM_SHA256_HMAC (0x00000251) and CKM_SHA256_HMAC_GENERAL (0x00000252) when access control point (ACP) XCP_CPB_SIGN_SYMM (3) or XCP_CPB_SIGVERIFY_SYMM (4) is not set. In the first line, both the ACP and the affected mechanisms are specified as name, and in the second line, the ACP is specified by a decimal number and the mechanisms are identified by their hexadecimal values.

A list of ACPs can be found in document Enterprise PKCS#11 (EP11) Library structure (Table 11: Control points) that is provided as part of the EP11 library.

Note that you must not change the access control point filter configuration file shipped with openCryptoki unless you are advised to do so by IBM.

The access control point filter configuration file is read once during token initialization (that is, within the **`C_Initialize`** function). It is kept in memory for the whole lifetime of the token. When changes are made to this configuration file, the token must be terminated and initialized again to have it read the file and activate the changes.

Note: Do not disable access control point 13 (generate or derive symmetric keys including DSA parameters), because during token initialization, the EP11 token uses mechanism **`CKM_AES_KEY_GEN`**, which is dependent on this ACP 13.

Importing keys

You can import keys of type CKK_AES, CKK_3DES, CKK_DSA, CKK_RSA, CKK_DH, and CKK_EC from plain text key values using function `C_CreateObject`. The resulting key objects are secure (CKA_SENSITIVE `= CK_TRUE and CKA_ALWAYS_SENSITIVE = CK_FALSE`).

To import keys of type CKK_AES, CKK_DES2, CKK_3DES, and CKK_GENERIC_SECRET, you must provide a template that contains the following attributes:

- CKA_VALUE

To import keys of type CKK_DSA, you must provide a template that contains the following attributes:

- CKA_PRIME (also called p)
- CKA_SUBPRIME (also called q)
- CKA_BASE (also called g)
- CKA_VALUE (private key x or public key y)

To import keys of type CKK_RSA, you must provide a template that contains the following attributes:

- CKA_MODULUS
- CKA_PUBLIC_EXPONENT

- CKA_PRIVATE_EXPONENT (for private key import only)
- CKA_PRIME_1 (for private key import only)
- CKA_PRIME_2 (for private key import only)
- CKA_EXPONENT_1 (for private key import only)
- CKA_EXPONENT_2 (for private key import only)
- CKA_COEFFICIENT (for private key import only)

To import keys of type CKK_DH, you must provide a template that contains the following attributes:

- CKA_PRIME (also called p)
- CKA_BASE (also called g)
- CKA_VALUE (private key x or public key y)

To import keys of type CKK_EC (synonym CKK_ECDSA), you must provide a template that contains the following attributes:

- CKA_EC_PARAMS
- CKA_EC_POINT
- CKA_VALUE (for private key import only)

Quantum safe cryptography with the EP11 token

The EP11 token offers features for quantum safe cryptography.

Quantum safe or post-quantum cryptography denotes cryptographic algorithms that resist attacks from classical as well as from quantum computers. The CRYSTALS-Dilithium Digital Signature Algorithm is a digital signature scheme and one of the candidate algorithms in the NIST Post-Quantum Cryptography Standardization Process.

In the EP11 token, the CRYSTALS-Dilithium algorithm provides security category *SHA384 / SHA3-384* and performance category *Dilithium-1536x1280 (also called Dilithium-6-5)*. On the TKE workstation, you must enable Dilithium by setting domain (access) control point 65 on the used cryptographic coprocessors:

```
65    XCP_CPB_ALG_PQC_DILITHIUM          enable support for Dilithium algorithm
```

Because Dilithium keys can only sign or verify, the EP11 token only provides one single mechanism for all three operations: key generation, sign, and verify: CKM_IBM_DILITHIUM (see also Table 2 on page 48).

With the EP11 token, you can also import and export Dilithium keys by wrapping or unwrapping them using AES or TDES key encrypting keys (KEKs). That is, you can protect Dilithium keys that are sent to another system, received from another system, or stored with data in a file.

Restrictions for using Dilithium keys

- IBM Dilithium keys cannot actively be used to transport (wrap and unwrap) other keys, but they can be transported using standard key types (AES, TDES).
- IBM Dilithium keys cannot be derived from given keys. They can only be generated or imported from given key values.

Supported curves with elliptic curve cryptography in the EP11 token

View a list of curves that are supported by the EP11 token for elliptic curve cryptography (ECC).

For the support of elliptic curve cryptography, the EP11 token provides standard mechanisms and IBM-specific mechanisms for key derivation and for sign and verify operations. For more information, refer to "Supported mechanisms for EP11 tokens" on page 47.

Table 3. PKCS #11 mechanisms supported by the EP11 token

Curve	**Purpose**
brainpoolP160r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
brainpoolP160t1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
brainpoolP192r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
brainpoolP192t1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
brainpoolP224r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
brainpoolP224t1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
brainpoolP256r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
brainpoolP256t1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
brainpoolP320r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
brainpoolP320t1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
brainpoolP1384r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
brainpoolP384t1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
brainpoolP512r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
brainpoolP512t1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
prime192v1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
prime256v1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE

Table 3. PKCS #11 mechanisms supported by the EP11 token (continued)

Curve	Purpose
secp224r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
secp256k1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
secp384r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
secp521r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH mit CKM_ECDH1_DERIVE
Montgomery curves, only for ECDH with certain IBM-specific mechanisms	
curve448	ECDH with CKM_IBM_EC_C448
curve25519	ECDH with CKM_IBM_EC_C25519
Edwards Curves, only for Sign/Verify (EdDSA) with certain IBM-specific mechanisms	
ed448	Sign/Verify with CKM_IBM_ED448_SHA3
ed25519	Sign/Verify with CKM_IBM_ED25519_SHA512

The EP11 host library provides access control point 55 to enable support of curve25519, c448, and related algorithms, including EdDSA:

```
55    XCP_CPB_ALG_EC_25519     enable support of curve25519, c448 and related algorithms
                               incl. EdDSA
```

Re-encrypting data with a mechanism

The vendor-specific function `C_IBM_ReencryptSingle` is available in openCryptoki and is supported by all tokens. You can use it to re-encrypt data encrypted with a given key and mechanism with another key and mechanism. This function is useful for secure key encryption with an EP11 token or a CCA token, because during the process, the data is never visible in the clear anywhere outside the cryptographic coprocessor.

The `C_IBM_ReencryptSingle` function has the following signature:

```
CK_RV C_IBM_ReencryptSingle(CK_SESSION_HANDLE hSession,
                            CK_MECHANISM_PTR pDecrMech,
                            CK_OBJECT_HANDLE hDecrKey,
                            CK_MECHANISM_PTR pEncrMech,
                            CK_OBJECT_HANDLE hEncrKey,
                            CK_BYTE_PTR pEncryptedData,
                            CK_ULONG ulEncryptedDataLen,
                            CK_BYTE_PTR pReencryptedData,
                            CK_ULONG_PTR pulReencryptedDataLen);
```

Because the new function is non-standard, it does not appear in the PKCS #11 `CK_FUNCTION_LIST` structure returned by `C_GetFunctionList()`. To invoke this function, you must either locate the desired function in the main DLL using `dlsym()`, or link the application program with the main DLL. You can also use `C_GetInterface()` to get the interface called `Vendor IBM`. This interface also provides the `C_IBM_ReencryptSingle` function.

Like other PKCS #11 functions, this function returns output in a variable-length buffer, conforming to the convention defined by PKCS #11.

If **pReencryptedData** is NULL_PTR, then the function only uses parameter ***pulReencryptedDataLen** to return a number of bytes which would suffice to hold the cryptographic output produced from the input to the function. This number may exceed the precise number of bytes needed, but not to a very high extent.

If **pReencryptedData** is not NULL_PTR, then ***pulReencryptedDataLen** must contain the size in bytes of the buffer pointed to by **pReencryptedData**. If that buffer is large enough to hold the cryptographic output produced by the function, then that cryptographic output is placed there, and **CKR_OK** is returned. If the buffer is not large enough, then **CKR_BUFFER_TOO_SMALL** is returned. In either case, ***pulReencryptedDataLen** is set to hold the exact number of bytes needed to hold the produced cryptographic output.

The function generally allows to specify any combination of decryption and encryption mechanisms. However, not all combinations work with all data sizes. Mechanisms that do not perform any padding, require that the data to be encrypted is a multiple of the block size. Also some mechanisms have certain size limitations (for example, RSA). If the data size after decryption with the decryption mechanism conflicts with the requirements of the encryption mechanisms, then the re-encrypt operation may fail with **CKR_DATA_LEN_RANGE**. Also, not all tokens may support all mechanism combinations. **CKR_MECHANISM_INVALID** is returned if one of the mechanisms specified is not supported for the re-encrypt operation.

Supporting the BSICC2017 compliance mode

The EP11 token provides an access control point (ACP) to enable the BSICC2017 compliance mode. When enabled, this compliance mode disables the RSA PKCS #1 v1.5 mechanisms.

The EP11 host library supports access control point 61 which is related to the BSICC2017 compliance mode.

```
61    XCP_CPB_ALG_NBSI2017            enable the BSICC2017 compliance mode
```

This ACP can be used to disable the following RSA PKCS #1 v1.5 mechanisms:

- CKM_RSA_PKCS
- CKM_SHA1_RSA_PKCS
- CKM_SHA224_RSA_PKCS
- CKM_SHA256_RSA_PKCS
- CKM_SHA384_RSA_PKCS
- CKM_SHA512_RSA_PKCS

Controlling access to cryptographic objects

You can decide to activate one or two session modes to limit the access to cryptographic objects in order to improve security. The available session modes are the *strict session mode* or the *virtual HSM (VHSM) mode*. Both of these modes generate an EP11 session. An EP11 session is a state on the EP11 cryptographic coprocessor and must not be confused with a PKCS #11 session.

You must configure the EP11 token to use either one of the available modes, or both.

- Protecting cryptographic objects with the *strict session mode*:

  This mode prohibits that a session key, copied from a PKCS #11 session that generated this key, is still valid even if the generating session has ended. Also, when this mode is used, session keys generated with this token can no longer be passed to other sessions of the same token.

  To enable the *strict session mode*, specify keyword STRICT_MODE in the EP11 token configuration file.
- Protecting cryptographic objects with the *virtual HSM (VHSM) mode*:

This mode is applicable when the same EP11 cryptographic coprocessors are used by multiple EP11 tokens. In such environments, a key generated by one token might be used by another token which uses the same domain on multiple EP11 cryptographic coprocessors. This is possible, because all keys are wrapped by the same master key. This mode prohibits that a key, generated by one token, is used by any other token (on the same system or on another system or z/VM or KVM guest).

To enable the *virtual HSM (VHSM) mode*, specify keyword `VHSM_MODE` in the configuration file. When this mode is used, session and token keys generated with this token can no longer be passed to other tokens using domains on the same EP11 cryptographic coprocessors.

The number of simultaneously supported EP11 sessions is limited on an EP11 cryptographic coprocessor. Programs that use a large number of sessions simultaneously should not use the *strict session mode* or the *virtual HSM mode*, because otherwise the EP11 cryptographic coprocessor may run out of session resources.

Therefore, it is important to delete any finished EP11 session, that is no longer required, in particular when the program for which it was logged in, terminated unexpectedly. For the purpose of a required session cleanup, you can use the **`pkcsep11_session`** utility. With this tool you can delete EP11 sessions from the EP11 cryptographic coprocessors that are left over by programs that did not terminate normally (see "Managing EP11 sessions with the pkcsep11_session tool" on page 65).

The **`pkcsep11_session`** tool also allows to display any currently stored EP11 sessions, and can also log out of those sessions.

Note: *Strict session mode* and *virtual HSM (VHSM) mode* only work for R/W or R/O user sessions. For public sessions or security officer (SO) sessions, the *strict session mode* and the *virtual HSM (VHSM) mode* are not used.

Strict session mode

To enable a strict implementation of the PKCS #11 session semantics, a *strict session mode* is available on a per token instance basis.

In *strict session mode*, for each new PKCS #11 session, a unique EP11 session ID is generated. Then the EP11 session is logged in on all adapter/domain pairs (adjunct processor queue numbers, APQNs) that are configured for the token. For each logged-in session, the returned session pin-blob (derived from the EP11 session ID) is stored in a special token object in the token directory.

During further processing, all session keys (that is, objects where attribute CKA_TOKEN is CK_FALSE) are bound to the session pin-blob that was created when the session was logged on.

At session end, the EP11 session with regards to the PKCS #11 session is logged out on all APQNs belonging to the token. The token object representing this EP11 session ID is deleted from the token directory.

An APQN that comes online after an EP11 session has started, is also logged in internally if a request to that APQN is encountered.

If multiple APQNs are available in the system, then all pin-blobs returned by the EP11 session login for the individual APQNs must be equal. Otherwise an error is returned and logged in the SYSLOG.

Virtual HSM (VHSM) mode

To restrict keys to only that token that was used to generate it, the *virtual HSM mode* (*VHSM mode*) is available on a per-token basis.

Similar to the *strict session mode*, for each new PKCS #11 session, a unique EP11 session ID is generated. However, an additional card-PIN is required to log into an EP11 session in *VHSM mode*.

The card-PIN used with *VHSM mode* must be set using the command **`cardpin`** from the **`pkcsep11_session`** tool (see "Managing EP11 sessions with the pkcsep11_session tool" on page 65).

It takes a slot ID, the user PIN, and the card-PIN as input. The card-PIN is stored in a special token object in the token directory. The card-PIN must be between 8 and 16 characters in length.

That way, after setting the card-PIN, an EP11 session can log in on all configured APQNs. The returned pin-blob derived from the card-PIN is stored in a special token object in the token directory (separately from the token object for the card-PIN), the same way as for *strict session mode*.

During further processing, all keys (session keys as well as token keys) are bound to the pin-blob that was derived from the card-PIN when the session was logged in.

At session end, the EP11 session with regards to the PKCS #11 session is logged out from all APQNs and the token object representing this EP11 session ID is deleted from the token directory. In contrast to the session pin-blob, the card-PIN remains persistent and thus can be used for future sessions in *VHSM mode*.

When the *VHSM mode* is enabled, but no card-PIN has been set, then the PKCS #11 session login fails, and an appropriate message is logged to syslog.

Combined strict session mode and virtual HSM mode

The *strict session mode* can be combined with the *VHSM mode*. This binds all keys to the card-PIN, and additionally binds session keys (that is, objects where attribute CKA_TOKEN is FALSE) to the PKCS #11 session.

Strict session mode	VHSM mode	Session objects	Token objects
off	off	not bound	not bound
on	off	bound to PKCS #11 session	not bound
off	on	bound to token by card PIN	bound to token by card PIN
on	on	bound to PKCS #11 session and token by card PIN	bound to token by card PIN

Restrictions with using the EP11 library functions

In this topic, you find information about certain limitations of the EP11 library.

- The EP11 library implements the *secure key concept* (that is, a key is wrapped (encrypted) by a master key, which is kept within the EP11 adapter). That means, that EP11 key values are never accessible. The secure key concept ensures that clear keys never leave the hardware security module (HSM), which is the EP11 module part that is installed on the cryptographic coprocessor.

  Therefore, the EP11 token only knows sensitive secret keys (CKO_SECRET_KEY). However, the PKCS #11 standard defines the default value of attribute CKA_SENSITIVE to be CK_FALSE. Thus, for previous versions of the EP11 token, all applications must have the attribute value of CKA_SENSITIVE explicitly changed to CK_TRUE whenever an EP11 secret key had been generated, unwrapped, or build with C_CreateObject.

  Starting with the EP11 token for openCryptoki version 3.10, an option is implemented to change the default value of attribute CKA_SENSITIVE to be CK_TRUE for all secret keys created with the EP11 token. This applies to functions C_GenerateKey, C_GenerateKeyPair, C_UnwrapKey, and C_DeriveKey when creating key with CKA_CLASS = CKO_SECRET_KEY, if the attribute CKA_SENSITIVE is not explicitly specified in the template.

  To enable this option, you must specify keyword FORCE_SENSITIVE in the EP11 token configuration file, as shown in Figure 40 on page 59. Note that the semantics specified with the FORCE_SENSITIVE keyword matches the semantics used by z/OS® for EP11.

```
#
# EP11 token configuration
#
FORCE_SENSITIVE
#
APQN_WHITELIST
5 2
6 2
END
```

Figure 40. Sample of an EP11 token configuration file

- Keys leaving the hardware security module (HSM) are encrypted by the HSM master key (wrapping key) and come as binary large object (BLOB). In openCryptoki, objects can have special attributes that describe the key properties. Besides dedicated attributes defined by the application, there are some attributes defined as token-specific by openCryptoki.

  Table 4 on page 59 and Table 5 on page 59 show the EP11 token-specific attributes and their default values for private and secure keys.

  Table 4. Private key (CKO_PRIVATE_KEY) default attributes of the EP11 token

Private key attributes	**value**
CKA_SENSITIVE	CK_TRUE
CKA_EXTRACTABLE	CK_TRUE

  Table 5. Secret key (CKO_SECRET_KEY) default attributes of the EP11 token

Secret key attributes	**value**
CKA_EXTRACTABLE	CK_TRUE

- When you create keys the default values of the attributes CKA_ENCRYPT, CKA DECRYPT, CKA_VERIFY, CKA_SIGN, CKA_WRAP and CKA_UNWRAP are CK_TRUE. Note, no EP11 mechanism supports the Sign/Recover or Verify/Recover functions.

  Even if settings of `CKA_SENSITIVE`, `CKA_EXTRACTABLE`, or `CKA_NEVER_EXTRACTABLE` would allow accessing the key value, then openCryptoki returns `00..00` as key value (due to the secure key concept).

  For information about the key attributes, see the PKCS #11 Cryptographic Token Interface Standard.

- All RSA keys must have a public exponent (CKA_PUBLIC_EXPONENT) greater than or equal to 17.
- The Crypto Express EP11 coprocessor restricts RSA keys (primes and moduli) according to ANSI X9.31. Therefore, in the EP11 token, the lengths of the RSA primes (p or q) must be a multiple of 128 bits. Also, the length of the modulus (`CKA_MODULUS_BITS`) must be a multiple of 256.
- The mechanisms `CKM_DES3_CBC` and `CKM_AES_CBC` can only wrap keys, which have a length that is a multiple of the block size of DES3 or AES respectively. See the mechanism list and mechanism information (**`pkcsconf -m`**) for supported mechanisms together with supported functions and key sizes.
- The EP11 coprocessor adapter can be configured to restrict the cryptographic capabilities in order for the adapter to comply with specific security requirements and regulations. Such restrictions on the adapter impact the capability of the EP11 token (see also "Filtering mechanisms" on page 51).
- The PKCS #11 function `C_DigestKey()` is not supported by the EP11 library.

Restriction to extended evaluations

For openCryptoki versions up to 3.8, the EP11 token only supported those functions and mechanisms that are available on an adapter that is configured to comply to the extended evaluations. These extended evaluations meet public sector requirements with regard to both FIPS and Common Criteria certifications. For more details, see the *IBM z14 Technical Guide*.

Starting with the current version of the EP11 enablement, you can control the use of certain mechanisms within a domain of an EP11 cryptographic coprocessor by configuring this coprocessor by means of access control points (ACPs). So except for one restriction, the use of mechanisms is no longer restricted to the limitations imposed by the extended evaluations.

Read “Filtering mechanisms” on page 51 for information on how to manage the access to PKCS #11 mechanisms using ACPs. The available mechanisms and their attributes are then reflected by the openCryptoki functions `C_GetMechanismList` and `C_GetMechanismInfo`. However, there is one restriction on RSA mechanisms that cannot be reflected in the result of `C_GetMechanismInfo`: The `CKA_PUBLIC_EXPONENT` must have a value of at least 17.

Chapter 6. Troubleshooting EP11

Troubleshooting can provide helpful information, if problems occur while you work with the Linux on Z EP11 enablement.

The contained subtopics introduce different methods, which support troubleshooting:

- "Checking the device driver status" on page 61
- "Checking the EP11 token status" on page 61
- "Enabling the logging support while running the EP11 token " on page 62

Checking the device driver status

The first step of troubleshooting while working with the EP11 enablement may be to check the device driver status as described in this topic.

Use the **lszcrypt** command like shown to retrieve basic status information. Type **lszcrypt -V** to achieve an output similar to the following:

```
# lszcrypt -V

CARD.DOMAIN TYPE  MODE        STATUS  REQUEST_CNT   ...  REQUESTQ_CNT HW_TYPE Q_DEPTH FUNCTIONS
------------------------------------------------------------------------------------------------
02          CEX6A Accelerator online         6895                   0      11      08 0x6a000000
02.004c     CEX6A Accelerator online         6895                   0      11      08 0x6a000000
03          CEX6C CCA-Coproc  online         4627                   0      11      08 0x92000000
03.004c     CEX6C CCA-Coproc  online         4627                   0      11      08 0x92000000
05          CEX6P EP11-Coproc online         2284                   0      11      08 0x06000000
05.004c     CEX6P EP11-Coproc online         2284                   0      11      08 0x06000000
```

This call can be used to check whether the EP11 requests are sent to a specific crypto adapter.

For more information about using IBM cryptographic adapters with Linux on Z and LinuxONE, see *Device Drivers, Features, and Commands*, SC33-8411 available at

```
www.ibm.com/developerworks/linux/linux390/documentation_dev.html
```

Checking the EP11 token status

You can request information about the EP11 token status by using the **pkcsconf -t** command. The *Flags* entry shows the actual status flags for the token and whether the token is ready to be used. In the shown example, the SO PIN needs to be changed before the token can be used.

```
$ pkcsconf -t

Token #1 Info:
    ...
    Model: IBM EP11Tok
    ...
    Flags: 0x80044D
          (RNG|LOGIN_REQUIRED|USER_PIN_INITIALIZED|CLOCK_ON_TOKEN|TOKEN_INITIALIZED
              |SO_PIN_TO_BE_CHANGED)
 ...
 ...
    Time: 15:29:43
```

Enabling the logging support while running the EP11 token

Read about the tasks how to run the EP11 token with enabled logging support.

You can enable logging support by setting the environment variable OPENCRYPTOKI_TRACE_LEVEL. If the environment variable is not set, logging is disabled by default.

Table 6. EP11 log levels

Log level	Description
1	Log error messages.
2	Log warning messages.
3	Log informational messages.
4	Log development debug messages. These messages may help debug while developing PKCS #11 applications.
5	Log debug messages that are useful to openCryptoki developers. This level must be enabled via option --enable-debug in the **configure** script.

If a log level > 0 is defined in the environment variable OPENCRYPTOKI_TRACE_LEVEL, then log entries are written to file /var/log/opencryptoki/trace.<pid>. In this file name specification, <pid> denotes the ID of the running process that uses the EP11 token.

The log file is created with ownership *user*, and group *pkcs11*, and permission 640 (user: read, write; group: read only; others: nothing). For every application, which is using openCryptoki with the EP11 token, a new log file is created during token initialization. Prerequisite for a working EP11 stack is the existence of the EP11 coprocessor card and an appropriate device driver with EP11 support.

A log level > 3 is only recommended for developers.

How to avoid common mistakes

- Do not configure or use an EP11 token before the master key is set on the associated adapters. Otherwise, token initialization fails and an appropriate syslog message is issued.
- Do not let a user invoke openCryptoki who does not belong to the *pkcs11 group*. Be aware that adding a user to a new group does not change the group membership of users that were logged in before.

Chapter 7. Tools and utilities

Various tools and utilities are available which provide general information about EP11, which support you in migrating EP11 master keys, and which help you to manage EP11 sessions.

The available utilities are introduced in :

- **ep11info** in "Obtaining information about an EP11 environment with the ep11info tool" on page 63
- **pkcsep11_migrate** in "Migrating master keys with the pkcsep11_migrate tool" on page 63
- **pkcsep11_session** in "Managing EP11 sessions with the pkcsep11_session tool" on page 65

Obtaining information about an EP11 environment with the ep11info tool

Use the **ep11info** tool to obtain general information about EP11 cryptographic coprocessors and about the domains configured on the system. Especially the wrapping key identifier of the configured domains are shown.

The syntax of the **ep11info** tool is as shown:

```
ep11info [ -h | --help ] | [ -m <module_nr> | --module <module_nr> ]
         [ -d <domain_index> | --domain <domain_index> ]
```

If you do not specify any options, all configured domains on all available EP11 cryptographic coprocessors are shown.

The **ep11info** tool is included in the EP11 package (RPM or DEB), starting with EP11 version 2.0.0.-n. For more information refer to the **ep11info** man page which is also included in the package and which is installed as part of the EP11 token.

Migrating master keys with the pkcsep11_migrate tool

There may be situations when the master key on (a domain of) a CEX*P adapter must be changed, for example, if company policies require periodic changes of all master keys. Simply changing the master keys using the TKE results in all secure keys stored in the EP11 token to become useless. Therefore all data encrypted by these keys are lost. To avoid this situation, you must accomplish a master key migration process, where activities on the TKE and on the Linux system must be interlocked.

Per default, the EP11 token stores all token key objects in the Linux file system in the /var/lib/opencryptoki/ep11tok/TOK_OBJ directory. For information about using multiple EP11 token instances, see "Adding EP11 tokens to openCryptoki" on page 39.

All secret and private keys are secure keys, that means they are enciphered (wrapped) with the master key (*MK*) of the CEX*P adapter domain. Therefore, the master key is often also referred to as wrapping key. If master keys are changed in a domain of a CEX*P adapter, all key objects for secure keys in the EP11 token object repository become invalid. Therefore, all key objects for secure keys must be re-enciphered with the new *MK*. In order to re-encipher secure keys that are stored as EP11 key objects in the EP11 token object repository, openCryptoki provides the master key migration tool **pkcsep11_migrate**.

How to access the master key migration tool

The *pkcsep11_migrate* key migration utility is part of openCryptoki version 3.1, which includes the EP11 support.

Prerequisites for the master key migration process

The master key migration process for the EP11 token requires a TKE version 7.3 environment. How to set up this environment is described in "Setting up the TKE environment" on page 19.

To use the *pkcsep11_migrate* migration tool, the EP11 crypto stack including openCryptoki must be installed and configured. For information on how to set up this environment, refer to Chapter 3, “Building the EP11 crypto stack,” on page 9.

The master key migration process

Prerequisite for re-encipherment: The EP11 token may be configured to use more than one adapter/ domain pair to perform its cryptographic operations. This is defined in the EP11 token configuration file. If the EP11 token is configured to use more than one adapter/domain pair, then all adapter/domain pairs must be configured to each have the same set of master keys. Therefore, if a master key on one of these adapter/domain pairs is changed, it must be changed on all those other adapter/domain pairs, too.

To migrate master keys on the set of adapter/domain pairs used by an EP11 token, you must perform the following steps:

1. On the TKE workstation (TKE), submit and commit the same new master key on all CEX*P adapter/ domain combinations used by the EP11 token.
2. On Linux, stop all processes that are currently using openCryptoki with the EP11 token.
3. On Linux, back up the token object repository of the EP11 token. For example, you can use the following commands:

```
cd /var/lib/opencryptoki/ep11
tar -cvzf ~/ep11TOK_OBJ_backup.tgz TOK_OBJ
```

4. On Linux, migrate the keys of the EP11 token object repository with the *pkcsep11_migrate* migration tool (see the invocation information provided at the end of these process steps). The *pkcsep11_migrate* tool must only be called once for one of the adapter/domain pairs that the EP11 token uses. If a failure occurs, restore the backed-up token repository and try this step again.

   Attention: Do not continue with step “5” on page 64 unless step “4” on page 64 was successful. Otherwise you will lose your encrypted data.

5. On the TKE, activate the new master keys on all EP11 adapter/domain combinations that the EP11 token uses.
6. On Linux, restart the applications that used openCryptoki with the EP11 token.

In step “1” on page 64 of the master key migration process, the new master key must be submitted and committed via the TKE interface. That means the *new EP11 master key* must be in the state `Full Committed`. The current MK is in the state `Valid`. Now both (current and new) *EP11 master keys* are available and accessible. The utility can now decrypt all relevant key objects within the token and re-encrypt all these key objects with the new master key.

Note: All the decrypt and encrypt operations are done inside the EP11 coprocessor card, that means that at no time clear key values are visible within memory.

Invocation:

```
pkcsep11_migrate -slot <number> -adapter <number> -domain <number>
```

The following parameters are mandatory:

-slot
- slot number for the EP11 token

-adapter
- the card ID; can be retrieved form the card ID in the *sysfs* (to be retrieved from `/sys/devices/ap/cardxx`, or with **`lszcrypt`**.

-domain
- the decimal card domain number (to be retrieved from `/sys/bus/ap/ap_domain` or with **`lszcrypt -b`**)

All token objects representing secret or private keys that are found for the EP11 token, are re-encrypted.

Note: The adapter and domain numbers can be specified in decimal, octal (with suffix 0), or hexadecimal (with suffix 0x) notation. The **lszcrypt** utility displays these fields in hexadecimal values.

Usage: You are prompted for your user PIN.

Examples:

```
pkcsep11_migrate  -slot 2 -adapter 8 -domain 48
pkcsep11_migrate  -slot 0x2 -adapter 010 -domain 0x30
```

Both invocations migrate the master key for the cryptographic coprocessor 8 (octal 010) and domain 48 (hex 0x30) used by the EP11 token from slot 2.

Note: The program stops if the re-encryption of a token object fails. In this case, restore the back-up.

After this utility re-enciphered all key objects, the new master key must be activated. This activation must be done by using the TKE interface command **Set, immediate**. Finally, the new master key becomes the current master key and the previous master key must be deleted.

Note: This tool is embedded in the users `sbin` path and therefore callable from everywhere.

To prevent token object generation during re-encryption, openCryptoki with the EP11 token must not be running during re-encryption. It is recommended to make a back-up of the EP11 token object directory (`/usr/local/var/lib/opencryptoki/ep11tok/TOK_OBJ`).

Managing EP11 sessions with the pkcsep11_session tool

An EP11 session is a state on the EP11 cryptographic coprocessor and must not be confused with a PKCS #11 session. An EP11 session is generated by the *strict session mode* or the *VHSM mode*. They are implicitly stored and deleted by openCryptoki if the according modes are set. So under normal circumstances, you need not care about the management of these EP11 sessions. But in some cases, for example, when programs crash or when programs do not close their sessions or do not call `C_Finalize` before exiting, some explicit EP11 session management may be required.

The *pkcsep11_session* tool allows to delete an EP11 session from the EP11 cryptographic coprocessors left over by programs that did not terminate normally. An EP11 cryptographic coprocessor supports only a certain number of EP11 sessions at a time. Because of this, it is important to delete any EP11 session, in particular when the program for which it was logged in, terminated unexpectedly. The *pkcsep11_session* tool is also used to set the card-PIN required for the *VHSM mode*.

pkcsep11_session usage examples

- Show all left over sessions:

  ```
  pkcsep11_session show
  ```

  A sample output for two left-over EP11 sessions could look as shown:

  ```
  # pkcsep11_session show -slot 4
  Using slot #4...

  Enter the USER PIN:
  List of EP11 sessions:

  30D5457762D8DDC158B558FCCC79FAB6:
          Pid:    48196
          Date:   2018/ 7/12
  30D5457762D8DDC158B558FCCC79FAB6:
          Pid:    48196
          Date:   2018/ 7/12

  2 EP11-Sessions displayed
  ```

  Note that only the first 16 bytes of the EP11 sessions ID are stored in the session object and therefore, the session IDs are displayed only partially. Otherwise, a user would be able to re-login on an EP11 adapter and re-use keys generated with this EP11 session, when the full EP11 session ID would be visible to the outside. Thus there may be identical session IDs when the *strict session mode* and the *virtual HSM (VHSM) mode* are combined for a session, as shown in the example.

- Show all left over EP11 sessions that belong to a specific process id (pid):

  ```
  pkcsep11_session show -pid 1234
  ```

- Show all left over EP11 sessions that have been created before a specific date:

  ```
  pkcsep11_session show -date 2018/06/29
  ```

- Logout all left over EP11 sessions:

  ```
  pkcsep11_session logout
  ```

- Logout all left over EP11 sessions that belong to a specific process id (pid):

  ```
  pkcsep11_session logout -pid 1234
  ```

- Logout all left over EP11 sessions that have been created before a specific date:

  ```
  pkcsep11_session logout -date 2018/07/27
  ```

- Logout all left over EP11 sessions even when the logout does not succeed on all adapters:

  ```
  pkcsep11_session logout -force
  ```

- Set a card-PIN:

  ```
  pkcsep11_session cardpin
  ```

  The card-PIN must be between 8 and 16 characters in length.

The *pkcsep11_session* tool provides its own man page that is installed as part of the EP11 package.

Migrating to FIPS compliance using the pkcstok_migrate tool

Use the **pkcstok_migrate** tool to transform an EP11 token, a CCA token, an ICA token, or a Soft Token into a data format that was generated by FIPS compliant operations. You can use this tool to migrate tokens created with all versions of openCryptoki, because also for version 3.12 or later, the old non-compliant format is the default. Being FIPS compliant, the token data is stored in a format that is better protected against attacks than the previously used data format.

For further information, read the **pkcstok_migrate** man page.

Parameters

```
# pkcstok_migrate -h

Help:         pkcstok_migrate -h
-h, --help    Show this help

Options:

-s, --slotid SLOTID            PKCS slot number (required)
-d, --datastore DATASTORE      token datastore location (required)
-c, --confdir CONFDIR          location of opencryptoki.conf (required)
-u, --userpin USERPIN          token user pin (prompted if not specified)
-p, --sopin SOPIN              token SO pin (prompted if not specified)
-v, --verbose LEVEL            set verbose level (optional):
none (default), error, warn, info, devel, debug
```

Functionality

The utility:

- directly accesses the token objects via file operations;
- assumes that no other action is currently running. It checks if the slot manager **pkcsslotd** is running and asks the user to end it if yes.

Before making any changes to the repository, a temporary copy is created. Migration takes place on this copy. The copied folder is suffixed with `_PKCSTOK_MIGRATE_TMP`. If the migration fails, the old repository is still available.

Running a migration again, would remove any remaining backups from previous runs, create a new backup, and then do the migration.

- After successfully migrating all token objects, the original repository folder is renamed by appending the suffix _BAK, and the new repository folder gets the name of the original one.
- Also, the `opencryptoki.conf` file is updated by inserting (or updating) the **tokversion** parameter in the token's slot configuration. The old configuration file is still available with the same suffix _BAK.

This makes the new repository immediately usable after restarting the **pkcsslotd** daemon, but also allows the user to switch back manually to the old token format.

Chapter 8. Programming examples for openCryptoki

The provided program segments in C illustrate some openCryptoki sample APIs to be used for EP11.

The contained openCryptoki code samples provide an insight into how to deal with the openCryptoki API's. After describing some basic functions such as initialization, session and log-in handling, the samples provide an introduction about how to create key objects and process symmetric encryption/ decryption (AES). The last section shows RSA key generation with RSA encrypt and decrypt operations.

To develop an application that uses the openCryptoki library, you need to access the library. You achieve the loading of shared objects by using dynamic library calls (dlopen) as described in the sample provided in "Base procedures" on page 70.

At compile time, you need to specify the openCryptoki library:

```
gcc test_ock.c -g -O0 -o test_ock -lopencryptoki -ldl
-I /usr/include/opencryptoki/
```

The exact location of the include files depends on your Linux distribution.

The following sample categories are provided:

- Base procedures
- Session and log-in
- Object handling
- Cryptographic operations

Base procedures

View some openCryptoki code samples for base procedures, such as a main program, an initialization procedure, and finalize information.

Main program

```
/* Example program to test opencryptoki
 * build: gcc test_ock.c -g -O0 -o test_ock -lopencryptoki -ldl
                         -I /root/opencryptoki/usr/include/pkcs11/
 * execute: ./test_ock -c <slot> -p <PIN> */
#include <stdlib.h>
#include <errno.h>
#include <stdio.h>
#include <dlfcn.h>
#include <pkcs11types.h>
#include <string.h>
#include <unistd.h>
#define OCKSHAREDLIB "libopencryptoki.so"

void *lib_ock;
char *pin = NULL;
int count, arg;
CK_SLOT_ID  slotID = 0;
CK_ULONG rsaKeyLen = 2048, cipherTextLen = 0, clearTextLen = 0;
CK_BYTE *pCipherText = NULL, *pClearText = NULL;
CK_BYTE *pRSACipher = NULL, *pRSAClear = NULL;
CK_FLAGS rw_sessionFlags = CKF_RW_SESSION | CKF_SERIAL_SESSION;
CK_SESSION_HANDLE hSession;
CK_BYTE keyValue[] = {0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,
                      0xCA,0xFE,0xBE,0xEF,0xCA,0xFE,0xBE,0xEF};
CK_BYTE msg[] = "The quick brown fox jumps over the lazy dog";
CK_ULONG msgLen = sizeof(msg);
CK_OBJECT_HANDLE hPublicKey, hPrivateKey;

/*** <insert helper functions (provided below) here> ***/
/*** usage / help ***/
void usage(void)
{
  printf("Usage:\n");
  printf(" -s <slot number> \n");
  printf(" -p <user PIN>\n");
  printf("\n");
  exit (8);  }

int main(int argc, char *argv[]) {
   while ((arg = getopt (argc, argv, "s:p:")) != -1) {
    switch (arg) {
    case 's':   slotID = atoi(optarg);
             break;
    case 'p':   pin = malloc(strlen(optarg));
                strcpy(pin,optarg);
             break;
    default:    printf("wrong option %c", arg);
             usage();
    }  }

  if ((!pin) || (!slotID)) {
    printf("Incorrect parameter given!\n");
    usage();
    exit (8);   }

  init();
  openSession(slotID, rw_sessionFlags, &hSession);
  loginSession(CKU_USER, pin, 8, hSession);
  createKeyObject(hSession, (CK_BYTE_PTR)&keyValue, sizeof(keyValue));
  AESencrypt(hSession, (CK_BYTE_PTR)&msg, msgLen, &pCipherText, &cipherTextLen);
  AESdecrypt(hSession, pCipherText, cipherTextLen, &pClearText, &clearTextLen);
  generateRSAKeyPair(hSession, rsaKeyLen, &hPublicKey, &hPrivateKey);
  RSAencrypt(hSession, hPublicKey, (CK_BYTE_PTR)&msg, msgLen, &pRSACipher, &rsaKeyLen);
  RSAdecrypt(hSession, hPrivateKey, pRSACipher, rsaKeyLen, &pRSAClear, &rsaKeyLen);
  logoutSession(hSession); closeSession(hSession);
  finalize();
  return 0;
}
```

C_Initialize

```
/*
 * initialize
 */
CK_RV init(void){
  CK_RV rc;
  lib_ock = dlopen(OCKSHAREDLIB, RTLD_GLOBAL | RTLD_NOW);
  if (!lib_ock) {
    printf("Error loading shared lib '%s' [%s]", OCKSHAREDLIB, dlerror());
    return 1;
  }
  rc = C_Initialize(NULL);
  if (rc != CKR_OK) {
     printf("Error initializing the opencryptoki library: 0x%X\n", rc);
  }
  return CKR_OK;
}
```

C_Finalize

```
/*
 * finalize
 */
CK_RV finalize(void) {
  CK_RV rc;
  rc = C_Finalize(NULL);
  if (rc != CKR_OK) {
         printf("Error during finalize: %x\n", rc);
  }
  if (pCipherText) free(pCipherText);
  if (pClearText)  free(pClearText);
  if (pRSACipher)  free(pRSACipher);
  if (pRSAClear)   free(pRSAClear);
  return rc;
}
```

Session and log-in procedures

When you use your sample code with a static linked library you can access the APIs directly. View some openCryptoki code samples for opening and closing sessions and for log-in.

C_OpenSession:

```
/*
 * opensession
 */

CK_RV openSession(CK_SLOT_ID slotID, CK_FLAGS sFlags,
                                     CK_SESSION_HANDLE_PTR phSession) {
 CK_RV rc;
  rc = C_OpenSession(slotID, sFlags, NULL, NULL, phSession);
  if (rc != CKR_OK) {
     printf("Error opening session: %x\n", rc);
     return rc;
  }
  printf("Open session successful.\n");
  return CKR_OK;
}
```

C_CloseSession:

```
/*
 * closesession
 */
CK_RV closeSession(CK_SESSION_HANDLE hSession) {
  CK_RV  rc;
  rc  = C_CloseSession(hSession);
  if (rc != CKR_OK) {
    printf("Error closing session: 0x%X\n", rc);
    return rc;
  }
  printf("Close session successful.\n");
  return CKR_OK;
}
```

C_Login:

```
/*
 * login
 */
CK_RV loginSession(CK_USER_TYPE userType, CK_CHAR_PTR pPin,
           CK_ULONG ulPinLen, CK_SESSION_HANDLE hSession) {
  CK_RV rc;
  rc = C_Login(hSession, userType, pPin, ulPinLen);
  if (rc != CKR_OK) {
    printf("Error login session: %x\n", rc);
    return rc;
  }
  printf("Login session successful.\n");
  return CKR_OK;
}
```

C_Logout:

```
/*
 * logout
 */
CK_RV logoutSession(CK_SESSION_HANDLE hSession) {
  CK_RV rc;
  rc = C_Logout(hSession);
  if (rc != CKR_OK) {
    printf("Error logout session: %x\n", rc);
    return rc;
  }
  printf("Logout session successful.\n");
  return CKR_OK;
}
```

Object handling procedures

When you use your sample code with a static linked library you can access the APIs directly. View some openCryptoki code samples for procedures dealing with object handling.

C_CreateKeyObject:

```
/*
 * createKeyObject
 */
CK_RV createKeyObject(CK_SESSION_HANDLE hSession, CK_BYTE_PTR key, CK_ULONG keyLength) {
  CK_RV rc;

  CK_OBJECT_HANDLE hKey;
  CK_BBOOL true = TRUE;
  CK_BBOOL false = FALSE;
  CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
  CK_KEY_TYPE keyType = CKK_AES;
  CK_ATTRIBUTE keyTempl[] = {
    {CKA_CLASS, &keyClass, sizeof(keyClass)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_ENCRYPT, &true, sizeof(true)},
    {CKA_DECRYPT, &true, sizeof(true)},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_VERIFY, &true, sizeof(true)},
    {CKA_TOKEN, &true, sizeof(true)},         /* token object  */
    {CKA_PRIVATE, &false, sizeof(false)},    /* public object */
    {CKA_VALUE, keyValue, keyLength},    /* AES key       */
    {CKA_LABEL, "My_AES_Key", sizeof("My_AES_Key")}
  };
  rc = C_CreateObject(hSession, keyTempl, sizeof (keyTempl)/sizeof (CK_ATTRIBUTE), &hKey);
  if (rc != CKR_OK) {
    printf("Error creating key object: 0x%X\n", rc); return rc;
  }
  printf("AES Key object creation successful.\n");
  return CKR_OK;
}
```

C_FindObjects:

```
/*
 * findObjects
 */
CK_RV getKey(CK_CHAR_PTR label, int labelLen, CK_OBJECT_HANDLE_PTR hObject,
         CK_SESSION_HANDLE hSession) {
  CK_RV rc;
  CK_ULONG ulMaxObjectCount = 1;
  CK_ULONG ulObjectCount;
  CK_ATTRIBUTE objectMask[] = { {CKA_LABEL, label, labelLen} };
  rc = C_FindObjectsInit(hSession, objectMask, 1);
  if (rc != CKR_OK) {
    printf("Error FindObjectsInit: 0x%X\n", rc); return rc;
  }
  rc = C_FindObjects(hSession, hObject, ulMaxObjectCount, &ulObjectCount);
  if (rc != CKR_OK) {
    printf("Error FindObjects: 0x%X\n", rc); return rc;
  }
  rc = C_FindObjectsFinal(hSession);
  if (rc != CKR_OK) {
    printf("Error FindObjectsFinal: 0x%X\n", rc); return rc;
  }
  return CKR_OK;
}
```

Cryptographic operations

When you use your sample code with a static linked library you can access the APIs directly. View some openCryptoki code samples for procedures that perform cryptographic operations.

C_Encrypt (AES):

```
/*
 * AES encrypt
 */
CK_RV AESencrypt(CK_SESSION_HANDLE hSession,
         CK_BYTE_PTR pClearData,  CK_ULONG ulClearDataLen,
         CK_BYTE **pEncryptedData, CK_ULONG_PTR pulEncryptedDataLen) {
  CK_RV rc;
  CK_MECHANISM myMechanism = {CKM_AES_CBC_PAD, "01020304050607081122334455667788", 16};
  CK_MECHANISM_PTR pMechanism = &myMechanism;
  CK_OBJECT_HANDLE hKey;
  getKey("My_AES_Key", sizeof("My_AES_Key"), &hKey, hSession);
  rc = C_EncryptInit(hSession, pMechanism, hKey);
  if (rc != CKR_OK) {
    printf("Error initializing encryption: 0x%X\n", rc);
    return rc;
  }
  rc = C_Encrypt(hSession, pClearData, ulClearDataLen,
         NULL, pulEncryptedDataLen);
  if (rc != CKR_OK) {
    printf("Error during encryption (get length): %x\n", rc);
    return rc;
  }
  *pEncryptedData = (CK_BYTE *)malloc(*pulEncryptedDataLen * sizeof(CK_BYTE));

  rc = C_Encrypt(hSession, pClearData, ulClearDataLen,
         *pEncryptedData, pulEncryptedDataLen);
  if (rc != CKR_OK) {
    printf("Error during encryption: %x\n", rc);
    return rc;
  }
  printf("Encrypted data: ");
  CK_BYTE_PTR tmp = *pEncryptedData;
  for (count = 0; count < *pulEncryptedDataLen; count++, tmp++) {
    printf("%X", *tmp);
  }
  printf("\n");

  return CKR_OK;
}
```

C_Decrypt (AES):

```
/*
 * AES decrypt
 */
CK_RV AESdecrypt(CK_SESSION_HANDLE hSession,
         CK_BYTE_PTR pEncryptedData, CK_ULONG ulEncryptedDataLen,
         CK_BYTE **pClearData, CK_ULONG_PTR pulClearDataLen) {
  CK_RV rc;
  CK_MECHANISM myMechanism = {CKM_AES_CBC_PAD, "01020304050607081122334455667788", 16};
  CK_MECHANISM_PTR pMechanism = &myMechanism;
  CK_OBJECT_HANDLE hKey;
  getKey("My_AES_Key", sizeof("My_AES_Key"), &hKey, hSession);
  rc = C_DecryptInit(hSession, pMechanism, hKey);
  if (rc != CKR_OK) {
    printf("Error initializing decryption: 0x%X\n", rc);
    return rc;
  }
  rc = C_Decrypt(hSession, pEncryptedData, ulEncryptedDataLen, NULL, pulClearDataLen);
  if (rc != CKR_OK) {
    printf("Error during decryption (get length): %x\n", rc);
    return rc;
  }
  *pClearData = malloc(*pulClearDataLen * sizeof(CK_BYTE));
  rc = C_Decrypt(hSession, pEncryptedData, ulEncryptedDataLen, *pClearData,
       pulClearDataLen);
  if (rc != CKR_OK) {
    printf("Error during decryption: %x\n", rc);
    return rc;
  }
  printf("Decrypted data: ");
  CK_BYTE_PTR tmp = *pClearData;
  for (count = 0; count < *pulClearDataLen; count++, tmp++) {
    printf("%c", *tmp);
  }
  printf("\n");
  return CKR_OK;
}
```

C_GenerateKeyPair (RSA):

```
/*
 * RSA key generate
 */
CK_RV generateRSAKeyPair(CK_SESSION_HANDLE hSession, CK_ULONG keySize,
             CK_OBJECT_HANDLE_PTR phPublicKey, CK_OBJECT_HANDLE_PTR phPrivateKey ) {
  CK_RV rc;
  CK_BBOOL true = TRUE;
  CK_BBOOL false = FALSE;
  CK_OBJECT_CLASS keyClassPub = CKO_PUBLIC_KEY;
  CK_OBJECT_CLASS keyClassPriv = CKO_PRIVATE_KEY;
  CK_KEY_TYPE keyTypeRSA = CKK_RSA;
  CK_ULONG modulusBits = keySize;
  CK_BYTE_PTR pModulus = malloc(sizeof(CK_BYTE)*modulusBits/8);
  CK_BYTE publicExponent[] = {1, 0, 1};
  CK_MECHANISM rsaKeyGenMech = {CKM_RSA_PKCS_KEY_PAIR_GEN, NULL_PTR, 0};
  CK_ATTRIBUTE pubKeyTempl[] = {
    {CKA_CLASS, &keyClassPub, sizeof(keyClassPub)},
    {CKA_KEY_TYPE, &keyTypeRSA, sizeof(keyTypeRSA)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_PRIVATE, &true, sizeof(true)},
    {CKA_ENCRYPT, &true, sizeof(true)},
    {CKA_VERIFY, &true, sizeof(true)},
    {CKA_WRAP, &true, sizeof(true)},
    {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},
    {CKA_PUBLIC_EXPONENT, publicExponent, sizeof(publicExponent)},
    {CKA_LABEL, "My_Private_Token_RSA1024_PubKey",
    sizeof("My_Private_Token_RSA1024_PubKey")},
    {CKA_MODIFIABLE, &true, sizeof(true)},
  };
  CK_ATTRIBUTE privKeyTempl[] = {
    {CKA_CLASS, &keyClassPriv, sizeof(keyClassPriv)},
    {CKA_KEY_TYPE, &keyTypeRSA, sizeof(keyTypeRSA)},
    {CKA_EXTRACTABLE, &true, sizeof(true)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_PRIVATE, &true, sizeof(true)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_DECRYPT, &true, sizeof(true)},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_UNWRAP, &true, sizeof(true)},
    {CKA_LABEL, "My_Private_Token_RSA1024_PrivKey",
    sizeof("My_Private_Token_RSA1024_PrivKey")},
    {CKA_MODIFIABLE, &true, sizeof(true)},
  };
  rc = C_GenerateKeyPair(hSession, &rsaKeyGenMech ,
             &pubKeyTempl, sizeof(pubKeyTempl)/sizeof (CK_ATTRIBUTE),
             &privKeyTempl, sizeof(privKeyTempl)/sizeof (CK_ATTRIBUTE),
             phPublicKey, phPrivateKey);
  if (rc != CKR_OK) {
    printf("Error generating RSA keys: %x\n", rc);
    return rc;
  }
  printf("RSA Key generation successful.\n");
  return CKR_OK;
}
```

C_Encrypt (RSA):

```
/*
 * RSA encrypt
 */
CK_RV RSAencrypt(CK_SESSION_HANDLE hSession, CK_OBJECT_HANDLE hKey,
         CK_BYTE_PTR pClearData, CK_ULONG ulClearDataLen,
         CK_BYTE **pEncryptedData, CK_ULONG_PTR pulEncryptedDataLen) {
  CK_RV rc;
  CK_MECHANISM rsaMechanism = {CKM_RSA_PKCS, NULL_PTR, 0};
  rc = C_EncryptInit(hSession, rsaMechanism, hKey);
  if (rc != CKR_OK) {
    printf("Error initializing RSA encryption: %x\n", rc);
    return rc;
  }
  rc = C_Encrypt(hSession, pClearData, ulClearDataLen,
         NULL, pulEncryptedDataLen);
  if (rc != CKR_OK) {
    printf("Error during RSA encryption: %x\n", rc);
    return rc;
  }

  *pEncryptedData = (CK_BYTE *)malloc(rsaKeyLen * sizeof(CK_BYTE));
  rc = C_Encrypt(hSession, pClearData, ulClearDataLen,
         *pEncryptedData, pulEncryptedDataLen);
  if (rc != CKR_OK) {
    printf("Error during RSA encryption: %x\n", rc);
    return rc;
  }

  printf("Encrypted data: ");
  CK_BYTE_PTR tmp = *pEncryptedData;
  for (count = 0; count < *pulEncryptedDataLen; count++, tmp++) {
    printf("%X", *tmp);
  }
  printf("\n");
  return CKR_OK;
}
```

C_Decrypt (RSA):

```
/*
 * RSA decrypt
 */
CK_RV RSAdecrypt(CK_SESSION_HANDLE hSession, CK_OBJECT_HANDLE hKey,
         CK_BYTE_PTR pEncryptedData, CK_ULONG ulEncryptedDataLen,
         CK_BYTE **pClearData, CK_ULONG_PTR pulClearDataLen) {
  CK_RV rc;
  CK_MECHANISM rsaMechanism = {CKM_RSA_PKCS, NULL_PTR, 0};
  rc = C_DecryptInit(hSession, rsaMechanism, hKey);
  if (rc != CKR_OK) {
    printf("Error initializing RSA decryption: %x\n", rc);
    return rc;
  }
  rc = C_Decrypt(hSession, pEncryptedData, ulEncryptedDataLen,
         NULL, pulClearDataLen);
  if (rc != CKR_OK) {
    printf("Error during RSA decryption: %x\n", rc);
    return rc;
  }

  *pClearData = malloc(rsaKeyLen*sizeof(CK_BYTE));
  rc = C_Decrypt(hSession, pEncryptedData, ulEncryptedDataLen,
         *pClearData, pulClearDataLen);
  if (rc != CKR_OK) {
    printf("Error during RSA decryption: %x\n", rc);
    return rc;
  }
  printf("Decrypted data: ");
  CK_BYTE_PTR tmp = *pClearData;
  for (count = 0; count < *pulClearDataLen; count++, tmp++) {
    printf("%c", *tmp);
  }
  printf("\n");
  return CKR_OK;
}
```

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 and LinuxONE 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 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.

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.

The registered trademark Linux is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Glossary

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.

asymmetric cryptography
Synonym for public key cryptography..

Central Processor Assist for Cryptographic Function (CPACF)
Hardware that provides support for symmetric ciphers and secure hash algorithms (SHA) on every central processor. Hence the potential encryption/decryption throughput scales with the number of central processors in the system.

Chinese-Remainder Theorem (CRT)
A mathematical problem described by Sun Tsu Suan-Ching using the remainder from a division operation.

Cipher Block Chaining (CBC)
A method of reducing repetitive patterns in cipher-text by performing an exclusive-OR operation on each 8-byte block of data with the previously encrypted 8-byte block before it is encrypted.

Cipher block length
The length of a block that can be encrypted or decrypted by a symmetric cipher. Each symmetric cipher has a specific cipher block length.

clear key
Any type of encryption key not protected by encryption under another key.

CPACF instructions
Instruction set for the CPACF hardware.

Crypto Express card, Crypto Express adapter (CEX*S)
Beginning with CEX4S, the PCIe adapter on a CEX*S feature can be configured in three ways: Either as cryptographic accelerator (CEX*A), or as CCA coprocessor (CEX*C) for secure key encrypted transactions, or in EP11 coprocessor mode (CEX*P) for exploiting Enterprise PKCS #11 functionality.

A CEX*P adapter only supports secure key mode.

ECC
See *Elliptic curve cryptography*.

electronic code book mode (ECB mode)
A method of enciphering and deciphering data in address spaces or data spaces. Each 64-bit block of plain-text is separately enciphered and each block of the cipher-text is separately deciphered.

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. Due to the algebraic structure of elliptic curves over finite fields, ECC provides a similar amount of security to that of RSA algorithms, but with relatively shorter key sizes.

libica
Library for IBM Cryptographic Architecture.

master key (MK)
In computer security, the top-level key in a hierarchy of key-encrypting keys.

Mode of operation
A schema describing how to apply a symmetric cipher to encrypt or decrypt a message that is longer than the cipher block length. The goal of most modes of operation is to keep the security level of the cipher by avoiding the situation where blocks that occur more than once will always be translated to the same value. Some modes of operations allow handling messages of arbitrary lengths.

modulus-exponent (Mod-Expo)

A type of exponentiation performed using a modulus.

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.

Rivest-Shamir-Adleman (RSA)

An algorithm used in public key cryptography. These are the surnames of the three researchers responsible for creating this asymmetric or public/private key algorithm.

Secure Hash Algorithm (SHA)

An encryption method in which data is encrypted in a way that is mathematically impossible to reverse. Different data can possibly produce the same hash value, but there is no way to use the hash value to determine the original data.

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.

symmetric cryptogrphy

An encryption method that uses the same key for encryption and decryption. Keys of symmetric ciphers are private keys.

zcrypt device driver

Kernel device driver to access Crypto Express adapters. Formerly, a monolithic module called **`z90crypt`**. Today, it consists of multiple modules that are implicitly loaded when loading the **ap** main module of the device driver.

Index

SC34-2713-02