Secure Cryptography and Certificate Services Toolkit

Developer's Guide

Table of Contents

LIST OF FIGURES

List of Tables

Chapter 1. Introduction

Recently cryptography has come into widespread use in meeting multiple security needs, such as confidentiality, integrity, authentication and non-repudiation. In order to address these requirements in the emerging Internet, Intranet, and Extranet application domains, the Secure Cryptography and Certificate Services (SCCS) Toolkit was developed. The SCCS Toolkit is a comprehensive set of layered security services suitable for use in operating systems, such as IBM AIX, MVS, OS/400, and Windows NT/95, Solaris, HP-UX, as middleware in embedded applications, or provided as a component of cryptographic security toolkits. The SCCS Toolkit focuses on security in peer-to-peer, store-and-forward, and archival applications. It is designed to be compliant with industry standards such as OpenGroup, and is applicable to a broad range of hardware and operating system platforms. SCCS is intended to include full lifecycle key management and portable credentials. The definition of such a set of layered security services and an open architecture protects the investment made in implementation of security applications by facilitating the reuse of core components of the architecture for different products.

The security services available in the SCCS Toolkit are defined by the categories of service provider modules that the architecture accommodates. These service providers are:

- Cryptographic Service Providers
- Key Recovery Service Providers
- Trust Policy Library Service Providers
- Certificate Library Service Providers
- Data Storage Library Service Providers

The central component of this architecture is the SCCS Framework, which is a layer of middleware that lies between application code and the service provider modules. The SCCS Framework is based on Intel's Common Security Services Manager (CSSM); however the existing interfaces of CSSM have been enhanced to include key recovery features. Unlike basic security features such as cryptographic functions, certificates, and trust policy, key recovery is a relatively new field and is the focus of innovations related to the SCCS Toolkit.

1.1 SCCS Toolkit Architecture

The Secure Cryptography and Certificate Services (SCCS) Toolkit Architecture consists of a set of layered security services and associated programming interfaces designed to furnish an integrated set of information and communication security capabilities. Each layer builds on the more fundamental services of the layer directly below it.

These layers start with fundamental components such as cryptographic algorithms, random numbers, and unique identification information in the lower layers, and build up to digital certificates, key management and recovery mechanisms and secure transaction protocols in higher layers. The SCCS Architecture is intended to be the multi-platform security architecture that is both horizontally broad and vertically robust.

Figure 1 shows a simplified view of the layered architecture of an SCCS-based system. There are four major layers in the SCCS Toolkit Architecture; the application domain layer, the protocol handler layer, the SCCS Framework, and the service provider layer.

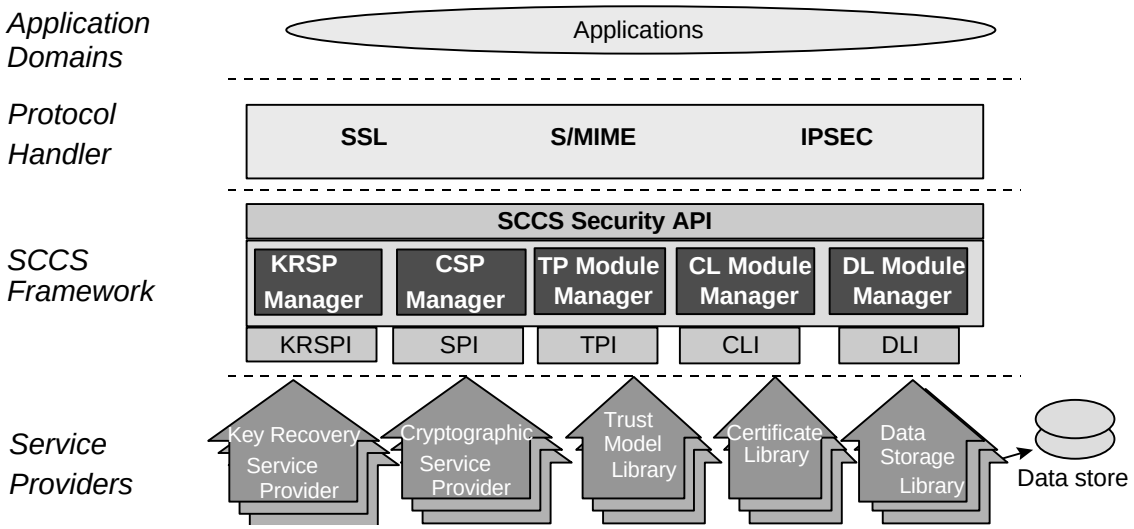

Figure 1. The Secure Cryptography and Certificate Services Toolkit Architecture

The Application Domain layer implements the application domain services, such as Secure Electronic Transactions (SET) and E-Wallet, E-mail services, or file archival services. The Protocol Handler layer is between the application domain layer and the SCCS Framework layer. It implements security protocols that are used by the application domain layer. Software at this layer may implement cryptographic protocol handlers such as SSL, IPSEC, S/MIME and EDI. The protocol handler layer also includes tools and utilities for installing, configuring, and maintaining the SCCS Framework and service provider modules.

The Secure Cryptography and Certificate Services (SCCS) Framework is the central component of this extensible architecture that provides mechanisms to dynamically manage service provider modules. The SCCS Framework defines a common security API that must be used to access services of service provider modules. Applications request security services through the SCCS security API or through protocol handlers implemented over the SCCS API. The service provider modules actually perform the requested security services. IBM provides a number of SP modules. Additional SP modules may be available from other independent software and hardware vendors. Applications may direct their requests to modules from specific vendors or to any module that performs the required services. Both the SCCS Framework and the service provider interfaces are discussed in detail in this document.

1.2 Audience

This document provides an overview of the Secure Cryptography and Certificate Services Toolkit for Independent Software Vendors (ISVs) who develop their own operating systems or other security products either as complete applications or as plug-ins to extensible platforms. This document is intended for use by:

- Advanced programmers
- Experienced software designers
- Security architects who work in high-end cryptography
- Sophisticated integrators familiar with numerous forms of network computing
- Vendors of customizable service providers (SPs) for cryptographic, trust, and database services

This audience understands the requirements for a ubiquitous security infrastructure upon which they can build security-aware application products.

1.3 System Requirements

The following software is required in order to develop applications using the SCCS Toolkit:

- A C/C++ compiler for developing the applications (e.g., IBM Visual Age, Microsoft Visual C++)

1.4 Calling Conventions

Applications that perform SCCS Toolkit API calls need to follow *cdecl* calling conventions (i.e. push parameters on the stack, in reverse order (right to left)). Specifically, the minimum requirement is that the application's callback functions which are passed to the SCCS framework (e.g. the memory functions passed into the CSSM_Init() API call) need to be declared as *cdecl* functions. However, setting the default calling convention for all compilations in your development environment to *cdecl* is strongly recommended. Please refer to the appropriate manual for your compiler.

1.5 Documentation Set

The Secure Cryptography and Certificate Services Toolkit documentation set consists of the following manuals. These manuals are provided in electronic format and can be viewed using the Adobe Acrobat Reader distributed with Secure Cryptography and Certificate Services Toolkit. Both the electronic manuals and the Adobe Acrobat Reader are located in the SCCS Toolkit `doc` subdirectory.

- *Secure Cryptography and Certificate Services Toolkit Developer's Guide*
  Document filename: sccs_dev.pdf
  This document presents an overview of the Secure Cryptography and Certificate Services (SCCS) Toolkit. It explains how to integrate SCCS into applications and contains a sample SCCS application.
- *Secure Cryptography and Certificate Services Toolkit Application Programming Interface Specification*
  Document filename: sccs_api.pdf
  This document defines the interface that applications developers employ to access security services provided by the SCCS Framework and service provider modules.

- *Secure Cryptography and Certificate Services Toolkit Service Provider Module Structure & Administration*
  Document filename: sccs_mod.pdf
  This document describes the features common to all SCCS service provider modules. It should be used in conjunction with the individual SCCS Service Provider Interface Specifications in order to build a service provider module.
- *Secure Cryptography and Certificate Services Toolkit Cryptographic Service Provider Interface Specification*
  Document filename: sccs_spi.pdf
  This document defines the interface to which cryptography service provider modules must conform in order to be accessible through SCCS.
- *Key Recovery Service Provider Interface Specification*
  Document filename: kr_spi.pdf
  This document defines the interface to which key recovery service provider modules must conform in order to be accessible through SCCS.
- *Secure Cryptography and Certificate Services Toolkit Trust Policy Interface Specification*
  Document filename: sccs_tp_spi.pdf
  This document defines the interface to which policy makers, such as Certificate Authorities (CAs), Certificate Issuers, and policy-making application developers, must conform in order to extend SCCS with model or application specific policies.
- *Secure Cryptography and Certificate Services Toolkit Certificate Library Interface Specification*
  Document filename: sccs_cl_spi.pdf
  This document defines the interface to which certificate library developers must conform to provide format-specific certificate manipulation services to numerous SCCS applications and trust policy modules.
- *Secure Cryptography and Certificate Services Toolkit Data Storage Library Interface Specification*
  Document filename: sccs_dl_spi.pdf
  This manual defines the interface to which library developers must conform to provide format-specific or format-independent persistent storage of certificates.

1.6 Other References

BSAFE*	*BSAFE Cryptography Toolkit*, RSA Data Security, Inc., Redwood City, CA
PKCS*	*The Public-Key Cryptography Standards*, RSA Laboratories, Redwood City, CA: RSA Data Security, Inc.
X.509	*CCITT. Recommendation X.509: The Directory – Authentication Framework*. 1988. CCITT stands for Comite Consultatif Internationale Telegraphique et Telphonique (International Telegraph and Telephone Consultative Committee)
Cryptography	*Applied Cryptography, Second Edition Protocols, Algorithms, and Source Code in C*, Bruce Schneier: John Wiley & Sons, Inc., 1996

Chapter 2. Secure Cryptography and Certificate Services Framework

The Secure Cryptography and Certificate Services (SCCS) Framework layer is the central component in the SCCS architecture; it integrates and manages all the security services. SCCS enables tight integration of individual services, while allowing those services to be provided by interoperable service provider modules. The SCCS Framework has a rich API to support the development of secure applications and system services, and a service provider interface (SPI) that supports service provider modules that implement building blocks for secure operations.

The primary function of the SCCS Framework layer is to maintain state regarding the connections between the application layer code and the service providers underneath. Additionally, the SCCS mediates all interactions between applications and the service provider modules, and implements and enforces the applicable key recovery policy. Finally, the SCCS Framework allows the seamless integration of the key recovery functions and the other security functions provided by independent service provider modules.

The SCCS Framework does not prescribe or implement any security services. Application-specific security services are defined and implemented by service provider modules and layered services. The SCCS Framework defines a common API for accessing the services provided by service provider modules. SCCS re-directs application API calls to the selected service provider module that will perform the request.

The SCCS API calls can be categorized as service operations or core services. Service operations are functions that invoke an SP module security operation, such as encrypting data, adding a certificate to a certificate revocation list, or verifying that a certificate is trusted/authorized to perform some action. SCCS Module Managers are responsible for carrying out service operations. Core services include functions that performs the following:

- Module management
- Memory management
- Security context management
- Integrity verification

This chapter discusses the SCCS Framework core services. The individual SCCS Module Managers are discussed in Chapter 3 through Chapter 7. For information on the IBM service provider modules and the SPI for the types of modules supported, see Chapter 8.

2.1 Module Management

The SCCS Framework defines a set of API calls that allow application developers to access and use service provider (SP) modules. These module management functions support the installation of service provider modules, the dynamic selection and loading of modules, and the querying of module features and status. System administration utilities use install and uninstall functions to maintain service provider modules on a local system.

2.1.1 Installing and Uninstalling Service Provider Modules

SCCS manages a registry that records the logical name of each SP module that is installed on the system, the information required to locate and dynamically initiate the SP, and some minimal meta-data describing the algorithms implemented by the SP.

When an SP is loaded, it must register its services with the SCCS Framework using CSSM_ModuleInstall before its services can be used by an application or another service provider module. The SP, or a proxy for the SP (such as its Adaptation Layer), registers a set of callback functions with the SCCS Framework. There is one callback function for each SCCS-defined SPI call. The SP may or may not implement all SPI calls defined by SCCS. Unimplemented functions must be registered as null. The SP may implement additional functions outside of the SCCS-defined SPI calls. The SP may register a single callback function, and instruct applications and modules developers (through documentation) to activate these functions through the message-based, SCCS *pass-through* function. There is one pass-through function defined in each SPI. For example, the pass-through function defined for the cryptographic SPI is CSP_PassThrough.

SP modules may also be uninstalled using the CSSM_ModuleUninstall function. This function removes the SP name and its associated attributes from the SCCS Framework's SP registry. Uninstall must be performed before a new version of the same SP module is installed in the SCCS Framework registry.

2.1.2 Listing Service Provider Modules and Services

Before attaching a service module, an application can query the SCCS Framework registry, using the CSSM_ListModules function, to obtain information on

- the modules installed on the system
- the capabilities (and functions) implemented by those modules
- the GUID associated with a given module

Applications use this information to dynamically select a module for use. A multi-service module has multiple capability descriptions associated with it,at least one per functional area supported by the module. Some areas (such as CSP and TP) may have multiple independent capability descriptions for a single functional area. There is one SCCS Framework registry entry for a multi-service module, which records all service types for the module. SCCS returns all information about a module's capabilities when queried by the application. Each set of capabilities includes a type identifier to distinguish CSPinfo from CLinfo, etc.

Applications can query about the SCCS Framework itself. One function, CSSM_GetInfo, returns version information about the running SCCS Framework. Another function, CSSM_Init, verifies whether the SCCS Framework version the application expects is compatible with the currently-running SCCS Framework version. The general function to query service provider module information also returns the module's version information.

2.1.3 Attaching and Detaching SP modules

Applications select the particular security services they will use by selectively attaching service provider modules. Each module has an assigned, Globally Unique ID (GUID), and a set of descriptive attributes to assist applications in selecting appropriate modules for their use. A module can implement a range of services across the SCCS APIs (e.g., cryptographic functions and data storage functions) or a module can restrict its services to a single SCCS category of service (e.g., certificate library services only). Modules that span service categories are called Multi-Service modules.

Applications use a module's GUID to specify the module to be attached. The attach function, CSSM_ModuleAttach, returns a handle representing a unique pairing between the caller and the attached module. This handle must be provided as in input parameter when requesting services from the attached module. SCCS uses the handle to match the caller with the appropriate service module.

The calling application uses the handle to obtain any and all types of services implemented by the attached module. Figure 2 shows how the handle for an attached PKCS#11 service provider is used to perform cryptographic operations and persistent storage of certificates. The single handle value can be used as the CSPHandle in cryptographic operations and as the DLHandle in data storage operations.

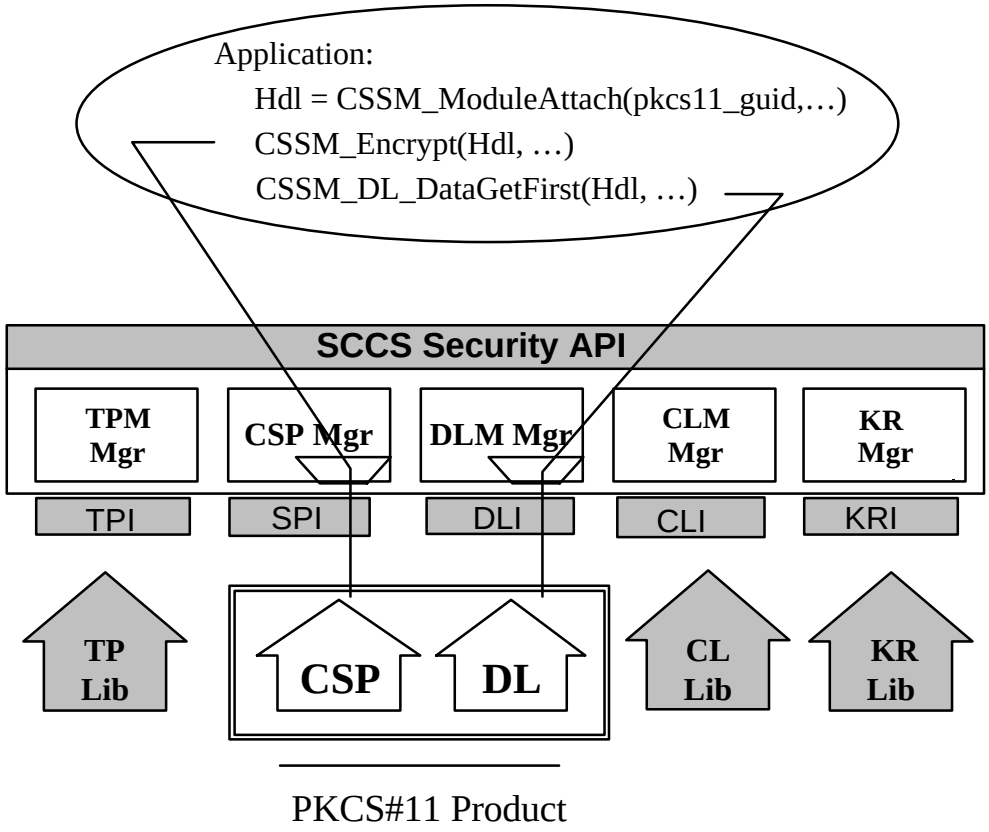

Figure 2. Application using cryptographic services and persistent storage services of a class 2, PKCS#11 device.

Multiple calls to attach are viewed as independent requests. Each attach request returns separate, independent handles that do not share execution state.

SP modules may be detached using the CSSM_ModuleDetach function. However, an application should not invoke this operation unless all requests to the target SP have been completed.

2.1.4 Managing Calls Between Service Provider Modules

Applications directly or indirectly select the modules that will be used to provide security services to the application. Service provider modules may (and often will) invoke other service provider modules to

perform necessary operations. SCCS forwards all calls uniformly regardless of their origin. Figure 3 illustrates the process by which the SCCS Framework manages calls between modules.

In Figure 3, the application invokes `func1` in the cryptographic module identified by the handle `CSP1`. SCCS forwards the function call to `func1` in the `CSP1` module. The application also invokes `func7` in the trust policy module identified by the handle `TP2`. Again SCCS forwards the function call to `func7` in the `TP2` module. The implementation of `func7` in the `TP2` module uses functions implemented by a certificate library module. The `TP2` module must invoke the certificate library functions through the SCCS Framework. To accomplish this, the `TP2` module attaches the certificate library module, obtaining the handle `CL1`, and invokes `func13` in the certificate library identified by the handle `CL1`. SCCS forwards the function call to `func13` in the `CL1` module.

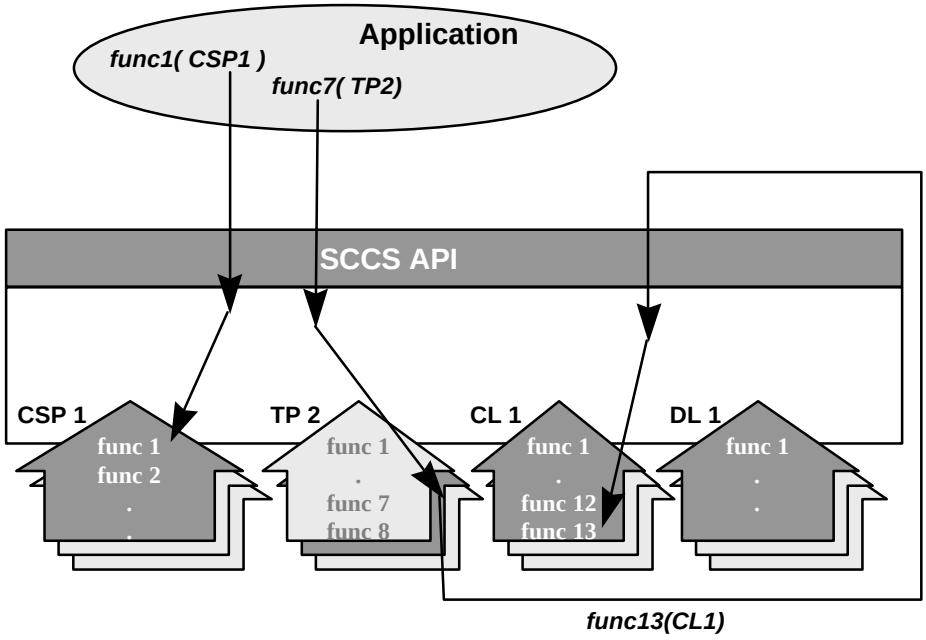

Figure 3. The SCCS Framework Directs Calls to Selected Service Provider Modules.

Modules must be loaded before they can receive function calls from the SCCS Framework. An error condition occurs if the invoked function is not implemented by the selected module.

2.2 Memory Management

The SCCS memory management functions are a class of routines for reclaiming memory allocated by SCCS on behalf of an application from the SCCS memory heap. When SCCS allocates objects from its own heap and returns them to an application, the application must inform SCCS when it no longer requires the use of that object. Applications use specific APIs to free SCCS-allocated memory. When an application invokes an API free function SCCS can choose to retain or free the indicated object depending on other conditions known only to SCCS. In this way SCCS and applications work together to manage these objects in the SCCS memory heap.

2.3 Security Context Management

Security-context management provides secured run-time caching of user-specific state information and secrets. Multi-step cryptographic operations, such as staged hashing, require multiple calls to a CSP and the intermediate operation states must be managed. These intermediate states are stored in run-time data structures known as security contexts. The SCCS API provides a number of context functions that applications can use to create, initialize, and cache security contexts. Security contexts provide mechanisms that:

- Allow an application to use multiple CSPs concurrently
- Allow an application to concurrently use different parameters for a single CSP algorithm
- Support layered implementations in their transparent use of multiple CSPs or different algorithm parameters for the same CSP
- Enable development of re-entrant CSPs, layered services, and applications

Applications retain handles to each security context used during execution. The context handle is a required input parameter to many security service functions. Most applications instantiate and use multiple security contexts. Only one context may be passed to a function, but the application is free to switch among contexts at will, or as required (even per function call).

An application may create multiple contexts directly or indirectly. Indirect creation may occur when invoking layered services, system utilities, key recovery service providers, trust policy modules, certificate library modules, or data storage library modules that create and use their own appropriate security context as part of the service they provide to the invoking application. Figure 4 shows an example of a hidden security context. An application creates a context specifying the use of `sec_context1`. The application invokes `func1` in the certificate library using `sec_context1` as a parameter. The certificate library performs two calls to the cryptographic service provider. For the call to `func5`, the hidden security context is used. For the call to `func6`, the application's security context is passed as a parameter to the CSP.

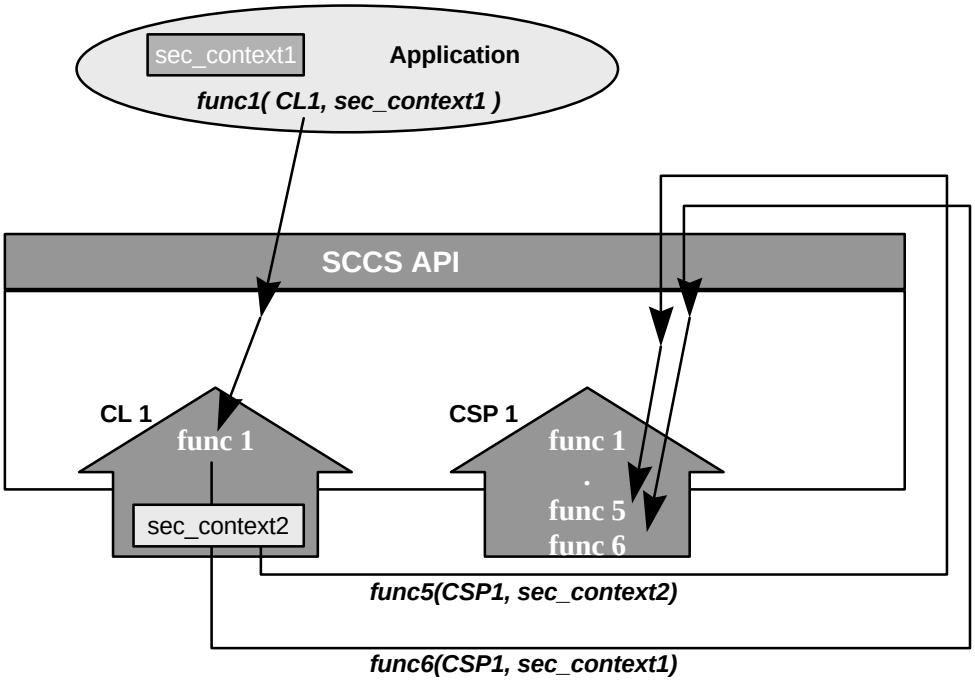

Figure 4. Indirect Creation of a Security Context.

These transparent contexts do not concern the application developer, as they are managed entirely by the layered service or service provider module that created them. Each process or thread that creates a security context is responsible for explicitly terminating that context.

SCCS provides a number of API functions to create security contexts. The function used and type of context created depends on the cryptographic operation being performed. For example, the CSSM_CSP_CreateSymmetricContext is used in cryptographic operations involving a symmetric key; the CSSM_CSP_CreateAsymmetricContext is used operations involving an asymmetric key. The CSSM_DeleteContext function is paired up with the create context functions. These functions are designed to be used by applications and force notify events to be sent to a service provider module. In contrast, the CSSM_GetContext and CSSM_FreeContext functions are designed to be used by service provider modules since they do not generate events.

2.4 Integrity Verification

As a security framework, SCCS provides each application with additional assurance of the integrity of the SCCS environment in which the application is running. With dynamic link-loading of service modules, viruses and other forms of impersonation are real threats. SCCS reduces the risk of these threats by requiring that modules be digitally signed and dynamically checking the identity and integrity of each module at attach time. The digital signature represents the SP provider's attestation of ownership and a guarantee that the SP, with the SP adaptation layer, conforms to the SCCS SPI specification. SCCS checks the authenticity of every SP that is loaded on the local system. Verification improves the chances that any modification, whether accidental or malicious, may be detected prior to performing trusted operations.

Module verification has three aspects:

- verification of module identity based on a digitally signed certificate
- verification of object code integrity based on a signed hash of the object
- tightly binding the verified module identity with the verified set of object code

Chapter 3. Cryptographic Module Manager

The Cryptographic Module Manager administers the Cryptographic Service Providers (CSP) modules that may be installed on the local system, and defines a common API for accessing CSP modules. All cryptography functions are implemented by the CSPs. This localizes all cryptography into exchangeable modules. SCCS administers a queryable registry of local CSPs. The registry lists the locally accessible CSPs and their cryptographic services (and algorithms).

The nature of the cryptographic functions contained in any particular CSP depends on the task the CSP was designed to perform. For example, a VISA smartcard would be able to digitally sign credit card transactions on behalf of the card's owner. A digital employee badge would be able to authenticate a user for physical or electronic access.

The Cryptographic Module Manager doesn't assume any particular form for a CSP. CSPs can be implemented in hardware, software, or both; operationally the distinction must be transparent. The two visible distinctions between hardware and software implementations are the degree of trust the application receives by using a given CSP, and the cost of developing that CSP. A hardware implementation should be more tamper-resistant than a software implementation. Hence a higher level of trust is achieved by the application.

Software CSPs are the default and are portable. They can be carried as an executable file. The modules that implement a CSP must be digitally signed (to authenticate their origin and integrity), and they should be made as tamper-resistant as possible. This requirement extends to software and hardware implementations. Multiple CSPs may be loaded and active within the SCCS at any time, and a single application may use multiple CSPs concurrently. Interpreting the resulting level of trust and security is the responsibility of the application or the trust policy module used by the application.

The Cryptographic Module Manager defines a high-level, certificate-based API for cryptographic services to support application development. This API is documented in *the Secure Cryptography and Certificate Services Toolkit Application Programming Interface Specification* manual. The Cryptographic Module Manager defines a lower-level Service Provider Interface (SPI) that more closely resembles typical CSP APIs, and provides CSP developers with a single interface to support. A CSP may or may not support multi-threaded applications. For information on the SPI interface, see the *Secure Cryptography and Certificate Services Toolkit Cryptography Service Provider Interface Specification* manual.

3.1 Supporting Legacy CSPs

CSPs existed prior to the definition of the SCCS Cryptographic API. These legacy CSPs have defined their own APIs for cryptographic services. These interfaces are CSP-specific, nonstandard, and (in general) low-level key-based interfaces. They present a considerable development effort to the application developer attempting to secure an application by using those services.

Acknowledging legacy CSPs, the SCCS defines an optional adaptation layer between the Cryptographic Module Manager and a CSP. The adaptation layer allows the CSP vendor to implement a shim to map the SCCS SPI to the CSP's existing API and to implement any additional management functions that are required for the CSP to function as an service provider module in the extensible SCCS. New CSPs may support the SCCS SPI directly (without the aid of an adaptation layer).

3.2 Cryptographic Services API

The security services API defined by the Cryptographic Module Manager are certificate-based. This contrasts with the approach taken by many CSPs, where low-level concepts such as key type, key size, hash functions, and byte ordering are the standard granularity of interface options. The Cryptographic Module Manager hides these behind high-level operations such as:

- SignData
- VerifyData
- DigestData
- EncryptData
- DecryptData
- GenerateKeyPair
- GenerateNonce

Security-conscious applications use these high-level concepts to provide authentication, data integrity, data and communication privacy, and nonrepudiation of messages to the end-users.

A CSP may implement any algorithm. For example, CSPs may provide one or more of the following algorithms, in one or more modes:

- Bulk encryption algorithm: DES, Triple DES, IDEA, RC2, RC4, RC5, Blowfish, CAST
- Digital signature algorithm: RSA, DSS
- Key negotiation algorithm: Diffie Hellman
- Cryptographic hash algorithm: MD4, MD5, SHA
- Unique identification number: hard-coded or random generated
- Random number generator: attended and unattended
- Encrypted storage: symmetric-keys, private-keys

The application's associated security context defines parameter values for the low-level variables that control the details of cryptographic operations. Setting input parameters to cryptographic algorithms is not a policy decision of the SCCS Framework. Applications use CSPs that provide the services and features required by the application. For example, an application issuing a request to *EncryptData* may reference a security context that defines the following parameters:

- The algorithm to be used (such as RC5)
- Algorithm-specific parameters (such as key length)
- The cryptographic variables (such as the key)

Most applications will use default SCCS contexts that are available through API function calls such as CSSM_CSP_CreateSignatureContext. Typically a distinct context will be used for encrypting, hashing, and signing. For a given application, once initialized, these contexts will change little (if at all) during the application's execution or between executions. This allows the application developer to implement

security by manipulating certificates, using previously defined security contexts, and maintaining a high-level view of security operations.

Application developers who demand fine-grained control of cryptographic operations can achieve this by directly and repeatedly updating the security context to direct the CSP for each operation, and by using the Cryptographic Module Manager API *pass-through* feature.

3.3 Dependencies with the Key Recovery Module Manager

The Cryptographic Module Manager of the CSSM is responsible for handling the cryptographic functions of SCCS. In order to introduce the necessary dependencies between the cryptographic operations and the key recovery enablement operations, the cryptographic module manager of SCCS has been modified.

The cryptographic context data structure of the SCCS has been augmented to include the following key recovery extension fields:

- a usability field for key recovery
- a workfactor field for law enforcement key recovery

The usability field denotes whether a cryptographic context needs to have key recovery enablement operations (either for law enforcement or enterprise needs) performed before it can be used for cryptographic operations such as encrypt or decrypt. The workfactor field holds the allowable workfactor value for law enforcement key recovery. These two additional fields of the cryptographic context are not available to the API for modification. They are set by the KRMM when the latter makes the key recovery policy enforcement decision for law enforcement and enterprise policies.

Although the SCCS API has been left intact in the SCCS Toolkit, the behavior of some of the cryptographic functions has been modified somewhat to accommodate the above-mentioned extensions to the cryptographic context. The basic changes are as follows:

- invoke key recovery policy enforcement functions for cryptographic context creation and update operations
- set the usability field in the cryptographic context to render the context unusable if key recovery enablement operations are mandated
- check the cryptographic context usability field before allowing encrypt/decrypt operations to occur

Whenever a cryptographic context is created or updated using the SCCS API functions, the cryptographic module manager invokes a KRMM policy enforcement function module; the latter checks the law enforcement and enterprise policies to determine whether the cryptographic context defines an operation where key recovery is mandated, If so, the usability field value is set in the cryptographic context data structure to signify that the context is unusable until key recovery enablement operations are performed on this context. The usability field is essentially a bitmap that signifies whether key recovery is required by the law enforcement or enterprise key recovery policies. When the appropriate key recovery enablement operations are performed on this context, the bits in the usability field is appropriately toggled so that the cryptographic context becomes usable for the intended operations.

When the encryption /decryption operations of the SCCS are invoked, the cryptographic module manager checks the key recovery usability field in the cryptographic context to determine whether the context is usable for encryption / decryption operations. If the context is flagged as unusable, the encryption/decryption API function returns an error. When the appropriate key recovery enablement operations are performed on that context, the flag values are reset so that the context may then be usable for encryption/decryption.

Chapter 4. Key Recovery Module Manager

The Key Recovery Module Manager (KRMM) enables key recovery for cryptographic services obtained through the SCCS. It mediates all cryptographic services provided by the SCCS and applies the appropriate key recovery policy on all such operations. The Key Recovery Module Manager contains a Key Recovery Policy Table (KRPT), which defines the applicable key recovery policy for all cryptographic products.

KRMM routes the KR-API function calls made by an application to the appropriate KR-SPI functions. The KRMM also enforces the key recovery policy on all cryptographic operations that are obtained through the SCCS. It maintains key recovery state in the form of key recovery contexts.

Key Recovery Fields (KRFs) are generated so that they can be used by a Key Recovery Agent (KRA) to recover the original symmetric key, either because the user who generated the message has lost the key, or at the warranted request of law enforcement agents. The purpose of a KRSP is to properly generate the KRFs given a symmetric key and the appropriate Key Recovery Profile information. (See Section 4.2.2 for information on Key Recovery Profiles.)

Key Recovery Fields are required when a cryptographic context for symmetric encryption is created with a key length longer than that specified in the Key Recovery Policy Table (KRPT). The KRPT defines both the minimum key length, as well as an acceptable work factor, given the cryptographic algorithm and mode of encryption which are to be used. The work factor is the maximum number of key bits that can be left out when generating KRFs. If a KRF is created with a work factor specified, the Key Recovery Agent will only be able to recover a portion of the key, and reading the original message will require searching the remaining key space in order to find the key that will decrypt the message.

4.1 Introduction to Key Recovery

The term *key recovery* encompasses mechanisms that allow authorized parties to retrieve the cryptographic keys used for data confidentiality, with the ultimate goal of recovery of encrypted data. This section discusses the various types of key recovery mechanisms, the phases of key recovery, and the policies with respect to key recovery.

4.1.1 Key Recovery Types

There are two classes of key recovery mechanisms based on the way keys are held to enable key recovery: *key escrow* and *key encapsulation*. Key escrow techniques are based on the paradigm that the government or a trusted party called an *escrow agent*, holds the actual user keys or portions thereof. Key encapsulation techniques, on the other hand, are based on the paradigm that a cryptographically encapsulated form of the key is made available to parties that require key recovery; the encapsulation technique ensures that only certain trusted third parties called *recovery agents* can perform the unwrap operation to retrieve the key material buried inside. There may also be hybrid schemes that use some escrow mechanisms in addition to encapsulation mechanisms.

An orthogonal way to classify key recovery mechanisms is based on the nature of the key that is either escrowed or encapsulated. Some schemes rely on the escrow or encapsulation of long-term keys, such as private keys, while other schemes are based on the escrow or encapsulation of ephemeral keys such as bulk encryption keys. Since escrow schemes involve the actual archival of keys, they typically deal with long-term keys, in order to avoid the proliferation problem that arises when trying to archive the myriad ephemeral keys. Key encapsulation techniques, on the other hand, usually operate on the ephemeral keys

For a large class of key recovery (escrow as well as encapsulation) schemes, there are a set of *key recovery fields* that accompany an enciphered message or file. These key recovery fields may be used by the appropriate authorized parties to recover the decryption key and or the plaintext. Typically, the key recovery fields comprise information regarding the key escrow or recovery agent(s) that can perform the recovery operation; they also contain other pieces of information to enable recovery.

In a key escrow scheme for long-term private keys, the "escrowed" keys are used to recover the ephemeral data confidentiality keys. In such a scheme, the key recovery fields may comprise the identity of the escrow agent(s), identifying information for the escrowed key, and the bulk encryption key wrapped in the recipient's public key (which is part of an escrowed key pair); thus the key recovery fields include the key exchange block in this case. In a key escrow scheme where bulk encryption keys are archived, the key recovery fields may comprise information to identify the escrow agent(s), and the escrowed key for that enciphered message.

Table 1. Comparison of Typical Escrow and Encapsulation Schemes

Mechanism	*Advantages*	*Disadvantages*
Key Escrow for long-term keys	• Minimal change to existing communication protocols • no latency in using ephemeral key	• lack of privacy for individuals • coarse granularity for recoverable keys • latency involved in obtaining and using long-term key • need for individual to belong to government approved public key infrastructure (PKI)
Key Encapsulation for ephemeral keys	• more privacy for individuals • fine granularity for recoverable keys • no latency to obtain and use public keys • no need for individuals to belong to government approved public key infrastructure (PKI)	• requires modifications to existing communications protocols • some latency involved in key encapsulation for every ephemeral key

In a typical key encapsulation scheme for ephemeral bulk encryption keys, the key recovery fields are distinct from the key exchange block, (if any.) The key recovery fields identify the recovery agent(s), and contain the bulk encryption key encapsulated using the public keys of the recovery agent(s).

The key recovery fields are generated by the party performing the data encryption, and associated with the enciphered data. To ensure the integrity of the key recovery fields, and its association with the encrypted data, it may be required for processing by the party performing the data decryption. The processing mechanism ensures that successful data decryption cannot occur unless the integrity of the key recovery fields is maintained at the receiving end. In schemes where the key recovery fields contain the key exchange block, decryption cannot occur at the receiving end unless the key recovery fields are processed to obtain the decryption key; thus the integrity of the key recovery fields are automatically verified. In schemes where the key recovery fields are separate from the key exchange block, additional processing has to happen to ensure that decryption of the ciphertext occurs only after the integrity of the key recovery fields are verified.

Table 1 illustrates the advantages and disadvantages of typical examples of key escrow and key encapsulation schemes. With escrow schemes for long terms private keys, a major advantage is that the cryptographic communication protocol needs minimal adaptation (since the key recovery fields and the key exchange block are one and the same in most cases); however, the serious disadvantages are the lack of privacy for individuals (since their private keys are held by a separate entity), and the lack of granularity with respect to the recoverable keys. Additionally, there is the burden that every individual has to obtain and use public keys through an approved public key infrastructure in order for the key recovery scheme to work.

The major advantage to key encapsulation schemes based on ephemeral keys are that there is much greater privacy for the individuals; they can generate and keep their own private keys. Each ephemeral key can be recovered independently, so there is maximal granularity with respect to the recoverable keys. A disadvantage is that the communication protocol between sending and receiving parties needs more elaborate adaptation to allow the flow of the encapsulated key (which is separate from the key exchange block.) Another disadvantage is that there is some performance penalty in key encapsulation for each ephemeral key; however, this may be minimized by caching techniques.

4.1.2 Key Recovery Phases

The process of cryptographic key recovery involves three major phases. First, there is an optional *key recovery registration* phase where the parties that desire key recovery perform some initialization operations with the escrow or recovery agents; these operations include obtaining a user public key certificate (for an escrowed key pair) from an escrow agent, or obtaining a public key certificate from a recovery agent. Next, parties that are involved in cryptographic associations have to perform operations to enable key recovery (such as the generation of key recovery fields, etc.) - this is typically called the *key recovery enablement* phase. Finally, authorized parties that desire to recover the data keys, do so with the help of a recovery server and one or more escrow agents or recovery agents - this is the *key recovery request* phase.

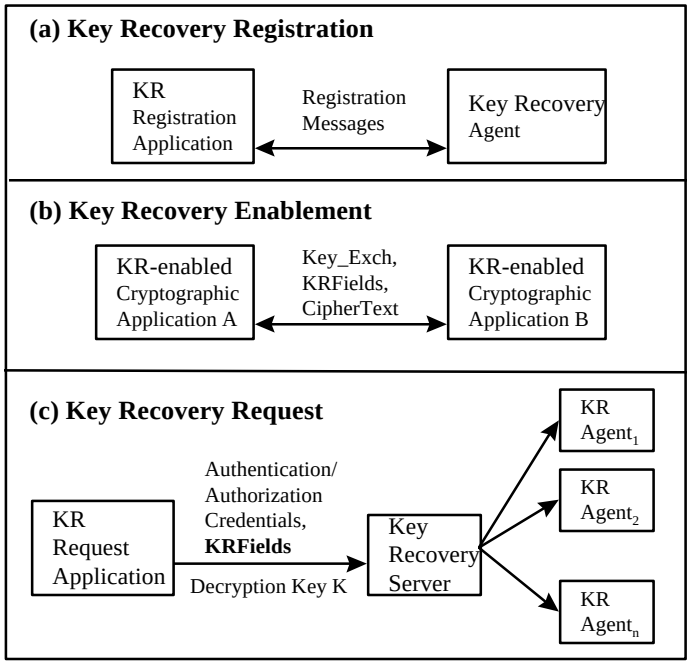

Figure 5. Key Recovery Phases

Figure 5 illustrates the three phases of key recovery. In Figure 5(a), a key recovery client registers with a recovery agent prior to engaging in cryptographic communication. In Figure 5(b), two key-recovery-enabled cryptographic applications are communicating using a key encapsulation mechanism; the key recovery fields are passed along with the ciphertext and key exchange block, to enable subsequent key recovery. The key recovery request phase is illustrated in Figure 5(c), where the key recovery fields are provided as input to the key recovery server along with the authorization credentials of the client requesting service. The key recovery server interacts with one or more local or remote key recovery agents to reconstruct the secret key that can be used to decrypt the ciphertext.

It is envisioned that governments or organizations will operate their own recovery server hosts independently, and that key recovery servers may support a single or multiple key recovery mechanisms. There are a number of important issues specific to the implementation and operation of the key recovery servers, such as vulnerability and liability. The issues with respect to the key recovery server and agents are beyond the scope of this chapter.

4.1.3 Lifetime of Key Recovery Fields

Cryptographic products fall into one of two fundamental classes: *archived-ciphertext products*, and *transient-ciphertext products*. When the product allows either the generator or the receiver of ciphertext to archive the ciphertext, the product is classified as an archived-ciphertext product. On the other hand, when the product does not allow the generator or receiver of ciphertext to archive the ciphertext, it is classified as a transient-ciphertext product.

It is important to note that the lifetime of key recovery fields should never be greater than the lifetime of the associated ciphertext. This is somewhat obvious, since recovery of the key is only meaningful if the key can be used to recover the plaintext from the ciphertext. Hence, when archived-ciphertext products are key recovery enabled, the key recovery fields are typically archived as long as the ciphertext. Similarly, when transient-ciphertext products are key recovery enabled, the key recovery fields are associated with the ciphertext for the duration of its lifetime. It is not meaningful to archive key recovery fields without archiving the associated ciphertext.

4.1.4 Key Recovery Policy

Key recovery policies are mandatory policies that are typically derived from jurisdiction-based regulations on the use of cryptographic products for data confidentiality. Often, the jurisdictions for key recovery policies coincide with the political boundaries of countries, in order to serve the law enforcement and intelligence needs of these political jurisdictions. Political jurisdictions may choose to define key recovery policies for cryptographic products based on export, import, or use controls. Enterprises may define internal and external jurisdictions, and may mandate key recovery policies on the cryptographic products within their own jurisdictions.

Key recovery policies come in two flavors: *key recovery enablement policies* and *key recovery inter-operability policies*. Key recovery enablement policies specify the exact cryptographic protocol suites (algorithms, modes, key lengths etc.) and perhaps usage scenarios, where key recovery enablement is mandated. Furthermore, these policies may also define the number of bits of the cryptographic key that may be left out of the key recovery enablement operation; this is typically referred to as the *workfactor*. Key recovery inter-operability policies specify to what degree a key-recovery-enabled cryptographic product is allowed to inter-operate with other cryptographic products.

4.1.5 Operational Scenarios for Key Recovery

There are three basic operational scenarios for key recovery:

- law enforcement key recovery
- enterprise key recovery
- individual key recovery

In the law enforcement scenario, key recovery is mandated by the jurisdictional law enforcement authorities in the interest of national security and law enforcement. For a specific cryptographic product, the key recovery policies for multiple jurisdictions may apply simultaneously. The policies (if any) of the jurisdiction(s) of manufacture of the product, as well as the jurisdiction of installation and use, need to be applied to the product such that the most restrictive combination of the multiple policies is used. Thus, law enforcement key recovery is based on mandatory key recovery policies; these policies are logically bound to the cryptographic product at the time the product is shipped. There may be some mechanism for vendor-controlled updates of such law enforcement key recovery policies in existing products; however, organizations and end users of the product are not able to modify this policy at their discretion. The escrow or recovery agents used for this scenario of key recovery need to be strictly controlled in most cases, to ensure that these agents meet the eligibility criteria for the relevant political jurisdiction where the product is being used.

Enterprise key recovery allows enterprises to enforce stricter monitoring of the use of cryptography, and the recovery of enciphered data when the need arises. Enterprise key recovery is also based on a mandatory key recovery policy; however, this policy is set (perhaps, through administrative means) by the organization or enterprise at the time of installation of a recovery enabled cryptographic product. The enterprise key recovery policy should not be modifiable or circumventable by the individual using the cryptographic product. Enterprise key recovery mechanisms may use special enterprise authorized escrow or recovery agents.

Individual key recovery is user-discretionary in nature, and is performed for the purpose of recovery of enciphered data by the owner of the data, if the cryptographic keys are lost or corrupted. Since this is a non-mandatory key recovery scenario, it is not based on any policy that is enforced by the cryptographic product; rather, the product may allow the user to specify when individual key recovery enablement is to be performed. There are few restrictions on the use of specific escrow or recovery agents.

In each of these scenarios, key recovery may be desired. However, the detailed aspects or characteristics of these three scenarios are somewhat varied. Table 2 summarizes the specific characteristics of the different operational scenarios.

Table 2. Comparison of Key Recovery Scenarios

Properties	Law Enforcement	Enterprise	Individual
mandatory key recovery	yes	yes	no
Escrow or Recovery Agents are controlled	yes	yes	no
recovery enablement needs to be noncircumventable	yes	yes	no
dual sided key recovery enablement	maybe	maybe	no
use of workfactor when generating KRF	maybe	no	no
KRF contains agent identification	yes	maybe	no
user registration needed at escrow or recovery agents	maybe	maybe	maybe
user authentication information needed within KRF	no	no	yes
end-user knowledge/cooperation required	no	no	yes

Key recovery enabled cryptographic products must be designed so that the key recovery enablement operation is mandatory and noncircumventable in the law enforcement and enterprise scenarios, and

discretionary for the individual scenario. The escrow and recovery agent(s) that are used for law enforcement and enterprise scenarios must be tightly controlled so that the agents are validated to belong to a set of authorized or approved agents. In the law enforcement and enterprise scenarios, the key recovery process typically needs to be performed without the knowledge and cooperation of the parties involved in the cryptographic association.

The components of the key recovery fields also varies somewhat between the three scenarios. In the law enforcement scenario, the key recovery fields must contain identification information for the escrow or recovery agent(s) whereas for the enterprise and individual scenarios, the agent identification information is not so critical, since this information may be available from the context of the recovery enablement operation. For the individual scenario, there needs to be a strong user authentication component in the key recovery fields, to allow the owner of the key recovery fields to authenticate themselves to the agents; however, for the enterprise and law enforcement scenarios, the authorization credentials checked by the agents may be in the form of legal documents, or enterprise-authorization documents for key recovery, that may not be tied to any authentication component in the key recovery fields. For the law enforcement and enterprise scenarios, the key recovery fields may contain recovery information for both the generator and receiver of the enciphered data; in the individual scenario, only the information of the generator of the enciphered data is typically included (at the discretion of the generating party.)

4.2 Components of SCCS Key Recovery Operations

The Key Recovery Module Manager is responsible for handling the KR-API functions and invocation of the appropriate KR-SPI functions. The KRMM enforces the key recovery policy on all cryptographic operations that are obtained through the SCCS. It maintains key recovery state in the form of key recovery contexts.

4.2.1 Operational Scenarios

The SCCS architecture supports three distinct operational scenarios for key recovery, namely, key recovery for law enforcement purposes, enterprise purposes, and individual purposes. The law enforcement and enterprise scenarios for key recovery are mandatory in nature, thus the SCCS layer code enforces the key recovery policy with respect to these scenarios through the appropriate sequencing of KR-API and cryptographic API calls. On the other hand, the individual scenario for key recovery is completely discretionary and is not enforced by the SCCS layer code. The application/user requests key recovery operations using the KR-APIs at their discretion.

The three operational scenarios for key recovery enablement drive certain design decisions with respect to the SCCS. The details of the specific features of the operational scenarios are described in the following subsections.

4.2.2 Key Recovery Profiles

The KRSPs require certain pieces of information related to the parties involved in a cryptographic association in order to generate and process key recovery fields. These pieces of information (such as the public key certificates of the key recovery agents) are contained in *key recovery profiles*. The profiles contain all of the parameters for key recovery field generation and processing for that KRSP. The KRSP GUID specifies the KRSP for which a given key recovery profile record is relevant.

The information contained in the profile comprises the following:

- a set of Key Recovery Agent (KRA) certificate chains for law enforcement key recovery
- a set of Key Recovery Agent (KRA) certificate chains for enterprise key recovery

- a set of Key Recovery Agent (KRA) certificate chains for individual key recovery
- authentication information for individual key recovery
- a set of key recovery flags that fine tune the behavior of a KRSP
- a public key certificate chain for the user
- an extension field

The key recovery profiles support a list of KRA certificate chains for each of the law enforcement, enterprise, and individual key recovery scenarios, respectively. While the profile allows full certificate chains to be specified for the KRAs, it also supports the specification of leaf certificates; in such instances, the KRSP and the appropriate TP modules are expected to dynamically discover the intermediate certificate authority certificates up to the root certificate of trust. One or more of these certificate chains may be set to NULL, if they are not needed or supported by the KRSP involved.

The user public key certificate chain is also part of a profile. This is a necessary parameter for certain key escrow and encapsulation schemes. Certain schemes support the notion of a user authentication field for individual and enterprise key recovery. This field is used by the key recovery server and/or agent(s) to verify that the individual or enterprise requesting key recovery is the owner of the key recovery fields, and can authenticate themselves based on the authentication information contained in the key recovery fields. One or both of these authentication information fields may be set to NULL, if their use is not required or supported by the KRSP involved.

The key recovery flags are defined values that are pertinent for a large class of escrow and recovery schemes. The extension field is for use by the KRSPs to define additional semantics for the key recovery profile. These extensions may be flag parameters or value parameters. The semantics of these extensions are defined by a KRSP; the application that uses profile extensions has to be cognizant of the specific extensions for a particular KRSP. However, it is envisioned that these extensions will be for optional use only. KRSPs are expected to have reasonable defaults for all such extensions; this is to ensure that applications do not need to be aware of specific KRSP profile extensions in order to get basic key recovery enablement services from a KRSP. Whenever the extensions field is set to NULL, the defaults should be used by a KRSP.

The profiles for the local and remote parties involved in a cryptographic association are input parameters to several of the KR-API functions. Thus, application layer code is allowed to specify the profiles for all KR-API functions where profiles are relevant. These profiles are used by the KRSP to perform its operations. The KRSP maintains a default for the local as well as the remote profiles that it uses whenever the profiles received through the KR-API functions are set to NULL or when the profiles contains NULL values for relevant fields. For example, if the local profile passed through the KR-API has NULL for the LE KRA list entry, the corresponding values from the KRSP default local profile are used by the KRSP when generating the LE KRFs. These default profiles are read by the KRSP (at the time it is initialized) from a KRSP Configuration File.

4.2.3 Key Recovery Context

All operations performed by the KRSPs are performed within a *key recovery context*. A key recovery context is programmatically equivalent to a cryptographic context; however the attributes of a key recovery context are different from those of other cryptographic contexts. There are three kinds of key recovery contexts - registration contexts, enablement contexts and recovery request contexts. A key recovery context contains state information that is necessary to perform key recovery operations. When the KR-API functions are invoked by application layer code, the KRMM passes the appropriate key recovery context to the KRSP using the KR-SPI function parameters.

A key recovery registration context contains no special attributes. A key recovery enablement context maintains information about the identities and profiles of the local and remote parties for a cryptographic association. When the KR-API function to create a key recovery enablement context is invoked, the identities and key recovery profiles for the specified communicating peers may be specified by the application layer code using the API parameters; however, if the profile parameters are set to NULL, the profiles may be retrieved automatically from the KRPR if a match is found with the specified local and remote identities. A key recovery request context maintains a set of key recovery fields which are being used to perform a recovery request operation, and a set of flags that denotes the operational scenario of the recovery request operation. Since the establishment of a context implies the maintaining of state information within the SCCS, contexts acquired should be released as soon as their need is over.

4.2.4 Key Recovery Policy

The SCCS enforces the applicable key recovery policy on all cryptographic operations. There are two key recovery policies enforced by the SCCS, a law enforcement (LE) key recovery policy, and the enterprise (ENT) key recovery policy. Since the requirements for these two mandatory key recovery scenarios are somewhat different, they are implemented by different mechanisms within the SCCS.

The law enforcement key recovery policy is predefined (based on the political jurisdictions of manufacture and use of the cryptographic product) for a given product. The parameters on which the policy decision is made are predefined as well. Thus, the LE key recovery policy is implemented using two key recovery policy tables, one table corresponding to the policy of the jurisdiction of manufacture, and the second corresponding to the jurisdiction of use of the product. These two LE policy tables are consulted by the key recovery policy enforcement function in the SCCS. The LE policy tables are implemented as two separate physical files for ease of implementation and upgrade (as law enforcement policies evolve over time); however, these files are protected using the same integrity mechanisms as the SCCS module, and thus has the same assurance properties.

The ENT key recovery policy, on the other hand, could vary anywhere between being set to NULL, and being very complex (e.g. based on parameters such as time of day.) Enterprises are allowed total flexibility with respect to the enterprise key recovery policy. The enterprise policy is implemented within the SCCS by invoking a key recovery policy function that is defined by the enterprise administrator. The KR-API provides a function that allows an administrator to specify the name of a file that contains the enterprise key recovery policy function. This API function allows the administrator to establish a passphrase for subsequent calls on this function. This mechanism assures a level of access control on the enterprise policy, once a policy function has been established. It goes without saying that the file containing the policy function should be protected using the maximal possible protection afforded by the operating system platform. The actual structure of the policy function file is operating system platform specific.

Every time a cryptographic context handle is returned to application layer code, the SCCS enforces the LE and ENT key recovery policies. For the LE policy, the SCCS policy enforcement function and the LE policy tables are used. For the ENT policy, the ENT policy function file is invoked in an operating system platform specific way. If the policy check determines that key recovery enablement is required for either LE or ENT scenarios, then the context is flagged as unusable, by setting specific bits of the context usability field. Otherwise, the context is flagged as usable. An unusable context handle becomes flagged as usable only after the appropriate key recovery enablement operation is completed using that context handle. A usable context handle can then be used to perform cryptographic operations.

4.2.5 Key Recovery Enablement Operations

The SCCS key recovery enablement operations comprise the generation and processing of key recovery fields. Within a cryptographic association, key recovery field generation is performed by the sending side; key recovery field processing is performed on the receiving side to ensure that the integrity of the recovery fields have been maintained in transmission between the sending and receiving sides. These two vital

operations are performed via the CSSM_KR_GenerateRecoveryFields and the CSSM_KR_ProcessRecoveryFields functions respectively.

The key recovery fields generated by the SCCS potentially comprises three sub-fields, for law enforcement, enterprise and individual key recovery scenarios, respectively. The law enforcement and enterprise key recovery sub-fields are generated when the law enforcement and enterprise bits of the usability field is appropriately set in the cryptographic context used to generate the key recovery fields. The individual key recovery sub-fields are generated when a certain flag value is set while invocation of the API function to generate the key recovery fields. The processing of the key recovery fields only applies to the law enforcement and enterprise key recovery sub-fields; the individual key recovery sub-fields are ignored by the key recovery fields processing function.

4.2.6 Key Recovery Registration and Request Operations

The SCCS also supports the operations of registration and recovery requests. The KRSP exchanges messages with the appropriate key recovery agent/server to obtain the results required. If additional inputs are required for the completion of the operation, the supplied callback may be used by the KRSP. The recovery request operation can be used to request of batch of recoverable keys. The result of the registration operation is a key recovery profile data structure, while the results of a recovery request operation are a set of recovered keys.

4.3 Relationship of the Key Recovery and Cryptographic Module Managers

There is some degree of interdependence between the KRMM and the cryptographic module manager in a key recovery mechanism-independent way. For example, the cryptographic module manager must invoke the key recovery policy checking function of the KRMM, which checks the law enforcement and enterprise policies for key recovery. A cryptographic context maintained by the cryptographic module manager must be made available to the KRMM so that the relevant key recovery fields may be generated or processed. The KRMM may modify the extension fields within the cryptographic context on which it is operating; these extension fields are then checked by the encrypt and decrypt operations of the cryptographic module manager. All of the above essentially imply that between the cryptographic and key recovery module managers, there is a way to share objects such as cryptographic contexts and/or their handles.

The primary rationale for this architecture is that it allows the KRSPs and the CSPs to operate oblivious of one another, thus satisfying the primary goal of this architecture. The interdependencies between the crypto and key recovery operations are captured completely within the two module managers. It may be argued that this approach exposes the key recovery APIs to the protocol handler code, and that this may be undesirable; however the protocol handler code will *have* to be key recovery aware, whether the KR-API is exposed to the application or not. There are certain parameters that have to be obtained from the application level that are essential to performing key recovery enablement operations, the application will have to be modified to handle these additional parameters for key recovery. For example, in an implementation of a key recovery enabled SSL, the code would have to handle special cipher suites and cipher specs in order to negotiate a key recovery mechanism with the receiving party; so exposing the KR-API to the SSL applications does not appear to be real disadvantage. On the other hand, hiding the KRMM under the crypto module manager, or incorporating the KR mechanism into a CSP has the obvious disadvantage that the CSP needs to be modified, and needs to be cognizant of specific KR mechanisms. Therefore, there does not appear to be any advantage to positioning the KRMM under the crypto module manager.

4.4 Key Recovery API

SCCS defines a set of APIs that allow key recovery enablement of cryptographic services provided by the SCCS. These functions include:

- an operation that establishes the filename containing the enterprise-based key recovery policy function for use by SCCS.
- key recovery operations that create key recovery registration, enablement, and request contexts.
- an operation that returns the key recovery policy pertaining to a given cryptographic context.
- registration operations that generate key recovery profiles.
- enablement operations that generate and process key recovery fields.
- request operations that initiate key recovery and retrieve recovered keys.

For detailed information on the key recovery functions, see *Secure Cryptography and Certificate Services Toolkit Application Programming Interface.*

Chapter 5. Trust Policy Module Manager

The Trust Policy Module Manager administers the trust policy modules that may be installed on the local system and defines a common API for these libraries. The Trust Policy (TP) API allows applications to request security services that require "policy review and approval" as the first step in performing the operation. Operations defined in the TP API include verifying trust in

- A certificate for signing or revoking another certificate
- A user or user-agent to perform an application-specific action
- The issuer of a certificate revocation list

A digital certificate binds an identification in a particular domain to a public key. When a certificate is issued (created and signed) by a certificate authority (CA), the binding between key and identity is attested by the digital signature on the certificate. The issuing authority also associates a level of trust with the certificate. The actions of the user, whose identity is bound to the certificate, are constrained by the trust policy governing the certificate's usage domain. A digital certificate is intended to be an unforgeable credential in cyberspace.

The use of digital certificates is the basis on which the SCCS is designed. The SCCS assumes the concept of digital certificates in its broadest sense, that is an identity bound to a public key. Certificates are often used for identification, authentication, and authorization. The way in which applications interpret and manipulate the contents of certificates to achieve these ends is defined by the real world trust model the application has chosen as its model for trust and security.

The primary purpose of a Trust Policy (TP) service provider is to answer the question "Is this certificate trusted for this action?" The SCCS Trust Policy API defines the generic operations that should be defined for certificate-based trust in every application domain. The specific semantics of each operation is defined by the

- Application domain
- Trust model
- Policy statement for a domain
- Certificate type

The trust model is expressed as an executable policy that is used/invoked by all applications that ascribe to that policy and the trust model it represents.

As an infrastructure, SCCS is policy neutral; it does not incorporate any single policy. For example, the verification procedure for a credit card certificate should be defined and implemented by the credit company issuing the certificate. Employee access to a lab housing a critical project should be defined by the company whose intellectual property is at risk. Rather than defining policies, SCCS provides the infrastructure for installing and managing policy-specific modules. This ensures extensibility of certificate-based trust on every platform hosting SCCS.

Different trust policies define different actions that may be requested by an application. There are also a few basic actions that should be common to every trust policy. These actions are operations on the basic objects used by all trust models. The basic objects common to all trust models are certificates and certificate revocation lists (CRLs). The basic operations on these objects are sign, verify, and revoke.

Application developers and trust domain authorities benefit from the ability to define and implement policy-based modules. Application developers are freed from the burden of implementing a policy description and certifying that their implementation conforms. Instead, the application only needs to build in a list of the authorities and certificate issuers it uses.

Domain authorities also benefit from an infrastructure that supports trust policy modules. Authorities are sure that applications using their module(s) will adhere to the policies of the domain. In addition, dynamic download of trust modules (possibly from remote systems) ensures timely and accurate propagation of policy changes. Individual functions within the module may combine local and remote processing. This flexibility allows the module developer to implement policies based on the ability to communicate with a remote authority system. This also allows the policy implementation to be decomposed in any convenient distributed manner.

Implementing a trust policy module may or may not be tightly coupled with one or more certificate library modules and one or more data storage library modules. The trust policy embodies the semantics of the domain. The certificate library and the data storage library embody the syntax of a certificate format and operations on that format. A trust policy can be completely independent of certificate format, or it may be defined to operate with a small number of certificate formats. A trust policy implementation may invoke a certificate library module and/or a data storage library module to manipulate certificates.

5.1 Trust Policy API

SCCS provides trust policy operations on certificates and certificate revocation (CRL) lists. These operations include:

- trust policy operations, such as signing, verifying, or revoking, on individual certificates and CRLs
- trust policy operations on groups of certificates, such as constructing an ordered group, verifying the signatures on a group, and removing certificates from a group.
- Pass-through operations for unique certificate and CRL operations

For detailed information on each of these functions, see Secure Cryptography and Certificate Services Toolkit Application Programming Interface.

Chapter 6. Certificate Library Module Manager

The Certificate Library Module Manager administers the Certificate Libraries that may be installed on the local system. It defines a common API for these libraries. The API allows applications to manipulate memory-resident certificates and certificate revocation lists.

Operations defined in the API include create, sign, verify, and extract field values. The certificate libraries modules implement all certificate operations. Application-invoked calls are dispatched to the appropriate library module. Each library incorporates knowledge of certificate data formats and how to manipulate that format. The SCCS Certificate Module Manager administers a queryable registry of local libraries. The registry enumerates the locally accessible libraries and attributes of those libraries, such as the certificate type manipulated by each registered library.

The primary purpose of a Certificate Library (CL) module is to perform memory-based, syntactic manipulations on the basic objects of trust: certificates and certificate revocation lists (CRLs). The data format of a certificate will influence (if not determine) the data format of CRLs used to track revoked certificates. For this reason, these objects should be manipulated by a single, cohesive library. Certificate library modules incorporate detailed knowledge of data formats. The Certificate Library Module Manager defines API calls to perform security operations, (such as signing, verifying, revoking, viewing, etc.) on memory-resident certificates and CRLs. The mechanics of performing these operations is tightly bound to the data format of a given certificate. One or more modules may support the same certificate format, such as X.509 DER encoded certificates, SDSI certificates, and SPKI certificates.

As new standard formats are defined and accepted by the industry, certificate library modules will be defined and implemented by industry members and used directly and indirectly by many applications. Certificate library modules encapsulate certificate and CRL data formats from the semantics of trust policies, which are implemented in trust policy modules.

Certificate library modules manipulate memory-based objects only. The persistence of certificates and CRLs is an independent property of these objects. It is the responsibility of the application and/or the trust policy module to use data storage modules to make these objects persistent (if appropriate). It must be possible for the storage mechanism used by a data storage module to be independent of the other modules. It must also be possible to design a certificate library module that depends on the storage mechanism of a data storage library module.

Application developers and trust policy module developers both benefit from the extensibility of certificate library modules. Applications are free to use multiple certificate types without requiring the application developer to write format-specific code to manipulate certificates and CRLs. Without increased development complexity, multiple certificate formats can be used on one system, within one application domain, or by one application. CAs who issue certificates also benefit. Dynamically downloading certificate libraries ensures timely and accurate propagation of data-format changes.

6.1 Certificate Library Services API

The Certificate Library Services API defines numerous operations on memory-resident certificates and certificate revocation lists (CRLs) as required by every certificate type. These operations include:

- Creating new certificates and new CRLs
- Signing existing certificates and existing CRLs
- Viewing certificates
- Verifying certificates and CRLs
- Extracting values (e.g., public keys) from certificates
- Importing and exporting certificates of other data formats
- Revoking certificates
- Reinstating revoked certificates
- Searching certificate revocation lists
- Pass-through for unique, format-specific certificate and CRL operations

For detailed information on the Certificate Library API functions, see *Secure Cryptography and Certificate Services Toolkit Application Programming Interface.*

Chapter 7. Data Storage Library Module Manager

The Data Store Library Module Manager defines an API for secure, persistent storage of certificates and certificate revocation lists (CRLs). The API allows applications to search and select certificates and CRLs, and to query meta-data about each data store (such as its name, date of last modification, size of the data store, etc.) Data storage library modules implement data store operations. These modules may be drivers or gateways to traditional, full-featured Database Management Systems (DBMS), customized services layered over a file system, or access to other forms of stable storage. A data storage module may execute and store its data locally or remotely.

The primary purpose of a Data Storage Library (DL) module is to provide secure, persistent storage, retrieval, and recovery of certificates and certificate revocation lists (CRLs). The persistence of these generic trust objects is independent of the memory-based manipulations performed by certificate library modules. DL modules may be invoked by applications, trust policy modules, or certificate library modules that make decisions about the persistence of these trust objects.

A single DL module may be tightly tied to a Certificate Library (CL) module or may be independent of all CL modules. A data storage library that is tightly tied to a certificate library module implements a persistent storage mechanism that is dependent on the data format of the certificate. An independent data storage library implements a blob-based storage mechanism that stores certificates and CRLs without regard for their specific format. A single, physical data store managed by such DL modules may even contain individual certificates of different formats.

Each DL module can manage any number of independent, physical data stores. Each data store must have a logical name used by callers to refer to the persistent data store. Implementation of the DL module may use local file system facilities, commercial database management products, and custom stable storage devices.

A DL module is responsible for the integrity of the records it stores. If the DL module uses an underlying commercial database management system (DBMS), it may choose to further secure the data store by leveraging integrity services provided by the DBMS. DL modules that choose to implement persistence using the local file system or a custom stable storage device must decide which (if any) integrity mechanisms to provide.

7.1 Data Storage Library Services API

The Data Storage Library Services API defines the following two categories of operations. For detailed information on the Data Storage Library API functions, see *Secure Cryptography and Certificate Services Toolkit Application Programming Interface.*

- Data store management functions
  The data store management functions operate on a data store as a single unit. These operations include opening and closing data stores, creating and deleting data stores, and importing and exporting data stores. A data store may contain certificates only, certificate revocation records only, or both. It is unusual for a DL module to manage a data store containing both certificates and certificate revocation records, but there is nothing in the SCCS or the DL module API that prevents a DL module from implementing persistence in this manner. Typically, separate physical data stores are used to store certificates and CRLs.
- Persistence operations on certificates and certificate revocation lists
  The persistence operations on data stores include:
  - Adding new certificates and new certificate revocation records
  - Updating existing certificates
  - Deleting certificates and certificate revocation records
  - Retrieving certificates and certificate revocation records
  - Pass-through for unique, module-specific operations

Chapter 8. Service Provider Modules

All cryptographic and key recovery functions, as well as the trust policies, certificates, and data store functions are performed by service provider (SP) modules. The SCCS framework itself only manages the interactions between service provider modules and applications that use them. The Secure Cryptography and Certificate Services Architecture supports the following types of service providers.

- Cryptographic Service Providers (CSPs)
- Key Recovery Service Providers (KRSPs)
- Trust Policy Modules (TPs)
- Certificate Library Modules (CLs)
- Data Storage Library Modules (DLs)

This chapter presents a brief overview of each type of service provider module. For a detailed discussion of the SCCS interface the service provider modules must support, see the individual interface specifications provided with the SCCS Toolkit documentation set. Independent software vendors who develop modules for use with SCCS must support the interface specifications(s) described in these documents. The modules may implement all or a subset of these APIs. A single module may also provide services in multiple categories of service. These are called multi-service modules.

Several service provider modules are provided with the SCCS Toolkit. These modules are described in Section 8.6.

8.1 Cryptographic Service Provider Modules

Cryptographic service providers (CSPs) are modules equipped to perform cryptographic operations and to securely store private keys. A CSP may implement one or more of the following cryptographic functions:

- Bulk encryption algorithm
- Digital signature algorithm
- Cryptographic hash algorithm
- Unique identification number
- Random number generator
- Secure key storage
- Custom facilities unique to the CSP

A CSP may be implemented in software, hardware, or both. All CSPs must enable encrypted storage for private keys and variables. CSPs must also deliver key management services, including key escrow, if it is supported. As a minimum, CSPs do not reveal key material unless it's been wrapped, but they must support importing, exporting, and generating keys. The key-generation module of a CSP should be made tamper resistant.

Every CSP must provide secured storage of private keys. Applications may query the CSP to retrieve private keys stored within the CSP. The CSP is responsible for controlling access to the private keys it secures. A callback function implemented by the requester is invoked by the CSP (or the CSP's adaptation layer) to obtain the identity and authorization of the user or process requesting the private key. Most

CSPs are capable of importing private keys created by other CSPs and providing secured storage for such keys.

8.2 Key Recovery Service Provider Modules

Key Recovery Service Providers (KRSPs) are modules that generate and process Key Recovery Fields (KRFs). The KRF may be used to retrieve the decryption key through the use of a Key Recovery Server and one or more Key Recovery Agents. KRSP APIs are defined to generate the KRFs prior to performing encryption, as well as to process the KRFs prior to decryption. This processing step is used to ensure the integrity of the KRFs prior to decrypting the data.

8.3 Trust Policy Modules

Trust Policy (TP) modules implement policies defined by certification authorities and institutions. Policies define the level of trust required before certain actions can be performed. Three basic categories or actions exist for all certificate-based trust domains:

- Actions on certificates
- Actions on certificate revocation lists
- Domain-specific actions (such as issuing a check or writing to a file).

The *SCCS Trust Policy Interface Specification* document defines the generic operations that should be supported by every Trust Policy module. Each module may choose to implement the subset of these operations that are required for its policy. When a Trust Policy function has determined the trustworthiness of performing an action, the Trust Policy function may invoke functions in the Certificate Library and Data Storage Library modules to carry out the mechanics of the approved action.

8.4 Certificate Library Modules

Certificate Library (CL) modules implement syntactic manipulation of memory-resident certificates and certificate revocation lists. The SCCS Certificate API defines the generic operations that should be supported by every CL module. Each module may choose to implement only those operations required to manipulate a specific certificate data format.

The implementation of the CL operations should be free of certificate semantics. Semantic interpretation of certificate values should be implemented in Trust Policy modules, layered services, and applications.

The SCCS Toolkit makes manipulation of certificates and certificate revocation lists orthogonal to persistence of those objects. Hence, it is not recommended that CL modules invoke the services of data storage library modules. Trust Policy modules, layered security services, and applications should make decisions regarding the persistence of certificates.

8.5 Data Storage Library Modules

A Data Storage library (DL) module provides stable storage for certificates and certificate revocation lists (CRLs). Stable storage could be provided by a:

- Commercially-available database management system product
- Native file system

- Custom hardware-based storage devices

Each DL module may choose to implement only those operations required to provide persistent storage for certificates and certificate revocation lists under its selected model of service.

Semantic interpretation of certificate values and CRL values is usually assumed to be implemented in Trust Policy modules. A pass-through function, `DL_PassThrough`, is defined in the DL API that allows each DL service provider to provide additional functions to store and retrieve certificates and CRLs, such as performance enhancing retrieval functions.

8.6 SCCS Toolkit Service Provider Modules

A number of service provider modules may be provided with the SCCS Toolkit. These modules can be incorporated into applications to perform cryptographic security operations The following sections describe the SCCS API functions supported by each service of these provider modules. For detailed information on the behavior of the individual APIs, see the SCCS service provider interface documents in the sccstk\doc subdirectory.

- Cryptographic Service Provider Module
  There are three cryptographic modules which may be provided with SCCS.

  IBM Software Cryptographic Service Provider, Version 1.0
  IBM PKCS11 Multi-Service Module, Version 1.0
  IBM CCA Multi-Service Module, Version 1.0

- Trust Policy Module
  There are two trust policy modules which may be provided with SCCS.

  IBM Standard Trust Policy, Version 1.0
  IBM Extended Trust Policy, Version 1.0

- Certificate Library Module
  There is one certificate library module which may be provided with SCCS.

  IBM Certificate Library, Version 1.0

- Data Store Library Module
  There is one data store library module which may be provided with SCCS.

  IBM Data Library, Version 1.0

- Key Recovery Service Provider Module
  There is one key recovery service module which may be provided with SCCS

  IBM Key Recovery Service Provider, Version 1.0

8.6.1 IBM Software Cryptographic Service Provider, Version 1.0

Files required:

- ibmswcsp.dll
- ibmswcsp.h

The IBM Software Cryptographic Service Provider module provides cryptographic functionality. Table 3 lists the SCCS API functions supported by this module.

All functions that require input/output buffers support only one buffer at a time and not a vector of buffers. If an application provides a buffer to the CSP module, it must also specify the buffer length. On return from an SCCS API function, the length field of an output buffer will be set to the length of returned data. If an output buffer's length is set to zero and its data pointer is set to NULL, the CSP will allocate the needed memory on the application behalf. It is the responsibility of the application to free this memory when done.

Note that for encrypt/decrypt operations using RC2 the effective bits attribute must be set using CSSM_UpdateContextAttributes.

Table 3. IBM Software Cryptographic Service Provider SCCS Functions

Function Name	Supported	Comments
CSSM_QuerySize	No	
CSSM_SignData CSSM_SignDataInit CSSM_SignDataUpdate CSSM_SignDataFinal	Yes	Algorithms supported: CSSM_ALGID_MD2WithRSA CSSM_ALGID_MD5WithRSA CSSM_ALGID_SHA1WithRSA CSSM_ALGID_SHA1WithDSA
CSSM_VerifyData CSSM_VerifyDataInit CSSM_VerifyDataUpdate CSSM_VerifyDataFinal	Yes	Algorithms supported: CSSM_ALGID_MD2WithRSA CSSM_ALGID_MD5WithRSA CSSM_ALGID_SHA1WithRSA CSSM_ALGID_SHA1WithDSA
CSSM_DigestData CSSM_DigestDataInit CSSM_DigestDataUpdate CSSM_DigestDataFinal	Yes	Algorithms Supported: CSSM_ALGID_MD2 CSSM_ALGID_MD5 CSSM_ALGID_SHA1
CSSM_DigestDataClone	No	
CSSM_GenerateMac	No	
CSSM_GenerateMacInit	No	
CSSM_GenerateMacUpdate	No	
CSSM_GenerateMacFinal	No	
CSSM_VerifyMac	No	
CSSM_VerifyMacInit	No	
CSSM_VerifyMacUpdate	No	
CSSM_VerifyMacFinal	No	

CSSM_EncryptData CSSM_EncryptDataInit CSSM_EncryptDataUpdate CSSM_EncryptDataFinal	Yes	See remarks below
CSSM_DecryptData CSSM_DecryptDataInit CSSM_DecryptDataUpdate CSSM_DecryptDataFinal	Yes	See remarks below
CSSM_QueryKeySizeInBits	Yes	
CSSM_GenerateKey	Yes	Algorithms/Modes Supported: CSSM_ALGID_DES CSSM_ALGID_3DES_3KEY CSSM_ALGID_RC2 CSSM_ALGID_RC4 CSSM_ALGID_RC5
CSSM_GenerateKeyPair	Yes	See remarks below
CSSM_GenerateRandom	Yes	Algorithms Supported: CSSM_ALGID_MD2Random CSSM_ALGID_MD5Random
CSSM_GenerateAlgorithmParams	Yes	See remarks below
CSSM_WrapKey	No	
CSSM_UnwrapKey	No	
CSSM_DeriveKey	Yes	See remarks below
CSSM_CSP_PassThrough	No	
CSSM_CSP_Login	No	
CSSM_CSP_Logout	No	
CSSM_CSP_ChangeLoginPassword	No	

CSSM_EncryptData
CSSM_EncryptDataInit
CSSM_EncryptDataUpdate
CSSM_EncryptDataFinal

Algorithms/Modes Supported:

Algorithm	*Mode*
CSSM_ALGID_RSA	----
CSSM_ALGID_RSA_PKCS	----
CSSM_ALGID_DES	CSSM_ALGMODE_CBCPadIV8
CSSM_ALGID_3DES_3KEY	CSSM_ALGMODE_CBCPadIV8
CSSM_ALGID_RC2	CSSM_ALGMODE_CBCPadIV8
CSSM_ALGID_RC4	CSSM_ALGMODE_NONE

CSSM_ALGID_RC5	CSSM_ALGMODE_CBCPadIV8

CSSM_DecryptData
CSSM_DecryptDataInit
CSSM_DecryptDataUpdate
CSSM_DecryptDataFinal

Algorithms/Modes Supported:

Algorithm	*Mode*
CSSM_ALGID_RSA	----
CSSM_ALGID_RSA_PKCS	----
CSSM_ALGID_DES	CSSM_ALGMODE_CBCPadIV8
CSSM_ALGID_3DES_3KEY	CSSM_ALGMODE_CBCPadIV8
CSSM_ALGID_RC2	CSSM_ALGMODE_CBCPadIV8
CSSM_ALGID_RC4	CSSM_ALGMODE_NONE
CSSM_ALGID_RC5	CSSM_ALGMODE_CBCPadIV8

CSSM_GenerateKeyPair

Algorithms Supported:

CSSM_ALGID_RSA, CSSM_ALGID_DSA, CSSM_ALGID_DSA_BSAFE, CSSM_ALGID_DH

Note: For CSSM_ALGID_DH, the public key contains the public part to be exchanged with the other side. The private key contains a temporary handle that is valid only during the attach session. The private key and the other side's public key will be input to the CSSM_DeriveKey to derive the agreed upon symmetric key.

CSSM_GenerateAlgorithmParams
This function must be called with a KEYGEN context with the *Params* input of the CSSM_CSP_CreateKeyGenContext set to NULL. The output *Param* of this function will then be passed to another CSSM_CSP_CreateKeyGenContext to generate the Diffie Hellman key pair.

Algorithms Supported:

CSSM_ALGID_DH

CSSM_DeriveKey
The *BaseKey* parameter should be set to the private key returned from the CSSM_GenerateKeyPair function. *Param* should be set to the public key received from the other side of the key exchange operation.

Algorithms Supported:

CSSM_ALGID_DH

8.6.2 IBM PKCS11 Multi-Service Module, Version 1.0

Files required:

- pkcsmsm.dll
- pkcs11msm.h

PKCS11 MSM is a multi service provider supporting cryptographic and data storage operations on PKCS11 V1.X tokens. Table 4 lists the SCCS API functions that this module supports. All functions requiring input/output buffers support only one buffer at a time and not a vector of buffers. All algorithms specified in the PKCS11 1.X spec are supported. However some algorithms may not be supported by individual tokens. Note that for encrypt/decrypt operations using:

- RC2 the effective bits attribute must be set using CSSM_UpdateContextAttributes.
- RC4 the maximum length for input data is 900 bytes.

Table 4. IBM PKCS11 Multi-Service Module SCCS Functions

Cryptographic Library Functions		
Function Name	**Supported**	**Comments**
CSSM_QuerySize	No	
CSSM_SignData	Yes	See remarks below
CSSM_SignDataInit	No	
CSSM_SignDataUpdate	No	
CSSM_SignDataFinal	No	
CSSM_VerifyData	Yes	See remarks below
CSSM_VerifyDataInit	No	
CSSM_VerifyDataUpdate	No	
CSSM_VerifyDataFinal	No	
CSSM_DigestData CSSM_DigestDataInit CSSM_DigestDataUpdate CSSM_DigestDataFinal	Yes	Algorithm(s) supported: CSSM_ALGID_MD2, CSSM_ALGID_MD5, CSSM_ALGID_SHA1
CSSM_DigestDataClone	No	
CSSM_GenerateMac CSSM_GenerateMacInit CSSM_GenerateMacUpdate CSSM_GenerateMacFinal	Yes	Algorithm(s) supported: CSSM_ALGID_RC2, CSSM_ALGID_DES, CSSM_ALGID_3DES_3KEY, CSSM_ALGID_3DES_2KEY
CSSM_VerifyMac CSSM_VerifyMacInit CSSM_VerifyMacUpdate CSSM_VerifyMacFinal	Yes	Algorithm(s) supported: CSSM_ALGID_RC2, CSSM_ALGID_DES, CSSM_ALGID_3DES_3KEY, CSSM_ALGID_3DES_2KEY

CSSM_EncryptData CSSM_EncryptDataInit CSSM_EncryptDataUpdate CSSM_EncryptDataFinal	Yes	See remarks below
CSSM_DecryptData	Yes	See remarks below
CSSM_DecryptDataInit CSSM_DecryptDataUpdate CSSM_DecryptDataFinal	Yes	See remarks below
CSSM_QueryKeySizeInBits	Yes	
CSSM_GenerateKey	Yes	See remarks below
CSSM_GenerateKeyPair	Yes	See remarks below
CSSM_GenerateRandom	Yes	Algorithm(s) supported: CSSM_ALGID_PKCS11Random
CSSM_GenerateAlgorithmParams	No	
CSSM_WrapKey CSSM_UnwrapKey	Yes	See remarks below
CSSM_DeriveKey	Yes	Algorithm(s) supported: CSSM_ALGID_DH
CSSM_CSP_PassThrough	No	
CSSM_CSP_Login	Yes	Not recommended. Instead use CSSM_DL_DbOpen to retrieve the DB handle needed for all DL operations.
CSSM_CSP_Logout	Yes	Not recommended. Instead use CSSM_DL_DbClose.
CSSM_CSP_ChangeLoginPassword	Yes	
Data Store Library Functions		
Function Name	**Supported**	**Comments**
CSSM_DL_Authenticate	No	
CSSM_DL_DbOpen	Yes	
CSSM_DL_DbClose	Yes	
CSSM_DL_DbCreate	No	
CSSM_DL_DbDelete	Yes	
CSSM_DL_DbImport	No	
CSSM_DL_DbExport	No	
CSSM_DL_DbSetRecordParsingFunctions	No	
CSSM_DL_DbGetRecordParsingFunctions	No	
CSSM_DL_GetDbNames	No	

CSSM_DL_GetDbNameFromHandle	No	
CSSM_DL_FreeNameList	No	
CSSM_DL_DataInsert	Yes	See remarks below.
CSSM_DL_DataDelete	Yes	
CSSM_DL_DataGetFirst	Yes	See remarks below.
CSSM_DL_DataGetNext	Yes	See remarks below.
CSSM_DL_FreeUniqueRecord	No	
CSSM_DL_AbortQuery	No	
CSSM_DL_PassThrough	No	

CSSM_SignData
Only single staged signing supported with asymmetric/ public key algorithms. Essentially used to sign digests. Two step signing (digest then sign) not supported with PKCS11 1.X tokens.

Algorithms Supported:
CSSM_ALGID_RSA_PKCS, CSSM_ALGID_RSA_ISO9796, CSSM_ALGID_RSA_RAW, CSSM_ALGID_DSA

CSSM_VerifyData
Only single staged verifying supported with public key algorithms. Essentially used to sign digests. Two step verifying (digest then verify) is not supported with PKCS11 1.X tokens.

Algorithms Supported:
CSSM_ALGID_RSA_PKCS, CSSM_ALGID_RSA_ISO9796, CSSM_ALGID_RSA_RAW, CSSM_ALGID_DSA

CSSM_EncryptData
Asymmetric algorithms are supported only in single stage mode. This implies the data has to be less than the modulus size in bytes minus eleven to be PKCS compliant. Some tokens may not support encryption with asymmetric algorithms.

Algorithms/Modes Supported:
CSSM_ALGID_RSA_PKCS
CSSM_ALGID_RSA_RAW

Algorithm	*Mode*
CSSM_ALGID_DES	CSSM_ALGMODE_ECB, CSSM_ALGMODE_CBC
CSSM_ALGID_3DES_2KEY	CSSM_ALGMODE_ECB, CSSM_ALGMODE_CBC
CSSM_ALGID_3DES_3KEY	CSSM_ALGMODE_ECB, CSSM_ALGMODE_CBC
CSSM_ALGID_RC2	CSSM_ALGMODE_ECB, CSSM_ALGMODE_CBC
CSSM_ALGID_RC4	CSSM_ALGMODE_NONE

CSSM_EncryptDataInit
CSSM_EncryptDataUpdate
CSSM_EncryptDataFinal
Asymmetric algorithms are supported only in single stage mode. This implies the data has to be less than the modulus size in bytes minus eleven to be PKCS compliant. Some tokens may not support encryption with asymmetric algorithms.

Algorithms/Modes Supported:

Algorithm	*Mode*
CSSM_ALGID_DES	CSSM_ALGMODE_ECB, CSSM_ALGMODE_CBC
CSSM_ALGID_3DES_2KEY	CSSM_ALGMODE_ECB, CSSM_ALGMODE_CBC
CSSM_ALGID_3DES_3KEY	CSSM_ALGMODE_ECB, CSSM_ALGMODE_CBC
CSSM_ALGID_RC2	CSSM_ALGMODE_ECB, CSSM_ALGMODE_CBC
CSSM_ALGID_RC4	CSSM_ALGMODE_NONE

CSSM_DecryptData
Asymmetric algorithms are supported only in single stage mode. This implies the data has to be equal to the modulus size in bytes to be PKCS compliant. Some tokens may not support decryption with asymmetric algorithms.

Algorithms/Mode Supported:

Algorithm	*Mode*
CSSM_ALGID_RSA_PKCS	----
CSSM_ALGID_RSA_RAW	----
CSSM_ALGID_DES	CSSM_ALGMODE_ECB, CSSM_ALGMODE_CBC
CSSM_ALGID_3DES_2KEY	CSSM_ALGMODE_ECB, CSSM_ALGMODE_CBC
CSSM_ALGID_3DES_3KEY	CSSM_ALGMODE_ECB, CSSM_ALGMODE_CBC
CSSM_ALGID_RC2	CSSM_ALGMODE_ECB, CSSM_ALGMODE_CBC
CSSM_ALGID_RC4	CSSM_ALGMODE_NONE

CSSM_DecryptDataInit
CSSM_DecryptDataUpdate
CSSM_DecryptDataFinal
Asymmetric algorithms are supported only in single stage mode. This implies the data has to be equal to the modulus size in bytes to be PKCS compliant. Some tokens may not support decryption with asymmetric algorithms.

Algorithms/Modes Supported:

Algorithm	*Mode*
CSSM_ALGID_DES	CSSM_ALGMODE_ECB, CSSM_ALGMODE_CBC
CSSM_ALGID_3DES_2KEY	CSSM_ALGMODE_ECB, CSSM_ALGMODE_CBC
CSSM_ALGID_3DES_3KEY	CSSM_ALGMODE_ECB, CSSM_ALGMODE_CBC
CSSM_ALGID_RC2	CSSM_ALGMODE_ECB, CSSM_ALGMODE_CBC
CSSM_ALGID_RC4	CSSM_ALGMODE_NONE

CSSM_GenerateKey
CSSM_GenerateKeyPair
The key returned will be of type CSSM_KEYBLOB_REFERENCE and format CSSM_KEYBLOB_REF_FORMAT_INTEGER. The key data contains a handle to the PKCS11 object. The returned key even when it is permanent is only valid during the module attach session. To use key at a later time it must be searched for using CSSM_DL_DataGetFirst and a set of attributes such as key label and key type.

Algorithms Supported (CSSM_Generate_Key):

CSSM_ALGID_RC2, CSSM_ALGID_DES, CSSM_ALGID_3DES_3KEY, CSSM_ALGID_3DES_2KEY

Algorithms Supported (CSSM_GenerateKeyPair):

CSSM_ALGID_RSA_PKCS, CSSM_ALGID_DSA, CSSM_ALGID_DH

CSSM_WrapKey
CSSM_UnwrapKey
Symmetric keys can be wrapped or unwrapped using either another symmetric key or a public key. Some tokens may not allow wrapping/unwrapping of private keys.

Algorithms/Modes Supported:

Algorithm	*Mode*
CSSM_ALGID_RSA_PKCS	----
CSSM_ALGID_RSA_RAW	----
CSSM_ALGID_DES	CSSM_ALGMODE_ECB
CSSM_ALGID_3DES_2KEY	CSSM_ALGMODE_ECB
CSSM_ALGID_3DES_3KEY	CSSM_ALGMODE_ECB
CSSM_ALGID_RC2	CSSM_ALGMODE_ECB

CSSM_DL_DataInsert
If the Attributes parameter is not NULL and the Data parameter is NULL this function is used to create a PKCS11 style object. If Attributes is NULL and Data is not NULL, Data.Data points to a symmetric key or public key to be inserted into the token. The inserted key must be of type CSSM_KEYBLOB_RAW and format CSSM_KEYBLOB_RAW_FORMAT_CDSA. The return unique record will points to a key in PKCS11 format that can be subsequently used in crypto operations.

CSSM_DL_DataGetFirst
CSSM_DL_DataGetNext
If the Query parameter is NULL, this function will return the first object found. And subsequent CSSM_DL_DataGetNext will return the next object found until there is no object left. The Attributes parameter will point to a list of all attributes belonging to the object except for any sensitive attributes. The Data parameter will point to a PKCS11 format key if the object is a key.

If the Query is not NULL it needs to point to a list of PKCS11 attributes to be search for. Most often the attributes will be a label and/or key type to find a key. The QueryFlags indicate whether the return object/key should be in PKCS11 format to be used in crypto operations or in SCCS format for exporting of public keys. Symmetric keys and private keys when returned in SCCS format will contain a NULL KeyData pointer if the keys are marked sensitive.

8.6.3 IBM CCA Multi-Service Module, Version 1.0

File required:

- libccacsp.a

The IBM CCA MSM provides cryptographic capabilities and cryptographic data storage capabilities to CDSA applications running on AIX version 4.1.3, 4.1.4, 4.1.5, or 4.2. Table 5 lists the SCCS API functions that this module supports. The IBM CCA MSM relies on underlying CCA hardware to provide its services. It currently supports the following capabilities using the IBM 4758 card:

- data digesting using MD5 and SHA-1 hashing algorithms (CSSM_ALGID_MD5 and CSSM_ALGID_SHA1)

- generation of random numbers

- DES encryption/decryption algorithm (CSSM_ALGID_DES). The following encryption/decryption modes (one of which must be explicitly included into the correspondent cryptographic context) are supported:

  CSSM_ALGMODE_CBC.

  CSSM_ALGMODE_CBC_IV8.

  CSSM_ALGMODE_CBCPadIV8.

  NOTE. If CSSM_ALGMODE_CBC or CSSM_ALGMODE_CBC_IV8 is used during encryption, the length of the data must be an integral multiple of 8 bytes.

- storage of DES keys

- wrapping of keys (both DES and RSA) using single or double-length DES keys algorithms (CSSM_ALGID_DES and CSSM_ALGID_3DES-2KEY).

- RSA key-pairs up to 1024 bits long for the following operations

  signature/verification

  DES key exchange

The following RSA family algorithms are supported:

  CSSM_ALGID_RSA_PKCS

  CSSM_ALGID_RSA_ISO9796

- storage of RSA key-pairs

- deletion of DES keys from the data storage

- data encryption/decryption using RSA OAEP algorithm (part of SET protocol) – CSSM_ALGID_WrapSET_OAEP. There is an optional encryption hashing mode supported for this

algorithm: CSSM_ALGMODE_OAEP_HASH. If the mode is not specified, encryption using default (non-hashing) mode is performed.

General notes:

Multiple buffers are not supported during encryption and decryption operations (although encryption/decryption using RSA OAEP algorithm makes use of two buffers, these buffers have a different significance than that described in the generic API document – see description of the CSSM_EncryptData() and CSSM_DecryptData() functions below).

The labels for the keys being stored are generated by the module and an application does not have any control over these labels. Therefore the input label parameters are disregarded.

Only one anonymous data base, signified by setting names to NULL, is supported at all times. Therefore parameters such as database name, etc. are disregarded.

If a function expects a CSSM_DATA structure as a parameter describing the output, and the Length element is zero and Data element is NULL, then the necessary memory will be allocated by the function.

The DL functions support DB records of CSSM_DB_RECORD_TYPE of

CSSM_DL_DB_RECORD_KEY

CSSM_DL_DB_RECORD_PUBLIC_KEY

CSSM_DL_DB_RECORD_PRIVATE_KEY

The only attribute supported by DL functions is CSSM_DL_ATTRIBUTE_KEY_TYPE. This attribute can have one of the following values: CSSM_ATTRIBUTE_KEYTYPE_RSA, CSSM_KEYTYPE_ATTRIBUTE_DES2, or CSSM_ATTRIBUTE_KEYTYPE_DES.

In a CSSM_DB_UNIQUE_RECORD structure, the *Data* element of *RecordDataValue* is presumed to point to a CSSM_KEY structure. Therefore, an application must ensure that the *Data* portion of any supplied CSSM_DB_UNIQUE_RECORD points to a valid CSSM_KEY structure. This also implies that if an application provides a CSSM_DB_UNIQUE_RECORD structure as an output parameter, then at the end of a DL function execution the *Data* portion of *RecordDataValue* element will point to a valid CSSM_KEY structure.

Table 5. IBM CCA Multi-Service Module SCCS Functions

Cryptographic Library Functions		
Function Name	**Supported**	**Comments**
CSSM_QuerySize	Yes	See remarks below
CSSM_SignData	Yes	
CSSM_SignDataInit	Yes	
CSSM_SignDataUpdate	Yes	
CSSM_SignDataFinal	Yes	
CSSM_VerifyData	Yes	See remarks below

CSSM_VerifyDataInit	Yes	See remarks below
CSSM_VerifyDataUpdate	Yes	
CSSM_VerifyDataFinal	Yes	
CSSM_DigestData CSSM_DigestDataInit CSSM_DigestDataUpdate CSSM_DigestDataFinal	Yes	Algorithm(s) supported: CSSM_ALGID_MD5, CSSM_ALGID_SHA1
CSSM_DigestDataClone	Yes	
CSSM_GenerateMac CSSM_GenerateMacInit CSSM_GenerateMacUpdate CSSM_GenerateMacFinal	Yes	Algorithm(s) supported: CSSM_ALGID_DES
CSSM_VerifyMac CSSM_VerifyMacInit CSSM_VerifyMacUpdate CSSM_VerifyMacFinal	Yes	Algorithm(s) supported: CSSM_ALGID_DES
CSSM_EncryptData CSSM_EncryptDataInit CSSM_EncryptDataUpdate CSSM_EncryptDataFinal	Yes	See remarks below
CSSM_DecryptData	Yes	See remarks below
CSSM_DecryptDataInit CSSM_DecryptDataUpdate CSSM_DecryptDataFinal	Yes	See remarks below
CSSM_QueryKeySizeInBits	Yes	
CSSM_GenerateKey	Yes	See remarks below
CSSM_GenerateKeyPair	Yes	
CSSM_GenerateRandom	Yes	
CSSM_GenerateAlgorithmParams	No	
CSSM_WrapKey CSSM_UnwrapKey	Yes	See remarks below
CSSM_DeriveKey	Yes	
CSSM_CSP_PassThrough	No	
CSSM_CSP_Login	Yes	See remarks below
CSSM_CSP_Logout	Yes	
CSSM_CSP_ChangeLoginPassword	No	
Data Store Library Functions		
Function Name	**Supported**	**Comments**
CSSM_DL_Authenticate	Yes	See remarks below

CSSM_DL_DbOpen	Yes	
CSSM_DL_DbClose	Yes	
CSSM_DL_DbCreate	Yes	
CSSM_DL_DbDelete	No	
CSSM_DL_DbImport	No	
CSSM_DL_DbExport	No	
CSSM_DL_DbSetRecordParsingFunctions	No	
CSSM_DL_DbGetRecordParsingFunctions	No	
CSSM_DL_GetDbNames	No	
CSSM_DL_GetDbNameFromHandle	No	
CSSM_DL_FreeNameList	No	
CSSM_DL_DataInsert	Yes	
CSSM_DL_DataDelete	Yes	
CSSM_DL_DataGetFirst	Yes	See remarks below.
CSSM_DL_DataGetNext	Yes	See remarks below.
CSSM_DL_FreeUniqueRecord	No	
CSSM_DL_AbortQuery	Yes	
CSSM_DL_PassThrough	No	

CSSM_CSP_Login
An application is expected to set *Param->Data* field of the input *Password* parameter to point to a filled out CCA_LOGIN_PARAMETERS structure prior to calling this function (please refer to ibmcca.h file).

CSSM_DecryptData
Multiple input/output buffers are not supported in the general sense.

NOTE 1. Asymmetric decryption using RSA OAEP algorithm is supported. The significance of the parameters in this case is as follows (for more information please see the SET specification):

- The ClearBufCount and CipherBufCount parameters should both equal 2
- The first (index 0) CipherBuf contains the OAEP block
- The second (index 1) CipherBuf contains the encrypted data (NOTE. Output CipherBuf buffers from CSSM_EncryptData() my be supplied without any modifications as parameters for CSSM_DecryptData())
- After decryption BC byte will be stored at the offset 0 of the first (index 0) ClearBuf buffer, and XDATA will be stored in the same buffer starting at the offset of 1 byte.

- The decrypted data will be stored in the second (index 1) ClearBuf buffer

(NOTE. Because of the specifics of the SET implementation the length returned for the first (index 0) ClearBuf is always going to be 95 regardless of the actual size of the XDATA supplied during the encryption. It is therefore recommended that an application initialized this buffer with zeros before comparing it with the XDATA supplied as input for CSSM_EncryptData().

(see also CSSM_EncryptData() function description).

NOTE 2. In addition to standard decryption, symmetric decryption using clear single length (8 bytes) DES key is supported. The DES key has to have been inserted into the decryption context as the CSSM_ATTRIBUTE_KEY attribute. The *BlobType* element of the key header needs to be set to CSSM_KEYBLOB_RAW.

CSSM_DecryptDataInit
NOTE. Symmetric decryption using clear single length (8 bytes) DES key is supported. The DES key has to have been inserted into the decryption context as the CSSM_ATTRIBUTE_KEY attribute. The *BlobType* element of the key header needs to be set to CSSM_KEYBLOB_RAW.

CSSM_DecryptDataUpdate
Multiple input and output buffers are not supported.

CSSM_DecryptDataFinal
Multiple input and output buffers are not supported.

CSSM_EncryptData
Multiple input and output buffers are not supported.

NOTE. Asymmetric encryption using RSA OAEP algorithm is supported. The significance of the parameters in this case is as follows (for more information please see the SET specification):

- The ClearBufCount and CipherBufCount parameters should both equal 2
- The first (index 0) ClearBuf buffer should contain BC byte at the offset 0, and XDATA starting at the offset of 1.
- The second (index 1) ClearBuf buffer should contain the data to be encrypted.
- The OAEP block will be stored in the first (index 0) CipherBuf
- The encrypted data will be stored in the second (index 1) CipherBuf

(see also CSSM_DecryptData() function description).

NOTE 2. In addition to standard encryption, symmetric encryption using clear single length (8 bytes) DES key is supported. The DES key has to have been inserted into the encryption context as the CSSM_ATTRIBUTE_KEY attribute. The *BlobType* element of the key header needs to be set to CSSM_KEYBLOB_RAW.

CSSM_EncryptDataInit
NOTE. In addition to standard encryption, symmetric encryption using clear single length (8 bytes) DES key is supported. The DES key has to have been inserted into the encryption context as the CSSM_ATTRIBUTE_KEY attribute. The *BlobType* element of the key header needs to be set to CSSM_KEYBLOB_RAW.

CSSM_EncryptDataUpdate
Multiple input and output buffers are not supported.

CSSM_EncryptDataFinal
Multiple input and output buffers are not supported.

CSSM_GenerateKey
In addition to generating regular DES keys (keys with CSSM_KEYUSE_ENCRYPT and CSSM_KEYUSE_DECRYPT key usage properties), this function can be used to generate keys to be used as wrapping keys during DES key exchange (keys with CSSM_KEYUSE_WRAP and/or CSSM_KEYUSE_UNWRAP key usage properties). These options are mutually exclusive, since simultaneous usage of CSSM_KEYUSE_ENCRYPT or CSSM_KEYUSE_DECRYPT with CSSM_KEYUSE_WRAP or CSSM_KEYUSE_UNWRAP is not supported.

CSSM_QuerySize
In addition to the conventional usage, this function may be used in order to find out the sizes of the necessary output buffers for the RSA OAEP encryption/decryption. In order to this, an application must set the ContextType field of the Context parameter to CSSM_ALGCLASS_ASYMMETRIC. The function will expect the following input parameters:

- DataBlock should be an array of 2 CSSM_QUERY_SIZE_DATA structures

The following values are expected in these structures on input and stored there on output:

if Encrypt parameter equals CSSM_TRUE:

	Input	Output
block 1	Size of plaintext data	size of encrypted data
block 2	Size of XDATA	size of OAEP block

if Encrypt parameter equals CSSM_FALSE:

	Input	Output

block 1	Size of encrypted data	size of decrypted data
block 2	Size of OAEP block	size of XDATA

CSSM_UnwrapKey
NOTE 1. In addition to standard semantics, if the key to be unwrapped is a previously wrapped RSA public key (see CSSM_WrapKey() function description), it is imported into the module's internal format to facilitate RSA public key exchange between cryptographic nodes.

NOTE 2. An application can also import a clear RSA public key or DES single length key into the module's internal format. In order to do this, it needs to create an appropriate CSSM_KEY structure and supply it as WrappedKey parameter. The *BlobType* element of the key header need to be set to CSSM_KEYBLOB_RAW for both DES and RSA clear keys. Additionally, for clear RSA public keys the *Format* element of the key header has to be as shown below:

8.6.4 Keyblob Format	**8.6.5 KeyData.Data points to**
CSSM_KEYBLOB_RAW_FORMAT_CDSA	CSSM_RSA_PUBLIC structure
CSSM_KEYBLOB_RAW_FORMAT_CCA	Structure containing an RSA public key stored in CCA internal format

CSSM_VerifyData
CSSM_VerifyDataInit
NOTE. In addition to standard verification, verification of a RSA signature using clear RSA key is supported. The RSA key has to have been inserted into the encryption context as the CSSM_ATTRIBUTE_KEY attribute. The *BlobType* element of the key header needs to be set to CSSM_KEYBLOB_RAW, and the *Format* element of the key header has to be as shown in CSSM_UnwrapKey above.

CSSM_WrapKey
In addition to standard semantics, if the key to be wrapped is an RSA public key, it is exported "in the clear" to facilitate RSA public key exchange between cryptographic nodes. (see also CSSM_UnwrapKey() function description).

CSSM_DL_Authenticate
NOTE. By the time this function is called, an application is expected to fill the following buffers. *Credential->Data* element of UserAuthentication CSSM_USER_AUTHENTICATION parameter is expected to point to a buffer containing the login CCA password; the password's length should be stored in *Credential->Length* element. The user ID needed for login should be stored in the buffer pointed to by *MoreAuthenticationData->Param->Data* element of UserAuthentication structure; its length should be stored in *MoreAuthenticationData->Param->Length*.

CSSM_DL_DataGetFirst
NOTE. It has been pointed out already that on return from this function the *Data* portion of *RecordDataValue* element of the return CSSM_DB_UNIQUE_RECORD structure will point to a valid

CSSM_KEY structure. If the retrieval of the RSA public keys has been requested (e.g. the *QueryFlags* element of the Query parameter equals CSSM_QUERY_RETURN_DATA) then the *KeyData* element of this CSSM_KEY structure will point to a CSSM_RSA_PUBLIC structure.

CSSM_DL_DataGetNext

NOTE. It has been pointed out already that on return from this function the *Data* portion of *RecordDataValue* element of the return CSSM_DB_UNIQUE_RECORD structure will point to a valid CSSM_KEY structure. If the retrieval of the RSA public keys has been requested (e.g. the *QueryFlags* element of the Query parameter supplied at the time of CSSM_DL_DataGetFirst() function call equals CSSM_QUERY_RETURN_DATA) then the *KeyData* element of this CSSM_KEY structure will point to a CSSM_RSA_PUBLIC structure.

8.6.4 IBM Standard Trust Policy Library, Version 1.0

Files required:
- ibmtp.dll
- ibmtp.h

The IBM Standard Trust Policy module (IBMTP) provides a simple generic service for verifying chains of X509 certificates. The current version does not support operations that require DL operations.

This module expects X509v3 signed certificates in DER encoded format. In order to verify a given certificate, the application should supply the complete chain. Table 6 lists the functions supported by this module.

Table 6. IBM Standard Trust Policy Library SCCS Functions

Function	Supported	Comments
CSSM_TP_CertSign	No	
CSSM_TP_CertRevoke	No	
CSSM_TP_CrlSign	No	
CSSM_TP_CrlVerify	No	
CSSM_TP_ApplyCrlToDb	No	
CSSM_TP_CertGroupConstruct	No	
CSSM_TP_CertGroupPrune	No	
CSSM_TP_CertGroupVerify	Yes	See remarks below
CSSM_TP_PassThrough	No	

CSSM_TP_CertGroupVerify
The application should supply one anchor certificate, and an *ordered* chain of certificates in the CertToBeVerified argument.

The following function arguments are ignored in this version: Evidence, EvidenceSize, Action, policyIdentifers, NumberOfPolicyIdentifiers, VerificationAbortOn, VerifyScope, ScopeSize, DBList, Data.

Error codes returned:
- CSSM_TP_INVALID_TP_HANDLE: TPHandle argument is 0.
- CSSM_TP_INVALID_CL_HANDLE: CLHandle argument is 0.
- CSSM_TP_INVALID_CSP_HANDLE: CSPHandle argument is 0.
- CSSM_TP_INVALID_DATA_POINTER: CertToBeVerified argument is NULL or invalid. This argument is invalid if the length is set to 0, or the pointer to data is NULL.
- CSSM_TP_INVALID_CC_HANDLE: This error occurs if TP is unable to create a cryptographic context using the supplied CSPHandle and the certificates.
- CSSM_TP_ANCHOR_NOT_SELF_SIGNED: The supplied anchor certificate is not self signed
- CSSM_TP_ANCHOR_NOT_FOUND: The supplied anchor certificate is not the anchor for any of the certificates in the supplied chain.

- CSSM_TP_CERT_VERIFY_FAIL: The supplied certificate chain can not be verified

8.6.5 IBM Extended Trust Policy Library, Version 1.0

Files Required:
- ibmtp2.dll
- ibmtp2.h

Additional Requirements:
- LDAP product
- IBM CSP and IBM DL modules

The Extended Trust Policy Library validates X.509v3 certificates and CRLs using two types of trust policies: Entrust and X.509. The module can accept the complete certificate chain or an incomplete certificate chain. If the module is passed an incomplete chain, it will attempt to fill-in the missing certificates by searching the associated data store. Table 7 lists the SCCS API functions that this module supports.

This module ignores the following arguments in all TP API.

```
const CSSM_FIELD_PTR Scope,
uint32 ScopeSize
```

Table 7. IBM Extended Trust Policy Library SCCS Functions

Function	Supported	Comments
CSSM_TP_CertSign	Yes	The argument pair (*SignScope* , *ScopeSize*) is ignored. This function takes the input *CertToBeSigned* as an unsigned X509 certificate and signs it entirely.
CSSM_TP_CertRevoke	Yes	The *Reason* argument is ignored.
CSSM_TP_CrlSign	Yes	The argument pair (*SignScope*, *ScopeSize*) is ignored. This function takes the input *CrlToBeSigned* as an unsigned certificate revocation list and signs it entirely.
CSSM_TP_CrlVerify	Yes	
CSSM_TP_ApplyCrlToDb	Yes	
CSSM_TP_CertGroupConstruct	No	
CSSM_TP_CertGroupPrune	No	
CSSM_TP_CertGroupVerify	Yes	See remarks below
CSSM_TP_PassThrough	No	

CSSM_TP_CertGroupVerify
The parameter values passed to this function must be set as follows.

- The argument *PolicyIdentifiers* should be given as one of the four policies specified in ibmtp.h or queried from IBMTP_GUID by CSSM_GetModuleInfo. If zero or more than one policies are given, default policy (X509 certificate verification policy) is followed.
- The argument *VerificationAbortOn* is ignored.

- The argument *Action* is left for the caller to perform. This function verifies only the certificates.

8.6.6 IBM Certificate Library, Version 1.0

Files required:
- ibmcl.dll
- ibmcl.h

Additional files:
The IBM CL requires the OSS ASN1 runtime libraries (version R4.2.2). The following DLLs should be installed in the DLL path before IBM CL is attached.

- ossapi.dll
- ossdmem.dll
- cstrain.dll
- soedapi.dll
- soedber.dll

This module performs X.509v3 certificate operations. It provides a library of functions needed for creating, signing, verifying and querying a certificate. The current version does not support X.509v3 extensions. The IBM Certificate Library (IBMCL) expects X.509v3 signed certificates in DER encoded format. It uses a set of Object identifiers (OID) to exchange certificate information with the application. The list of supported OIDs is defined in file ibmcl.h, which should be included in every application that uses the services of IBMCL.

The following example demonstrates the purpose and use of OIDs. If an application asks for the version of a given certificate, the CL builds the version object that is returned to the application as follows:

```
CSSM_FIELD_PTR      p_version;

/* p_version is a pointer to a generic structure containing FieldOid and
   FieldValue. FieldOid contains a number that indicates the type of the
field,
   e.g. version, serial number, etc. FieldValue contains the actual data.
*/

/* allocate memory for p_version…*/

p_version->FieldOid.Length = sizeof(uint32);
.
./* allocate memory for OID */
.
p_version->FieldOid.Data = IBMCL_OID_VERSION;
p_version->FieldValue.Length =  Version.length; /* length of Version data */
Copy(Version.value, p_version->FieldValue.Data );
```

All fields are returned as unsigned character arrays, which in turn need to be cast to the appropriate type. The OID indicates the type of the field and the structure it should be cast to. The following example shows an instance where OID is used to build the relevant data structure:

```
CSSM_FIELD_PTR      p_field;
X500Name            *p_name;

/* call a CL function to obtain some field in the Cert */

switch ( *p_field->FieldOid.Data) {
    case IBMCL_OID_VERSION:
    break;
case IBMCL_OID_ISSUER_NAME:
```

```
    /* cast to the correct structure */
    p_name = (X500Name *) p_field->FieldValue.Data;
    break;
default:
    break;
}
```

The IBMCL functions in Table 8 comply with the SCCS API. Most of the functions return error codes that are specific to this implementation and not defined in the SCCS API. These error codes are defined in ibmcl.h and described below as part of supported API functions. Also, note that function arguments *Scope* and *ScopeSize* are ignored in this version. Moreover, in order to construct an X.500 name-only country name (C), organization name (O), organization name unit (OU), and common name (CN) are supported.

Table 8. IBM Certificate Library SCCS Functions

Function Name	Supported	Comments
CSSM_CL_CertSign	Yes	See remarks below
CSSM_CL_CertVerify	Yes	See remarks below
CSSM_CL_CertCreateTemplate	Yes	See remarks below
CSSM_CL_CertGetFirstFieldValue	Yes	See remarks below~~See remarks below~~
CSSM_CL_CertGetNextFieldValue	No	
CSSM_CL_CertAbortQuery	No	
CSSM_CL_CertGetKeyInfo	Yes	See remarks below~~This function returns the DER encoded subject public key. The encoding contains the public key, algorithm ID and parameters if applicable.~~
CSSM_CL_CertGetAllFields	Yes	See remarks below
CSSM_CL_CertImport	No	
CSSM_CL_CertExport	No	
CSSM_CL_CertDescribeFormat	Yes	
CSSM_CL_CrlCreateTemplate	No	
CSSM_CL_CrlSetFields	No	
CSSM_CL_CrlAddCert	No	
CSSM_CL_CrlRemoveCert	No	
CSSM_CL_CrlSign	No	
CSSM_CL_CrlVerify	No	
CSSM_CL_IsCertInCrl	No	
CSSM_CL_CrlGetFirstFieldValue	No	
CSSM_CL_CrlGetNextFieldValue	No	
CSSM_CL_CrlAbortQuery	No	

CSSM_CL_CrlDescribeFormat	No	
CSSM_CL_PassThrough	No	

CSSM_CL_CertCreateTemplate
This function accepts the public key field in two formats:

1. If the key algorithm requires any parameters, they can be put in the template with a separate OID. So, the application can pass in three OIDs and the respective values:

    - IBMCL_OID_SUBJECT_PUB_KEY: the value is passed in as a string. The key should not be DER encoded.

    - IBMCL_OID_PUB_KEY_PARAMETERS: Data should point to the DER encoding of the parameters.

    - IBMCL_OID_PUB_KEY_ALGID: Data indicates what algorithm ID is used for generating the key, e.g. CSSM_ALGID_RSA.

2. The algorithm ID, parameters and the key can be DER encoded and passed in with OID IBMCL_OID_SUBJECT_PUB_KEY. There is no need to supply the other two OIDs.

The template requires these fields: signature algorithm ID, validity, subject name, issuer name, and subject public key in one of the two formats described above. Validity is specified as an array of two CSSM_DATE elements. Index 0 should contain the start date and index 1 the end date of certificate validity.

This function returns the following error codes:

Error Code	*Description*
CSSM_CL_INVALID_CL_HANDLE	CLHandle argument passed in is invalid
CSSM_CL_INVALID_INPUT_PTR	CertTemplate argument passed in is NULL.
CSSM_CL_INVALID_DATA	NumberOfFields argument passed in is 0.
CSSM_CL_SIGN_ALGID_NOT_SUPPORTED	The supplied signature algorithm ID in the template is not supported by IBM CL.
CSSM_CL_INVALID_TEMPLATE	The given template is missing or contains an invalid pointer to one of these mandatory items: serial number, signature algorithm ID, validity, subject name, or subject public key. Also, if en extension or unique id is present in the template but the pointers are invalid, this error is returned.
CSSM_CL_INVALID_CERT_ISSUER_NAME	The supplied issuer name is invalid.
CSSM_CL_MISSING_CERT_ISSUER_NAME	The field for issuer name is not present in the template. This field is required for creating a valid certificate.
CSSM_CL_KEY_ALGID_NOT_SUPPORTED	The supplied algorithm ID for the subject publid key is not supported.

CSSM_CL_KEY_FORMAT_UNKNOWN	The supplied subject public key is not in the correct format.
CSSM_CL_CERT_CREATE_FAIL	Failed to DER encode the certificate. This error could be cause by invalid data in the template or memory problem.

CSSM_CL_CertGetAllFields

This function returns DER encoding of the unsigned part of the certificate, signature algorithm Id and parameters if applicable, and the signature (length in bytes). To view the specific fields in the certificate such as version or validity use CSSM_CL_GetFirstFieldValue with the appropriate OID. If the signature algorithm ID is not recognized by IBM CL, it is set to CSSM_ALGID_NONE. The other fields, however, are still returned to the application.

This function returns the following error codes:

Error Code	*Description*
CSSM_CL_INVALID_CL_HANDLE	CLHandle argument passed in is invalid
CSSM_CL_INVALID_CERT_POINTER	Cert argument passed in is NULL.
CSSM_CL_CERT_GET_FIELD_VALUE_FAIL	Unable to decode the certificate correctly.
CSSM_MALLOC_FAILED	Failed to allocate memory in the application space.

CSSM_CL_CertGetFirstFieldValue

The ResultHandle will always be set to NULL and the NumberOfMatchedFields will be set to 1 if any field is found, regardless of how many.

This function returns the following error codes:

Error Code	*Description*
CSSM_CL_INVALID_CL_HANDLE	CLHandle argument passed in is invalid
CSSM_CL_INVALID_CERT_POINTER	Cert argument passed in is NULL.
CSSM_CL_INVALID_INPUT_PTR	CertField or CertField->Data argument passed in is NULL.
CSSM_MALLOC_FAILED	Unable to allocate memory in the application space.
CSSM_CL_FIELD_NOT_PRESENT	The requested field is not in the certificate.
CSSM_CL_KEY_ALGID_NOT_SUPPORTED	If the key field is requested, the algorithm ID is not supported.

CSSM_CL_CertGetKeyInfo
This function returns the DER encoded subject public key. The encoding contains the public key, algorithm ID and parameters if applicable.

Error Code	*Description*
CSSM_CL_INVALID_CL_HANDLE	CLHandle argument passed in is invalid
CSSM_CL_INVALID_CERT_POINTER	Cert argument passed in is NULL.
CSSM_CL_CERT_GET_KEY_INFO_FAIL	Failed to decode the cert and obtain the public key.
CSSM_MALLOC_FAILED	Failed to allocate memory in the application memory space.
CSSM_CL_KEY_ALGID_NOT_SUPPORTED	The algorithm id of the subject public key is not supported.

CSSM_CL_CertSign
This function returns the following error codes:

Error Code	*Description*
CSSM_CL_INVALID_CL_HANDLE	CLHandle argument passed in is invalid
CSSM_CL_INVALID_CC_HANDLE	CCHandle argument passed in is invalid
CSSM_CL_INVALID_CERT_POINTER	CertToBeSigned or SignerCert arguments are invalid
CSSM_CL_INVALID_CONTEXT	Unable to obtain a valid context using the CCHandle passed in.
CSSM_CL_GET_KEY_ATTRIBUTE_FAIL	Unable to obtain a valid Key attribute using the CCHandle passed in.
CSSM_CL_KEY_ALGID_NOT_SUPPORTED	The specified algorithm ID in the signature context is not supported.
CSSM_CL_CERT_SIGN_FAIL	The signature operation failed. This could be caused by invalid attributes in the signature context.
CSSM_CL_CERT_ENCODE_FAIL	Failed to DER encode the signed certificate. This error could be caused by memory problems, or invalid context attributes.

CSSM_CL_CertVerify
This function returns the following error codes:

Error Code	*Description*
CSSM_CL_INVALID_CL_HANDLE	CLHandle argument passed in is invalid
CSSM_CL_INVALID_CC_HANDLE	CCHandle argument passed in is invalid
CSSM_CL_INVALID_CERT_POINTER	Either CertToBeVerified or SignerCert argument is NULL
CSSM_CL_CERT_VERIFY_FAIL	Failed to verify the signature on the certificate.
CSSM_CL_CERT_GET_FIELD_VALUE_FAIL	Failed to decode the CertToBeVerified correctly.
CSSM_MALLOC_FAILED	Failed to allocate memory.

8.6.7 IBM Data Library, Version 1.0

Files required:
- ibmdl2.dll
- ibmdl2.h

The Data Library provides support for the persistence and retrieval of security-related objects to/from a flat-file database maintained in the local file system. This module is semantic-free and allows the application developer to define the database record structure and index. Table 9 lists the SCCS API functions that this module supports.

All errors returned by this module are reported as CSSM_DL_PRIVATE_ERROR. If an error occurs within this module, it is possible to determine the exact cause of the error by enabling exception logging. The environment variable IBMFILEDL_LOG may be set to a file in which all exceptions are to be logged by this module. If an error occurs, it is possible to look in the specified file to get a object dump of the exception which will indicate the file and line number where the error occurred thus allowing the module developer to determine the exact cause of the failure.

Table 9. IBM Data Library SCCS Functions

Function Name	Supported	Comments
CSSM_DL_Authenticate	Yes	See remarks below
CSSM_DL_DbOpen	Yes	See remarks below
CSSM_DL_DbClose	Yes	The *DLHandle* parameter must not be NULL. The *DBHandle* parameter must reference an opened data store
CSSM_DL_DbCreate	Yes	See remarks below
CSSM_DL_DbDelete	Yes	See remarks below
CSSM_DL_DbImport	No	
CSSM_DL_DbExport	No	
CSSM_DL_DbSetRecordParsingFunctions	Yes	See remarks below
CSSM_DL_DbGetRecordParsingFunctions	Yes	The *DLHandle* parameter must not be NULL. The *DbName* specifies the absolute or relative path name to the file data store containing the record parsing functions. This parameter must not be NULL.
CSSM_DL_GetDbNameFromHandle	Yes	*DLHandle* parameter must not be NULL. *DBHandle* parameter must reference an opened data store
CSSM_DL_DataInsert	Yes	The *DLHandle*, *Attributes*, and *Data* parameters must not be NULL. The *DBHandle* parameter must reference an opened data store. The write access permissions flag must be true.

CSSM_DL_DataDelete	Yes	The *DLHandle* parameter must not be NULL. *DBHandle* must reference an opened data store. *UniqueRecordIdentifier* must not be NULL. The write access permissions flag must be true.
CSSM_DL_DataGetFirst	Yes	See remarks below
CSSM_DL_DataGetNext	Yes	See remarks below
CSSM_DL_FreeUniqueRecord	Yes	The *DLHandle* parameter must not be NULL. The *DBHandle* parameter is ignored.
CSSM_DL_AbortQuery	Yes	The *DLHandle* parameter must not be NULL. *DBHandle* must reference an opened data store. *ResultsHandle* must reference a valid query. The read access permissions flag must be true.
CSSM_DL_PassThrough	No	

CSSM_DL_Authenticate
The parameter values passed to this function must be set as follows.

- DLHandle must not be NULL
- DBHandle must reference an opened data store
- AccessRequest must not be NULL
- UserAuthentication must not be NULL
- UserAuthentication->Credential must not be NULL
- UserAuthentication->Credential->Length must not be NULL
- UserAuthentication->Credential->Data must not be NULL
- The password is to be passed in the Credential portion of the user authentication and is applied to the opened data store only if the password has changed.
- The access request flags are applied to the opened data store. Note that only read/write access flags are used in this module.

CSSM_DL_DbOpen
The parameter values passed to this function must be set as follows.

- DLHandle must not be NULL
- DbName must not be NULL
- AccessRequest must not be NULL
- UserAuthentication must not be NULL
- UserAuthentication->Credential must not be NULL
- UserAuthentication->Credential->Length must not be NULL
- UserAuthentication->Credential->Data must not be NULL
- UserAuthentication->MoreAuthenticationData is ignored
- OpenParameters is ignored.
- The DbName specifies the absolute or relative path name to the file data store to be opened
- The password is to be passed in the Credential portion of the user authentication

CSSM_DL_DbCreate
The parameter values passed to this function must be set as follows.

- DLHandle must not be NULL
- DbName must not be NULL
- DBInfo must not be NULL
- AccessRequest must not be NULL
- UserAuthentication must not be NULL
- UserAuthentication->Credential must not be NULL
- UserAuthentication->Credential->Length must not be NULL
- UserAuthentication->Credential->Data must not be NULL
- UserAuthentication->MoreAuthenticationData is ignored
- OpenParameters is ignored
- The DbName specifies the absolute or relative path name to the file data store to be created
- The password is to be passed in the Credential portion of the user authentication

CSSM_DL_DbDelete
The parameter values passed to this function must be set as follows.

- DLHandle must not be NULL
- DbName must not be NULL
- UserAuthentication must not be NULL
- UserAuthentication->Credential must not be NULL
- UserAuthentication->Credential->Length must not be NULL
- UserAuthentication->Credential->Data must not be NULL
- UserAuthentication->MoreAuthenticationData is ignored
- The DbName specifies the absolute or relative path name to the file data store to be deleted
- The password is to be passed in the Credential portion of the user authentication

CSSM_DL_DbSetRecordParsingFunctions
The parameter values passed to this function must be set as follows.

- DLHandle must not be NULL
- DbName must not be NULL
- FunctionTable must not be NULL
- FunctionTable->RecordGetFirstFieldValue must not be NULL
- FunctionTable->RecordGetNextFieldValue must not be NULL
- FunctionTable->RecordAbortQuery must not be NULL
- The DbName specifies the absolute or relative path name to the file data store to be have the record parsing functions manipulated.

CSSM_DL_DataGetFirst
The parameter values passed to this function must be set as follows.

- DLHandle must not be NULL
- DBHandle must reference an opened data store
- Query must not be NULL
- Query->Conjunctive must equal CSSM_DB_NONE
- Query->NumSelectionPredicates must be 0 or 1
- Query->SelectionPredicate must not be NULL if Query->NumSelectionPredicates is 1
- ResultsHandle must be an allocated pointer
- EndOfDataStore must be an allocated pointer

- Attributes must be an allocated pointer
- Data must be an allocated pointer
- The read access permissions flag must be true
- Query->NumSelectionPredicates equals 1 denotes an indexed query for a given record type
- Query->NumSelectionPredicates equals 0 denotes a sequential query for a given record type

CSSM_DL_DataGetNext
The parameter values passed to this function must be set as follows.

- DLHandle must not be NULL
- DBHandle must reference an opened data store
- ResultsHandle must reference a valid query
- EndOfDataStore must be an allocated pointer
- Attributes must be an allocated pointer
- Data must be an allocated pointer
- The read access permissions flag must be true

8.6.8 IBM Key Recovery Service Provider, Version 1.0

Files required:

- ibmskr.dll
- ibmskr.h

The IBM Key Recovery Service Provider generates and processes key recovery blocks according to open group standards.

Table 10. IBM Key Recovery Service Provider Module SCCS Functions

Function Name	Supported	Comments
CSSM_KR_GetPolicyInfo	Yes	
CSSM_KR_CreateRecoveryEnablementContext	Yes	See remarks below
CSSM_KR_CreateRecoveryRegistrationContext	No	
CSSM_KR_CreateRecoveryRequestContext	No	
CSSM_KR_SetEnterpriseRecoveryPolicy	Yes	
CSSM_KR_RegistrationRequest	No	
CSSM_KR_RegistrationRetrieve	No	
CSSM_KR_RecoveryRequest	No	
CSSM_KR_RecoveryRetrieve	No	
CSSM_KR_GetRecoveredObject	No	
CSSM_KR_RecoveryRequestAbort	No	
CSSM_KR_GenerateRecoveryFields	Yes	
CSSM_KR_ProcessRecoveryFields	Yes	
CSSM_KR_FreeKRProfile	Yes	
CSSM_KR_PassThrough	No	

CSSM_KR_CreateRecoveryEnablementContext

IBM KRSP Version 1.0 enables applications to be designed to request generation of key recovery blocks for three scenarios (Law Enforcement, Enterprise, and Individual). To request generation of key recovery blocks for any specific scenario, related KR-flags must be set and profile information supplied as outlined in the following table:

Scenario	Flag	Profile Information
Individual	KR_INDIV	Must be provided through API
Enterprise	KR_ENT	Can be provided through API If not provided, default profile information will be used
Law Enforcement	KR_LE | KR_LE_USE | KR_LE_MAN	Not accepted through API -- Must use default profile information

The profile information, provided through the KRACertChainList API parameter, must be structured as follows. The last member of a KRACertChainList must point to an Anchor certificate which must be self-signed. The member preceding the Anchor certificate must point to the Key Recovery Server Certificate, and it must be signed by the Anchor. Preceding the KRS certificate will be the required KRA certificates, each signed by the Anchor. The *number* parameter in the KRACertChainList must be set to two plus the required number of KRA's.

Chapter 9. Developing Security Applications

This chapter presents a high-level overview of the steps involved in modifying an existing application or protocol handler to incorporate the strong encryption and key recovery services provided by the IBM SCCS Toolkit and the IBM Key Recovery Service Provider. For an in-depth discussion of the SCCS Toolkit API calls necessary to perform strong encryption and key recovery, see the sample application presented in **Error! Reference source not found.**. The code for this sample application appears in Appendix A.

The application example discussed in the following sections shows the SCCS API calls that must be added to an application in order to enable it for key recovery and strong encryption. The application is assumed to use a client/server architecture and be statically linked to a BSAFE crypto library. In addition to the SCCS Toolkit and Key Recovery Service Provider module, both the client and server computers must have installed key recovery policy modules and configuration files.

The example demonstrates the client's process of creating key recovery fields and performing strong encryption, followed by the server validation of the key recovery fields and decryption of the message. The SCCS API calls for both the client and the server are listed in pseudocode, without proper arguments or other details. They are meant to give a general overview of the changes needed, rather than show sample code

The example assumes that the session key has been generated outside of the SCCS Framework, and the key exchange has already been performed. For the case in which the session key needs to be distributed by using the SCCS Framework, an example of Diffie-Hellman key exchange is provided in Section 9.1.

Client Application SCCS API calls

SCCS API Function	Description
Application Startup:	
CSSM_Init	Initialize the framework, and pass pointers to memory functions.
CSSM_ListModules(CSP)	Lists all installed cryptographic service providers (CSPs).
CSSM_GetModuleInfo	For each installed CSP, get information about the services it provides. Select one with all required services and...
CSSM_ModuleAttach(CSP)	...attach the CSP.
CSSM_ListModules(KRSP) CSSM_GetModuleInfo CSSM_ModuleAttach(KRSP)	Perform the same steps for the key recovery service providers (KRSP).
Strong Encryption:	
CSSM_CSP_CreateSymmetricContext	Specify all information relevant to performing symmetric encryption, including algorithm, mode, key, and initialization vector.
CSSM_KR_GetPolicyInfo	Inspects the context returned above and tells the application whether key recovery fields are required because the context specifies strong crypto. Assume key recovery fields are required.
CSSM_CreateRecoveryEnablementContext	Specify all information required to create the key recovery fields
CSSM_GenerateKRFields	Given the symmetric context and the key recovery context, creates the key recovery fields. Now The application can perform strong crypto.
CSSM_EncryptData	Using the parameters specified in the symmetric context, encrypts the message to the server.
Transmission Send (not done through framework)	Send the ciphertext and the key recovery fields to the server. Could be socket transmission, or any other protocol. This need not change from the way the application previously transmitted data.
Clean Up: CSSM_ModuleDetach(CSP) CSSM_ModuleDetach(KRSP)	Unload the crypto and key recovery service providers.

Server Application SCCS API calls

SCCS API Function	Description
App Startup:	Perform the same Startup steps as the client program.
Transmission Receive:	
(not done through framework)	Receive the ciphertext and key recovery fields from the client application.
Strong Decryption:	
CSSM_CSP_CreateSymmetricContext	Specify all information for symmetric decryption.
CSSM_KR_GetPolicyInfo	Inspects the context returned above and tells the application whether key recovery fields are required because the context specifies strong crypto. Since the server is performing decryption, it will validate the client-generated key recovery fields. Assume key recovery fields are required.
CSSM_CreateRecoveryEnablementContext	Specify all information required to validate the key recovery fields
CSSM_KR_ProcessRecoveryFields	Given the symmetric context and the key recovery context, as well as the client- generated key recovery fields, verifies the integrity of the key recovery fields sent from the client. Now The application can perform strong crypto (as decryption) using the algorithm parameters specified in the symmetric context. If this step fails, the application is not allowed to proceed.
CSSM_DecryptData	Decrypts the message from the client.
Clean Up:	Perform the usual SCCS cleanup.

9.1 Diffie-Hellman Key Exchange Scenario

This section outlines the procedure for performing Diffie-Hellman key exchange on both the client and the server machine. These steps are in addition to those described in the preceding section.

Client Application SCCS API calls

SCCS API Function	Description
App Startup:	Client performs normal startup procedure.
Key Exchange: CSSM_GenerateAlgorithmParameters	Specifies that you are generating Diffie-Hellman key exchange parameters
CSSM_CSP_CreateAsymmetricContext	Using the parameters generated above, create a context for key pair generation
CSSM_GenerateKeyPair	Create a Diffie-Hellman asymmetric key pair.
Transmission Send: (not performed by framework)	Send the public key to the server.
CSSM_CSP_CreateDeriveKeyContext	Specify the information required to derive a session key from the Diffie-Hellman key pair.
CSSM_DeriveKey	Derive the session key.
Strong Encryption:	Client performs encryption and clean-up operations previously described.

Server Application SCCS API calls

SCCS API Function	Description
App Startup:	Server performs normal startup procedure
Transmission Receive: (not performed by framework)	Receive the Diffie-Hellman public key from the client.
CSSM_CSP_CreateDeriveKeyContext	Specify the information required to derive a session key from the Diffie-Hellman key sent by the client.
CSSM_DeriveKey	Derive the session key.
Strong Decryption:	Server performs decryption and clean-up operations previously described.

Chapter 10. Sample Application

The kr_file_encrypt program is a sample program that shows how the SCCS API can be used to generate key recovery fields and encrypt a clear file. Even if the original password is lost, the resulting key recovery fields can be used to recover the key and the encrypted data. The kr_encrypt_file application demonstrates not only the details involved in generating key recovery fields and encrypting files, it illustrates the steps necessary to create any SCCS-based application. The steps are:

- Initialize the SCCS framework
- Attach the necessary service provider modules
- Perform the desired security operations
- Detach the modules when they are no longer needed

Source code for this program appears in Appendix A. The `kr_file_encrypt` program is written in the C language and can be run under either Microsoft Windows 95/NT or IBM AIX.

To run this program you must have installed on your system a Key Recovery service provider module and a Cryptographic service provider module that supports DES. If you have not already done so, you can install the IBM Key Recovery and Cryptographic modules by running the Setup programs for the SCCS Toolkit and the IBM Key Recovery Service Provider. You must also have access to a C compiler with the standard C library set and the Microsoft Visual C++ MSVCRT40.DLL runtime library. Once you have compiled the application you can run it from the command line by typing

```
C:\ kr_encrypt_file <filename>
```

where *filename* is a file that is 4096 bytes in size or less. kr_file_encrypt will encrypt the input file and generate two output files; the encrypted file (*filename*.enc) and a key recovery file (*filename*.krf).

10.1 Program Execution

This section presents an overview of the program execution. For detailed information on any of the SCCS API function calls or data structures, see the *Secure Cryptographic and Certificate Services Toolkit Application Programming Interface Specification* manual.

Program execution begins in main.c, which makes the following function calls:

ProcessArguments,
Initialize,
AttachCSPByAlgorithm,
AttachKRSPByUserChoice, and
GenerateKeyRecoveryFieldsAndEncrypt.

Each of these function calls is discussed in the following sections.

10.1.1 ProcessArguments

Located in file: main.c

This routine simply checks the input entered by the end user. If too many or too few parameters were entered, ProcessArguments displays a message informing the user of the correct command format and exits. Otherwise, the pointer ClearFilename is set to the input character array and returned to main.

10.1.2 Initialize

Located in file: initialize.c

This function demonstrates how to initialize the SCCS framework. First, the initialize function sets the Version data structure to the current version level. (CSSM_MAJOR and CSSM_MINOR are defined in cssmtype.h.)

Next, the MemoryFuncs data structure is initialized to the memory management function wrappers declared at the beginning of the initialize.c file. Since applications may have their own procedures for creating, managing, and freeing memory, the MemoryFuncs table is the way these functions can be made available to SCCS and the service provider modules. Applications register memory functions with SCCS using CSSM_Init and with the service provider modules using CSSM_ModuleAttach.

Both the Version and the MemoryFuncs data structures are passed to the CSSM_INIT function in the following statement.

```
CSSM_Init(&Version, &MemoryFuncs, NULL)
```

SCCS ensures the version information matches and stores a pointer to the MemoryFuncs table within the framework memory heap. This function should be called only once in any application.

10.1.3 AttachCSPByAlgorithm

Located in file: attach.c

There are various levels of detail that applications can use when attaching to modules using the SCCS API. In the simplest case, an application can hardcode a particular module ID, a GUID, so that it only works when a particular module is installed. A more flexible application can be designed to look into the installed list of modules and choose one based on some attribute it has, such as capability, vendor name, hardware/software, etc.

In AttachCSPByAlgorithm, the list of installed software cryptographic service providers is searched to find one that supports the required algorithm. The function accepts two input parameters, a pointer to the CSP handle and an unsigned integer indicating the type cryptographic algorithm desired, in this case CSSM_ALGID_DES. (The header file, cssmtype.h, defines the supported algorithms.)

The function first determines which cryptographic modules are currently installed by calling CSSM_ListModules.

```
pModuleList = CSSM_ListModules(CSSM_SERVICE_CSP, CSSM_TRUE)
```

This function generates a data structure of type CSSM_LIST and returns a pointer to that structure, pModuleList. The CSSM_LIST data structure contains a GUID/name pair for each of the currently installed modules that match the service mask for cryptographic modules, CSSM_SERVICE_CSP. If there are no CSP modules installed, the CSSM_LIST.NumberOfItems element contains a zero.

When a module is installed on a system, it must provide certain information about itself. This information is stored in series of data structures in the operating system registry facility. Module information is made available to SCCS applications through the CSSM_GetModuleInfo function call.

```
pModuleInfo = CSSM_GetModuleInfo(&(pModuleList->Items[i].GUID),
                                 CSSM_SERVICE_CSP,
                                 0,
                                 CSSM_INFO_LEVEL_ALL_ATTR);
```

CSSM_GetModuleInfo returns a pointer, pModuleInfo, to a data structure containing the module information. In the code that follows the CSSM_GetModuleInfo call, the system searches the module information retrieved for each module (using its GUID) for a match on CSSM_ALGID_DES. Once the appropriate module is found, CSSM_ModuleAttach is called which returns a handle to that module.

```
*hCSP = CSSM_ModuleAttach(&(pModuleList->Items[i-1].GUID), /*module GUID*/
                          &pModuleInfo->Version, /*version info*/
                          &MemoryFuncs, /*MemoryFuncs table*/
                          0,
                          0,
                          0,
                          NULL,
                          NULL);
```

SCCS uses module handles to match a calling application with the appropriate service module. Handles represent a one-to-one pairing between an application and a module. Multiple calls to CSSM_ModuleAttach are viewed as independent requests. Each attach request returns separate, independent handles that do not share execution state.

10.1.4 AttachKRSPByUserChoice

Located in file: attach.c

This function is similar to AttachCSPByAlgorithm with one notable exception. In AttachKRSPByUserChoice, the list of installed key recovery service providers is presented and the user is asked to select one. This function also calls CSSM_ListModules but displays the resulting list of Key Recovery modules the user and prompts them to select one.

10.1.5 GenerateKeyRecoveryFieldsAndEncrypt

Located in file: encrypt.c

GenerateKeyRecoveryFieldsAndEncrypt performs several operations. It generates a symmetric key for use in encrypting the input file, and also generates a context for use in the encryption process. However, since the kr_file_encrypt application is also performing key recovery, the key recovery fields for the newly created key are also generated and output as a data blob. Finally, the input file is encrypted and both the encrypted file and the key recovery fields are written to separate files. These operations are performed in the following subroutines:

- GenerateKey
- GenerateSymmetricContext
- GenerateKeyRecoveryFieldsForContext
- WriteOutputFile

GenerateKey

GenerateKey function creates a symmetric key. It does this by creating a security context, generating a symmetric key using information in the context, and then destroying the context. Security contexts perform two functions; to provide security for user-specific information and to package information for easy exchange between functions. Rather than declare, pass, and delete multiple parameters, contexts allow this information to be assembled into one temporary data structure. The type of context to be created depends upon the type of operation to be performed. Since the application requires a symmetric key it must create a key generation context. However, later in the program execution different types of contexts will be created to perform operations such as key recovery enablement.

GenerateKey first calls CSSM_CSP_CreateKeyGenContext and passes it the parameters to be used when creating the key and specifies, among other things, a key size of 64 bits and the desired encryption algorithm – DES.

```
hKeyGenContext = CSSM_CSP_CreateKeyGenContext(hCSP,
                                              CSSM_ALGID_DES,
                                              NULL,
                                              64,
                                              NULL,NULL,NULL,NULL,NULL);
```

GenerateKey next initializes the Key data structure, of type CSSM_KEY, to zero using the statement:

```
memset (Key, 0, sizeof(CSSM_KEY));
```

By setting the Key.KeyData.Data and Key.KeyData.Length fields to zero, the user requests SCCS to allocate the memory necessary to represent the key when CSSM_GenerateKey is called.

```
CSSM_GenerateKey(hKeyGenContext, CSSM_KEYUSE_ENCRYPT | CSSM_KEYUSE_DECRYPT,
            CSSM_KEYATTR_MODIFIABLE, NULL, Key)
```

CSSM_GenerateKey generates the key and updates the Key data structure accordingly. Once the key has been generated it is up to the application to delete the security context now that it is no longer needed. It does this by calling CSSM_DeleteContext.

```
CSSM_DeleteContext(hKeyGenContext)
```

GenerateSymmetricContext

The GenerateSymmetricContext function creates and returns a cryptographic context handle by calling CSSM_CSP_CreateSymmetricContext. The resulting context is used for the file encryption operations that use a symmetric key. The function parameters specify the CSP module handle, the desired algorithm ID (DES) and algorithm mode (cipher block chain mode), the key data, an initialization vector for the encryption, the type of padding (none), and the number of encryption rounds, in this case 0.

```
*hCryptoContext = CSSM_CSP_CreateSymmetricContext(hCSP,
                                                  CSSM_ALGID_DES,
                                                  CSSM_ALGMODE_CBCPadIV8,
                                                  Key,
                                                  &DESIVData,
                                                  CSSM_PADDING_NONE,
                                                  0);
```

Note that if the encryption were being performed using an asymmetric key, the application would call CSSM_CSP_CreateAsymmetricContext instead.

GenerateKeyRecoveryFieldsForContext

The steps involved in creating the key recovery fields are similar to those used to generate the symmetric key. A key recovery enablement context is created by calling:

```
hKRContext = CSSM_KR_CreateRecoveryEnablementContext(hKRSP, NULL, NULL);
```

CSSM_KR_CreateRecoveryEnablementContext creates a key recovery enablement context based on a KRSP handle (which determines the key recovery mechanism that is in use), and key recovery profiles for the local and remote parties involved in a cryptographic exchange. The local and remote key recovery profiles are CSSM_KRSP_PROFILE data structures, which contain authentication information for the respective parties. Since the profile values are NULL, SCCS uses the default values for local and remote profiles.

Next the key recovery fields are created with the function call:

```
CSSM_KR_GenerateRecoveryFields(hKRContext,
                               hCryptoContext, /*symmetric encryption
context*/
                               NULL, /*session attributes*/
                               KRFlags, /* KRFlags = 0 */
                               pKRFields); /*the key recovery fields
(output)*/
```

CSSM_KR_GenerateRecoveryFields generates the key recovery fields for a cryptographic association given the key recovery context, the session specific key recovery attributes, and the handle to the cryptographic context containing the key that is to be made recoverable. The session attributes and the flags are not interpreted at the SCCS layer. The KRFlags parameter may be used to fine tune the contents of the KRFields produced by this operation. The KRFields are in the form of an uninterpreted data blob.

Lastly, the context is destroyed by calling:

```
CSSM_DeleteContext(hKRContext)
```

WriteOutputFile

This function is called twice, once to write the key recovery fields to a file and again to write the encrypted file. The actual file encryption is performed in GenerateKeyRecoveryFieldsAndEncrypt using the CSSM_EncryptData function.

```
CSSM_EncryptData(hCryptoContext,
                 &ClearData, /*pointer to the input buffer*/
                 1, /*number of input buffers*/
                 &EncryptedData, /*pointer to output buffer*/
                 1, /*number of output buffers*/
                 &BytesEncrypted, /*size of the encrypted data*/
                 &RemData); /*buffer for padding encrypted data*/
```

Appendix A. Source Code for KR_FILE_ENCRYPT

This appendix contains the source code for the kr_file_encrypt program. The program consists of the following files.

- kr_file_encrypt.h
  This files contains the prototypes of public functions.
- main.c
  This file is the main program and command line parser.
- initialize.c
  This file shows how to initialize the SCCS initialized for use.
- attach.c
  This file attaches to two service provider modules, a Key Recovery module and a Cryptographic module. It illustrates two different methods of attaching to service provider modules.
- encrypt.c
  This file performs actual encryption and associated key recovery field generation and storage. It generates two output files, one containing the key recovery fields and one containing the encrypted file.

A.1 KR_FILE_ENCYPT.H

```
//-----------------------------------------------------------------------------
//
// COMPONENT_NAME: kr_file_encrypt
//
// (C) COPYRIGHT International Business Machines Corp. 1997
// All Rights Reserved
// Licensed Materials - Property of IBM
//
//-----------------------------------------------------------------------------
//
// FILE: kr_file_encrypt.h
//
// This file contains functions to take a clear file and produce its
// associated encrypted file and key recovery field file.  Although
// the symmetric encryption algorithm being used here is DES, others
// could be easily substituted with minimal change.
//
//-----------------------------------------------------------------------------

void Initialize(
    void);

void AttachCSPByAlgorithm(
    CSSM_CSP_HANDLE *hCSP,
    uint32 AlgorithmRequired);

void AttachKRSPByUserChoice(
    CSSM_KRSP_HANDLE *hKRSP);

void GenerateKeyRecoveryFieldsAndEncrypt(
    CSSM_CSP_HANDLE hCSP,
    CSSM_KRSP_HANDLE hKRSP,
    char *InputFilename);

extern CSSM_API_MEMORY_FUNCS MemoryFuncs;
```

A.2 MAIN.C

```
//--------------------------------------------------------------------------
//
// COMPONENT_NAME: kr_file_encrypt
//
// (C) COPYRIGHT International Business Machines Corp. 1997
// All Rights Reserved
// Licensed Materials - Property of IBM
//
//--------------------------------------------------------------------------
//
// FILE: main.c
//
// This file contains the main program of the kr_file_encrypt program.
// The command line arguments are processed here and other functions
// are called to perform subtasks such as initializing the CSSM,
// attaching the required service providers, generating key recovery
// fields and encrypting.
//
//--------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>

#include "cssm.h"
#include "kr_file_encrypt.h"

//--------------------------------------------------------------------------
//
// Function: ProcessArguments
//
// This function checks the command line arguments and provides syntax
//
//--------------------------------------------------------------------------
static void ProcessArguments(int argc, char *argv[], char **ClearFilename)
{
    // Check the number of arguments
    if (argc != 2) {
        printf("\n");
        printf("Usage: kr_file_encrypt <file to encrypt>\n");
        printf("\n");
        printf("  This utility encrypts the given file and generates\n");
        printf("  the associated key recovery fields.  These are the files\n");
        printf("  generated:\n");
        printf("\n");
        printf("    <filename>.enc - the encrypted file\n");
        printf("    <filename>.krf - the key recovery fields\n");
        printf("\n");
        exit(1);
    }

    // Get the name of the clear file
    *ClearFilename = argv[1];
}

//--------------------------------------------------------------------------
//
// Function: main
//
//--------------------------------------------------------------------------
int main(int argc, char *argv[])
{
    // Handle to the cryptographic service provider
    CSSM_CSP_HANDLE             hCSP;
    // Handle to the key recovery service provider
    CSSM_KRSP_HANDLE            hKRSP;
    char                        *ClearFilename;

    ProcessArguments(argc, argv, &ClearFilename);

    Initialize();

    // Set up cryptographic service provider
    AttachCSPByAlgorithm(&hCSP, CSSM_ALGID_DES);
```

```
    // Set up key recovery service provider.  Strong encryption can only
    // occur if the appropriate key recovery fields have been generated.
    AttachKRSPByUserChoice(&hKRSP);

    // Generate required key recovery fields and then encrypt
    GenerateKeyRecoveryFieldsAndEncrypt(hCSP, hKRSP, ClearFilename);

    return 0;
}
```

A.3 INITIALIZE.C

```
//--------------------------------------------------------------------------
//
// COMPONENT_NAME: kr_file_encrypt
//
// (C) COPYRIGHT International Business Machines Corp. 1997
// All Rights Reserved
// Licensed Materials - Property of IBM
//
//--------------------------------------------------------------------------
//
// FILE: initialize.c
//
// This file encapsulates how an application initializes the CSSM.  Memory
// management function tables are passed and versions are checked.
//
//--------------------------------------------------------------------------

#include <stdlib.h>
#include <stdio.h>

#include "cssm.h"
#include "kr_file_encrypt.h"

//
// Memory mangement function table.  See below.
//

CSSM_API_MEMORY_FUNCS   MemoryFuncs;

//
// This set of memory management funciton wrappers are required by CSSM
// to manage memory on behalf of the calling application.  Note: since the
// calling application is linked separately, it may have its own distinct
// implementation of memory management functions.
//

void *app_malloc(uint32 size, void *ref)           { return(malloc(size));     }
void app_free(void * ptr, void *ref)               { free(ptr);                }
void *app_calloc(uint32 n, uint32 size, void *ref) { return(calloc(n, size));  }
void *app_realloc(void *p, uint32 size, void *ref) { return(realloc(p, size)); }

//--------------------------------------------------------------------------
//
// Function: Initialize
//
// This function sets up memory management functions and calls CSSM_Init
//
//--------------------------------------------------------------------------
void Initialize(void)
{
    CSSM_ERROR_PTR          pError;
    // This is the version of the CSSM itself.
    CSSM_VERSION            Version = { CSSM_MAJOR, CSSM_MINOR };

    //
    // Initialize the application's memory management function table
    //

    MemoryFuncs.malloc_func     = app_malloc;
    MemoryFuncs.free_func       = app_free;
    MemoryFuncs.realloc_func    = app_realloc;
    MemoryFuncs.calloc_func     = app_calloc;

    //
    // The CSSM_Init function must be called before performing any other
    // CSSM API calls.  The expected CSSM major/minor version numbers
    // and the memory management function table are passed down.
    //

    if (CSSM_Init(&Version, &MemoryFuncs, NULL) != CSSM_OK)
    {
        printf("Error: could not intialize CSSM\n");
```

```
        pError = CSSM_GetError();
        printf("CSSM_Init error code = %d\n", pError->error);
        exit(1);
    }
}
```

A.4 ATTACH.C

```
//--------------------------------------------------------------------------
//
// COMPONENT_NAME: kr_file_encrypt
//
// (C) COPYRIGHT International Business Machines Corp. 1997
// All Rights Reserved
// Licensed Materials - Property of IBM
//
//--------------------------------------------------------------------------
//
// FILE: attach.c
//
// There are various levels of detail that applications can use when
// attaching to modules using the CSSM API.  In the simplest case, an
// application can hardcode a particular GUID so that it only works when
// a particular module is installed.  On the other hand, a more flexible
// application can be designed to look into the installed list of modules
// and choose one based on some attribute it has (capability, vendor
// name, hardware/software, etc.).
//
// This file shows two methods (among many) that can be used to attach a
// module.  In AttachCSPByAlgorithm(), the installed list of software
// cryptographic service providers is searched to find one that supports
// the required algorithm.  In AttachKRSPByUserChoice(), the list of
// installed key recovery service providers is presented and the user is
// asked to select one.
//
//--------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>

#include "cssm.h"
#include "kr_file_encrypt.h"

//--------------------------------------------------------------------------
//
// Function: AttachCSPByAlgorithm
//
// This function searches the list of all installed modules for a
// CSP that supports the required algorithm.
//
//--------------------------------------------------------------------------
void AttachCSPByAlgorithm(
    CSSM_CSP_HANDLE *hCSP,
    uint32 AlgorithmRequired)
{
    CSSM_ERROR_PTR          pError;         // error information
    CSSM_LIST_PTR           pModuleList;    // list of modules
    CSSM_MODULE_INFO_PTR    pModuleInfo;    // module info
    CSSM_CSPSUBSERVICE_PTR  pCspInfo;       // CSP module info
    CSSM_SOFTWARE_CSPSUBSERVICE_INFO_PTR pInfo;  // software CSP module info
    CSSM_CSP_CAPABILITY_PTR pCap;           // capabilities list
    uint32                  Total;          // miscellaneous
    CSSM_BOOL               Found;          // boolean for search
    uint32                  i;              // index
    uint32                  j;              // index
    uint32                  k;              // index
    uint32                  l;              // index

    //
    // Retrieve the total list of CSPs installed on the system at this time.
    //

    if ((pModuleList = CSSM_ListModules(CSSM_SERVICE_CSP, CSSM_TRUE)) == NULL)
    {
        pError = CSSM_GetError();
        printf("Error: could not list installed modules\n");
        printf("CSSM_ListModules error code = %d\n", pError->error);
        exit(1);
    }

    if (pModuleList->NumberItems == 0)
```

```
{
    printf("Error: no CSPs installed.\n");
    exit(1);
}

//
// Search through installed software CSPs for one that supports the
// encryption algorithm required
//

Found = CSSM_FALSE;

for (i = 0; !Found && i < (int)pModuleList->NumberItems; i++)
{
    pModuleInfo = CSSM_GetModuleInfo(&(pModuleList->Items[i].GUID),
                                     CSSM_SERVICE_CSP,
                                     0,
                                     CSSM_INFO_LEVEL_ALL_ATTR);

    for (j = 0; !Found && j < (int) pModuleInfo->NumberOfServices; j++)
    {
        pCspInfo = pModuleInfo->ServiceList[j].CspSubServiceList;

        for (k = 0; !Found && k < pModuleInfo->ServiceList[j].NumberOfSubServices; k++)
        {
            //
            // Note: to extend the search to hardware CSPs, a case
            // could be added to this switch construct.
            //
            switch (pCspInfo->CspType)
            {
                case CSSM_CSP_SOFTWARE:
                    pInfo = &(pCspInfo->SoftwareCspSubService);
                    Total = pInfo->NumberOfCapabilities;
                    for (l = 0; l < Total; l++)
                    {
                        pCap = &(pInfo->CapabilityList[l]);
                        if (pCap->AlgorithmType == AlgorithmRequired)
                        {
                            Found = CSSM_TRUE;
                        }
                    }
                break;

                default:
                break;
            } // switch
        } // for each subservice
    } // for each usage type
} // for each module

if (!Found)
{
    //
    // There were CSPs, but none of them matched
    //
    printf("Error: there are no suitable cryptographic service providers installed\n");
    exit(1);
}
else
{
    *hCSP = CSSM_ModuleAttach(&(pModuleList->Items[i-1].GUID),
                              &pModuleInfo->Version,
                              &MemoryFuncs,
                              0,
                              0,
                              0,
                              NULL,
                              NULL);
    if (*hCSP == 0)
    {
        pError = CSSM_GetError();
        printf("Error: could not attach to suitable cryptographic service provider\n");
        printf("CSSM_ModuleAttach error code = %d\n", pError->error);
        exit(1);
```

```
        }

    }

    // Successfully attached to desired CSP
}

//----------------------------------------------------------------------------
//
// Function: AttachKRSPByUserChoice
//
// This function lists the installed modules which are key recovery service
// providers and prompts the user to choose one.
//
//----------------------------------------------------------------------------
void AttachKRSPByUserChoice(
    CSSM_KRSP_HANDLE *hKRSP)
{
    CSSM_ERROR_PTR          pError;         // error info
    CSSM_LIST_PTR           pModuleList;    // list of modules
    CSSM_MODULE_INFO_PTR    pModuleInfo;    // module info
    CSSM_GUID               KrspGuid;       // KRSP module identifier
    CSSM_BOOL               ChoiceMade;     // boolean for menu
    uint32                  number;         // index
    uint32                  i;              // index

    //
    // Retrieve the total list of KRSPs installed on the system at this time.
    //

    if ((pModuleList = CSSM_ListModules(CSSM_SERVICE_KR, CSSM_TRUE)) == NULL)
    {
        pError = CSSM_GetError();
        printf("Error: could not list installed modules\n");
        printf("CSSM_ListModules error code = %d\n", pError->error);
        exit(1);
    }

    if (pModuleList->NumberItems == 0)
    {
        //
        // Exit when there are no KRSPs installed
        //
        printf("Error: no KRSPs installed!  Aborting.\n");
        exit(1);
    }
    else
    {
    //
    // Present a list of installed KRSPs to choose from
    //

        ChoiceMade = CSSM_FALSE;

        printf("These key recovery service providers are installed:\n\n");

        while (!ChoiceMade)
        {
            printf("\n");

            // for each module found
            for (i = 0; i < pModuleList->NumberItems; i++) {
                // list this module's name
                printf(" [%d] %s\n", i + 1, pModuleList->Items[i].Name);
            }

            printf("\nPlease enter the number of the one you wish to attach.\n");

            // read user's selection
            if ((scanf ("%d", &number) == 1) &&
                (number > 0) &&
                (number <= pModuleList->NumberItems)) {
                ChoiceMade = CSSM_TRUE;
            } else {
                printf("Error: invalid choice\n\n");
```

```
            }

            fflush(stdout);

        } // while choice not made

    }

    //
    // Get the GUID of the choice made and attach it to use it
    //

    KrspGuid = pModuleList->Items[number - 1].GUID;

    pModuleInfo = CSSM_GetModuleInfo(&KrspGuid,
                                     CSSM_SERVICE_KR,
                                     0,
                                     CSSM_INFO_LEVEL_ALL_ATTR);

    *hKRSP = CSSM_ModuleAttach(&KrspGuid,
                               &pModuleInfo->Version,
                               &MemoryFuncs,
                               0,
                               0,
                               0,
                               NULL,
                               NULL);

    if (*hKRSP == 0)
    {
        printf("Error: could not attach to the chosen KRSP named \"%s\"\n",
               pModuleList->Items[number - 1].Name);
        pError = CSSM_GetError();
        printf("CSSM_ModuleAttach error code = %d\n", pError->error);
        exit(1);
    }
}
```

A.5 ENCRYPT.C

```
//--------------------------------------------------------------------------
//
// COMPONENT_NAME: kr_file_encrypt
//
// (C) COPYRIGHT International Business Machines Corp. 1997
// All Rights Reserved
// Licensed Materials - Property of IBM
//
//--------------------------------------------------------------------------
//
// FILE: encrypt.c
//
// This file contains functions to take a clear file and produce its
// associated encrypted file and key recovery field file.  Although
// the symmetric encryption algorithm being used here is DES, others
// could be easily substituted with minimal change.
//
//--------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>

#include "cssm.h"
#include "kr_file_encrypt.h"

//
// Suffixes used for the names of generated files
//

#define KR_FIELDS_FILE_SUFFIX   ".krb"
#define ENCRYPTED_FILE_SUFFIX   ".enc"

//
// File maximums
//
#define MAX_CLEAR_FILE_SIZE     4096        // for simplification
#define PATH_MAX                 256        // for simplification
//
// DES algorithm parameters
//

#define DES_PAD_LEN     8
#define DES_IV_LEN      8

static unsigned char
DESIVBuffer[DES_IV_LEN] = { 0x03, 0xC4, 0x98, 0x1E, 0x71, 0xFF, 0xA2, 0x23 };

static CSSM_DATA
DESIVData = { sizeof DESIVBuffer, DESIVBuffer };


//--------------------------------------------------------------------------
//
// Function: GenerateKey
//
// This function generates a key using the given CSP
//
//--------------------------------------------------------------------------
static void GenerateKey(
    CSSM_CSP_HANDLE hCSP,
    CSSM_KEY_PTR Key)
{
    CSSM_CC_HANDLE  hKeyGenContext;     // key generation context
    CSSM_ERROR_PTR  pError;             // error info

    //
    // Create a key generation context which basically packages all parameters
    // into a "handle" for later reference
    //

    hKeyGenContext =
```

```
        CSSM_CSP_CreateKeyGenContext(hCSP,
                                     CSSM_ALGID_DES,
                                     NULL,
                                     64,
                                     NULL,NULL,NULL,NULL,NULL);

    if (hKeyGenContext == 0)
    {
        printf("Error: could not perform key generation setup.\n");
        pError = CSSM_GetError();
        printf("CSSM_CSP_CreateKeyGenContext error code = %d\n", pError->error);
        exit(1);
    }

    //
    // Generate a key
    //

    memset(Key, 0, sizeof(CSSM_KEY));

    if (CSSM_GenerateKey(hKeyGenContext, CSSM_KEYUSE_ENCRYPT | CSSM_KEYUSE_DECRYPT,
            CSSM_KEYATTR_MODIFIABLE, NULL, Key) != CSSM_OK)
    {
        printf("Error: could not generate a key.\n");
        pError = CSSM_GetError();
        printf("CSSM_CSP_GenerateKey error code = %d\n", pError->error);
        exit(1);
    }

    //
    // Delete the unneeded key generation context
    //

    if (CSSM_DeleteContext(hKeyGenContext) != CSSM_OK)
    {
        printf("Error: could not delete key generation context\n");
        pError = CSSM_GetError();
        printf("CSSM_DeleteContext error code = %d\n", pError->error);
        exit(1);
    }
}


//---------------------------------------------------------------------------
//
// Function: GenerateSymmetricContext
//
// This function sets the encryption algorithm parameters including the key
// itself, the algorithm mode, etc.
//
//---------------------------------------------------------------------------
static void GenerateSymmetricContext(
    CSSM_CSP_HANDLE hCSP,
    CSSM_KEY *Key,
    CSSM_CC_HANDLE *hCryptoContext)
{
    CSSM_ERROR_PTR  pError;             // error info

    //
    // Create a symmetric encryption context to package encryption parameters
    //

    *hCryptoContext =
        CSSM_CSP_CreateSymmetricContext(hCSP,
                                        CSSM_ALGID_DES,
                                        CSSM_ALGMODE_CBCPadIV8,
                                        Key,
                                        &DESIVData,
                                        CSSM_PADDING_NONE,
                                        0);

    if (hCryptoContext == 0)
    {
        printf("Error: could not perform symmetric encryption setup\n");
        pError = CSSM_GetError();
```

```
        printf("CSSM_CSP_CreateSymmetricContext error code = %d\n", pError->error);
        exit(1);
    }
}

//---------------------------------------------------------------------------
//
// Function: GenerateKeyRecoveryFieldsForContext
//
// This function generates the key recovery fields associated with a given
// symmetric context.  These key recovery fields can later be used to
// recover the encryption key by authorized parties.
//
//---------------------------------------------------------------------------
static void GenerateKeyRecoveryFieldsForContext(
    CSSM_KRSP_HANDLE hKRSP,
    CSSM_CC_HANDLE hCryptoContext,
    CSSM_DATA *pKRFields)
{
    CSSM_CC_HANDLE  hKRContext; // context for key recovery field generation
    CSSM_RETURN     RC;         // return code
    uint32          KRFlags;    // key recovery algorithm flags
    CSSM_ERROR_PTR  pError;     // error info

    //
    // Create a key recovery enablement context to set up for generation
    // of key recovery fields
    //

    hKRContext = CSSM_KR_CreateRecoveryEnablementContext(hKRSP, NULL, NULL);

    if (hKRContext == 0)
    {
        printf("Error: could not perform key recovery generation setup\n");
        printf("CSSM_KR_CreateRecoveryEnablementContext error code = %d\n",
               CSSM_GetError()->error);
        exit(1);
    }

    //
    // Actually generate the key recovery fields that can be used later on
    // by authorized parties to recover the encryption key
    //

    KRFlags = 0;

    RC = CSSM_KR_GenerateRecoveryFields(hKRContext,
                                        hCryptoContext,
                                        NULL,
                                        KRFlags,
                                        pKRFields);

    if (RC != CSSM_OK)
    {
        printf("Error: could not generate key recovery fields\n");
        pError = CSSM_GetError();
        printf("CSSM_KR_GenerateRecoveryFields error code = %d\n", pError->error);
        exit(1);
    }

    //
    // Clean up
    //

    if ((RC = CSSM_DeleteContext(hKRContext)) != CSSM_OK)
    {
        printf("Error: could not delete key recovery enablement context\n");
        pError = CSSM_GetError();
        printf("CSSM_DeleteContext error code = %d\n", pError->error);
        exit(1);
    }
}

//---------------------------------------------------------------------------
//
```

```
// Function: WriteOutputFile
//
// This function takes a data buffer represented by a CSSM_DATA type and
// writes it out to new file.  The new file's name is composed of the base
// and suffix strings provided.  This function is used to write out the
// encrypted data as well as the key recovery field data.
//
//--------------------------------------------------------------------------
static void WriteOutputFile(
    CSSM_DATA DataToWrite,
    char *FilenameBase,
    char *FilenameSuffix)
{
    char    OutputFilename[PATH_MAX];
    FILE    *OutputFile;
    int     BytesLeft;
    char    *LastByte;
    int     CurrentWritten;
    int     CurrentSize;
    char    *pCurrent;

    //
    // Compose name and open output file
    //

    strcpy(OutputFilename, FilenameBase);
    strcat(OutputFilename, FilenameSuffix);

    if ((OutputFile = fopen(OutputFilename, "wb")) == NULL)
    {
        printf("Error: could not open %s\n", OutputFilename);
        perror("fopen");
        exit(1);
    }

    //
    // Write data
    //

    LastByte    = DataToWrite.Data + DataToWrite.Length - 1;
    BytesLeft   = DataToWrite.Length;

    pCurrent = DataToWrite.Data;

    while (BytesLeft > 0)
    {
        if (pCurrent + BUFSIZ > LastByte)
            CurrentSize = LastByte - pCurrent;
        else
            CurrentSize = BUFSIZ;

        CurrentWritten = fwrite(pCurrent, 1, CurrentSize, OutputFile);

        if (ferror(OutputFile))
        {
            printf("Error: failed to write to file %s\n", OutputFilename);
            perror("fwrite");
            exit(1);
        }

        BytesLeft -= CurrentWritten;
    }

    fclose(OutputFile);

}

//--------------------------------------------------------------------------
//
// Function: GenerateKeyRecoveryFieldsAndEncrypt
//
// This function encrypts a file using strong encryption.  It performs all
// the necessary prerequisites such as generation of a key (could be replaced
// by string to key derivation) for the encryption, generation of the
// necessary key recovery fields, and actual encryption.  The encrypted
```

```
// file and the key recovery field file will be written out.
//
//-----------------------------------------------------------------------------
void GenerateKeyRecoveryFieldsAndEncrypt(
    CSSM_CSP_HANDLE hCSP,
    CSSM_KRSP_HANDLE hKRSP,
    char *InputFilename)
{
    FILE                *ClearFile;         // clear file's handle
    CSSM_CC_HANDLE      hCryptoContext;     // context handle for encryption
    CSSM_KEY            Key;                // the symmetric key for encryption
    int                 BytesRead;          // byte reading counter
    uint32              BytesEncrypted;     // byte encrypting counter
    unsigned char       ClearBuf[MAX_CLEAR_FILE_SIZE];  // buffer for cleartext
    CSSM_DATA           ClearData;          // buffer for cleartext
    CSSM_DATA           EncryptedData;          // buffer for ciphertext
    unsigned char       RemBuf[DES_PAD_LEN];// buffer for padding
    CSSM_DATA           RemData;            // buffer for padding
    CSSM_DATA           KRFData;            // buffer for key recovery fields
    CSSM_RETURN         RC;                 // return code

    //
    // Normally one would prompt the user for a string and convert it to
    // a clear key, but here is an example of the key generation APIs
    //

    GenerateKey(hCSP, &Key);

    GenerateSymmetricContext(hCSP, &Key, &hCryptoContext);

    GenerateKeyRecoveryFieldsForContext(hKRSP, hCryptoContext, &KRFData);

    WriteOutputFile(KRFData, InputFilename, KR_FIELDS_FILE_SUFFIX);

    //
    // Read the clear file in one buffer for simplification
    //

    if ((ClearFile = fopen(InputFilename, "rb")) == NULL)
    {
        printf("Error: could not open %s\n", InputFilename);
        perror("fopen");
        exit(1);
    }

    BytesRead = fread(ClearBuf, 1, MAX_CLEAR_FILE_SIZE, ClearFile);
    ClearData.Length = BytesRead;
    ClearData.Data = ClearBuf;

    if (BytesRead == 0)
    {
        printf("Error: did not read any bytes from file\n");
        exit(1);
    }

    if (!feof(ClearFile))
    {
        printf("Error: exceeded currently supported maximum clear file size\n");
        exit(1);
    }

    fclose(ClearFile);

    //
    // Encrypt the buffer
    //

    // Initialize the buffer that will hold the final block of the encryption
    memset(RemBuf, 0, sizeof(RemBuf));
    RemData.Length  = sizeof(RemBuf);
    RemData.Data    = RemBuf;

    // setup CipherBuf with the same length as ClearBuf
    EncryptedData.Data = (uint8 *) malloc (ClearData.Length);
    EncryptedData.Length = ClearData.Length;
```

```
    RC = CSSM_EncryptData(hCryptoContext,
                          &ClearData,
                          1,
                          &EncryptedData,
                          1,
                          &BytesEncrypted,
                          &RemData);

    // Move the final block of data to the end of the EncryptedBuf
    memcpy(EncryptedData.Data + BytesEncrypted, RemData.Data, RemData.Length);
    EncryptedData.Length =BytesEncrypted + RemData.Length;

    //
    // Write the encrypted file
    //

    WriteOutputFile(EncryptedData, InputFilename, ENCRYPTED_FILE_SUFFIX);

}
```

Appendix B. List of Acronyms

API	Application Program Interface
CA	Certificate Authority
CDSA	Common Data Security Architecture
CLI	Certificate Library Service Provider Interface
CRL	certificate revocation lists
CSP	Cryptographic Service Provider
CSSM	Common Security Services Manger
DLI	Data Store Library Service Provider Interface
EDI	Electronic Data Interchange
ISV	Independent Software Vendor
KRF	Key Recovery Field
KRSPI	Key Recovery Service Provider Interface
PKI	Public Key Infrastructure
SET	Secure Electronic Transaction
SCCS	Secure Cryptography and Certificate Services
SSL	Secure Socket Layer
SP	Service Provider
SPI	Service Provider Interface
TPI	Trust Policy Service Provider Interface

Glossary

Asymmetric algorithms	Cryptographic algorithms where one key is used to encrypt, and a second key is used to decrypt. They are often called public-key algorithms. One key is called the public key, and the other is called the private key or secret key. RSA (Rivest-Shamir-Adelman) is the most commonly used public-key algorithm. It can be used for encryption and for signing.
Certificate Authority (CA)	An entity that guarantees or sponsors a certificate. For example, a credit card company signs a cardholder's certificate to assure that the cardholder is who he or she claims to be. The credit card company is a certificate authority. Certificate authorities issue, verify, and revoke certificates.
Certificate	See Digital certificate.
Certificate chain	The hierarchical chain of all the other certificates used to sign the current certificate. This includes the Certificate Authority (CA) who signs the certificate, the CA who signed that CA's certificate, and so on. There is no limit to the depth of the certificate chain.
Certificate signing	The Certificate Authority (CA) can sign certificates it issues or co-sign certificates issued by another CA. In a general signing model, an object signs an arbitrary set of one or more objects. Hence, any number of signers can attest to an arbitrary set of objects. The arbitrary objects could be, for example, pieces of a document for libraries of executable code.
Certificate validity date	A start date and a stop date for the validity of the certificate. If a certificate expires, the Certificate Authority (CA) may issue a new certificate.
Cryptographic Service Providers (CSPs)	Modules that provide secure key storage and cryptographic functions. The modules may be software only or hardware with software drivers. The cryptographic functions provided may include: • Bulk encryption and decryption • Digital signing • Cryptographic hash • Random number generation • Key exchange
Cryptographic algorithm	A method or defined mathematical process for implementing a cryptography operation. A cryptographic algorithm may specify the procedure for encrypting and decrypting a byte stream, digitally signing an object, computing the hash of an object, generating a random number, etc. SCCS accommodates DES, RC2, RC4, IDEA and other encryption algorithms.
Cryptoki	Short for cryptographic token interface. See Token.

Digital certificate	The binding of some identification to a public key in a particular domain, as attested to directly or indirectly by the digital signature of the owner of that domain. A digital certificate is an unforgeable credential in cyberspace. The certificate is issued by a trusted authority, covered by that party's digital signature. The certificate may attest to the certificate holder's identity, or may authorize certain actions by the certificate holder. A certificate may include multiple signatures and may attest to multiple objects or multiple actions.
Digital signature	A data block that was created by applying a cryptographic signing algorithm to some other data using a secret key. Digital signatures may be used to: • Authenticate the source of a message, data, or document • Verify that the contents of a message hasn't been modified since it was signed by the sender • Verify that a public key belongs to a particular person Typical digital signing algorithms include MD5 with RSA encryption, and DSS, the proposed Digital Signature Standard defined as part of the U.S. Government Capstone project.
Hash algorithm	A cryptographic algorithm used to hash a variable-size input stream into a unique, fixed-sized output value. Hashing is typically used in digital signing algorithms. Example hash algorithms include MD and MD2 from RSA Data Security. MD5, also from RSA Data Security, hashes a variable-size input stream into a 128 bit output value. SHA, a Secure Hash Algorithm published by the U.S. Government, produces a 160-bit hash value from a variable-size input stream.
Key Recovery Fields	A block of data which is created from a symmetric key and key recovery profile information. The profile information includes public key certificates for whichever Key Recovery Agents (KRAs) will be recovery the keys if necessary, as well as the identities of the sending and receiving parties. This information is used to cryptographically store the session key in such a way that one or all of the KRAs will be able to retrieve the key from the KRF if they are requested to do so.
Key Recovery Service Providers (KRSPs)	Modules that provide key recovery enablement functions. The cryptographic functions provided may include: • Key recovery field generation • Key recovery field processing
Leaf Certificate	The certificate in a certificate chain that has not been used to sign another certificate in that chain. The leaf certificate is signed directly or transitively by all other certificates in the chain.
Message digest	The digital fingerprint of an input stream. A cryptographic hash function is applied to an input message arbitrary length and returns a fixed-size output, which is called the digest value.
Nonce	A sequence of random bits.

Owned certificate	A certificate whose associated secret or private key resides in a local CSP. Digital-signing algorithms require using owned certificates when signing data for purposes of authentication and non-repudiation. A system may use certificates it does not own for purposes other than signing.
Private key	The cryptographic key used to decipher messages in public-key cryptography. This key is kept secret by its owner.
Public key	The cryptographic key used to encrypt messages in public-key cryptography. The public key is available to multiple users (i.e., the public).
Random number generators	A function that generates cryptographically strong random numbers that cannot be easily guessed by an attacker. Random numbers are often used to generate session keys.
Root certificate	The prime certificate, such as the official certificate of a corporation or government entity. The root certificate is positioned at the top of the certificate hierarchy in its domain, and it guarantees the other certificates in its certificate chain. Each Certificate Authority has a self-signed root certificate. The root certificate's public key is the foundation of signature verification in its domain.
Secret key	See Private key.
Secure Cryptography and Certificate Services Architecture (SCCS)	A set of layered security services that address communications and data security problems in the emerging PC business space.
Secure Cryptography and Certificate Services Framework (SCCS)	defines five key service components: • Cryptographic Module Manager • Key Recovery Module Manager • Trust Policy Module Manager • Certificate Library Module Manager • Data Storage Library Module Manager The SCCS binds together all the security services required by PC applications. In particular, it facilitates linking digital certificates to cryptographic actions and trust protocols.
Security Context	A control structure that retains state information shared between a cryptographic service provider and the application agent requesting service from the CSP. Only one context can be active for an application at any given time, but the application is free to switch among contexts at will, or as required. A security context specifies CSP and application-specific values, such as required key length and desired hash functions.
Security-relevant event	An event where a CSP-provided function is performed, a security module is loaded, or a breach of system security is detected.

Session key	A cryptographic key used to encrypt and decrypt data. The key is shared by two or more communicating parties, who use the key to ensure privacy of the exchanged data.
Signature	See Digital signature.
Signature chain	The hierarchical chain of signers, from the root certificate to the leaf certificate, in a certificate chain.
Symmetric algorithms	Cryptographic algorithms that use a single secret key for encryption and decryption. Both the sender and receiver must know the secret key. Well-known symmetric functions include DES (Data Encryption Standard) and IDEA. The U.S. Government endorsed DES as a standard in 1977. It's an encryption block cipher that operates on 64-bit blocks with a 56-bit key. It is designed to be implemented in hardware, and works well for bulk encryption. IDEA (International Data Encryption Algorithm), one of the best known public algorithms, uses a 128-bit key.
Token	The logical view of a cryptographic device, as defined by a CSP's interface. A token can be hardware, a physical object, or software. A token contains information about its owner in digital form, and about the services it provides for electronic-commerce and other communication applications. A token is a secure device. It may provide a limited or a broad range of cryptographic functions. Examples of hardware tokens are smartcards and PMCIA cards.
Verification	The process of comparing two message digests. One message digest is generated by the message sender and included in the message. The message recipient computes the digest again. If the message digests are exactly the same, it shows or proves there was no tampering of the message contents by a third party (between the sender and the receiver).
Web of trust	A trust network among people who know and communicate with each other. Digital certificates are used to represent entities in the web of trust. Any pair of entities can determine the extent of trust between the two, based on their relationship in the web. Based on the trust level, secret keys may be shared and used to encrypt and decrypt all messages exchanged between the two parties. Encrypted exchanges are private, trusted communications.