IBM KeyWorks Toolkit

Certificate Library Interface (CLI) Specification

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

The IBM KeyWorks Toolkit defines the infrastructure for a complete set of security services. It is an extensible architecture that provides mechanisms to manage service provider security modules, which use cryptography as a computational base to build security protocols and security systems. Figure 1 shows the four basic layers of the IBM KeyWorks Toolkit: Application Domains, System Security Services, IBM KeyWorks Framework, and Service Providers. The IBM KeyWorks Framework is the core of this architecture. It provides a means for applications to directly access security services through the KeyWorks security application programming interface (API), or to indirectly access security services via layered security services and tools implemented over the KeyWorks API. The IBM KeyWorks Framework manages the service provider security modules and directs application calls through the KeyWorks API to the selected service provider module that will service the request. The KeyWorks API defines the interface for accessing security services. The KeyWorks service provider interface (SPI) defines the interface for service providers who develop plug-able security service products.

Service providers perform various aspects of security services, including:

- Cryptographic Services
- Key Recovery Services
- Trust Policy Libraries
- Certificate Libraries
- Data Storage Libraries

Cryptographic Service Providers (CSPs) are service provider modules that perform cryptographic operations including encryption, decryption, digital signing, key pair generation, random number generation, and key exchange. Key Recovery Service Providers (KRSPs) generate and process Key Recovery Fields (KRFs), which can be used to retrieve the original session key if it is lost, or if an authorized party requires access to the decryption key. Trust Policy (TP) modules implement policies defined by authorities and institutions, such as VeriSign (as a Certificate Authority (CA)) or MasterCard (as an institution). Each TP module embodies the semantics of a trust model based on using digital certificates as credentials. Applications may use a digital certificate as an identity credential and/or an authorization credential. Certificate Library (CL) modules provide format-specific, syntactic manipulation of memory-resident digital certificates and Certificate Revocation Lists (CRLs). Data Storage Library (DL) modules provide persistent storage for certificates and CRLs.

1.1 Service Provider Modules

An IBM KeyWorks service provider module is a Dynamically Linked Library (DLL) composed of functions that implement some or all of the KeyWorks module interfaces. Applications directly or indirectly select the modules used to provide security services to the application. Independent Software Vendors (ISVs) and hardware vendors will provide these service providers. The functionality of the service providers may be extended beyond the services defined by the KeyWorks API, by exporting additional services to applications using a KeyWorks PassThrough mechanism.

The API calls defined for service provider modules are categorized as service operations, module management operations, and module-specific operations. Service operations include functions that perform a security operation such as encrypting data, inserting a CRL into a data source, or verifying that a certificate is trusted. Module management functions support module installation, registration of module features and attributes, and queries to retrieve information on module availability and features. Module-specific operations are enabled in the API through passthrough functions whose behavior and use is defined by the service provider module developer.

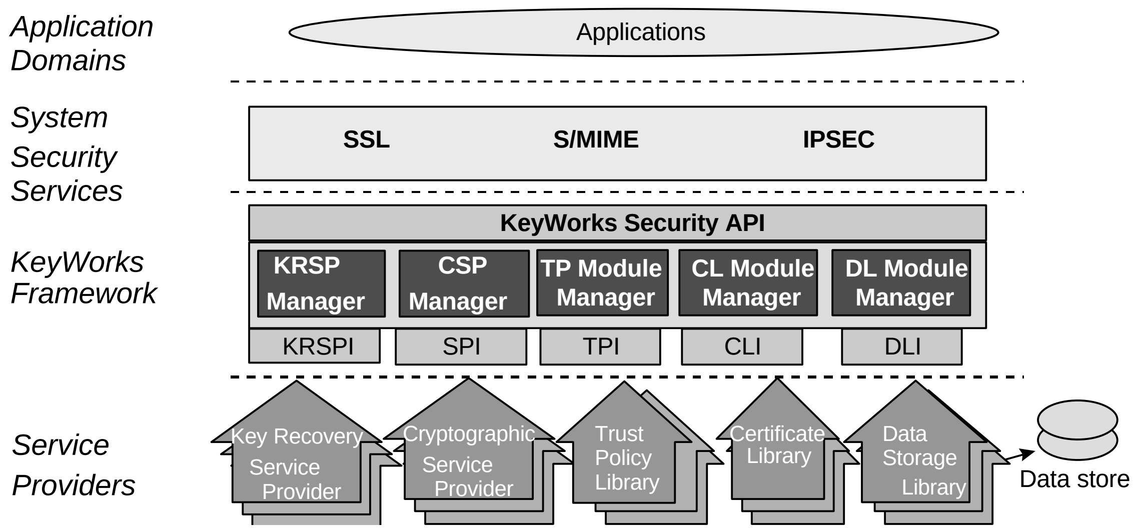

Figure 1. IBM KeyWorks Toolkit Architecture

Each module, regardless of the security services it offers, has the same set of module management responsibilities. Every module must expose functions that allow KeyWorks to indicate events such as module attach and detach. In addition, as part of the attach operation, every module must be able to verify its own integrity, verify the integrity of KeyWorks, and register with KeyWorks. Detailed information about service provider module structure, administration, and interfaces can be found in the *IBM KeyWorks Service Provider Module Structure & Administration Specification*.

1.2 Intended Audience

ISVs who want to develop their own TP service provider modules should use this document. These ISVs can be highly experienced software and security architects, advanced programmers, and sophisticated users. The intended audience of this document must be familiar with high-end cryptography and digital certificates. They must also be familiar with local and foreign government regulations on the use of cryptography, and the implication of those regulations for their applications and products. We assume that this audience is familiar with the basic capabilities and features of the protocols they are considering.

1.3 Documentation Set

The IBM KeyWorks 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 the IBM KeyWorks Toolkit. Both the electronic manuals and the Adobe Acrobat Reader are located in the IBM KeyWorks Toolkit `doc` subdirectory.

- *IBM KeyWorks Toolkit Developer's Guide*
  Document filename: kw_dev.pdf
  This document presents an overview of the IBM KeyWorks Toolkit. It explains how to integrate IBM KeyWorks into applications and contains a sample IBM KeyWorks application.

- *IBM KeyWorks Toolkit Application Programming Interface Specification*
  Document filename: kw_api.pdf
  his document defines the interface that application developers employ to access security services provided by IBM KeyWorks and service provider modules.

- *IBM KeyWorks Toolkit Service Provider Module Structure & Administration Specification*.
  Document filename: kw_mod.pdf
  This document describes the features common to all IBM KeyWorks service provider modules. It should be used in conjunction with the IBM KeyWorks service provider interface specifications in order to build a security service provider module.

- *IBM KeyWorks Toolkit Cryptographic Service Provider Interface Specification*
  Document filename: kw_spi.pdf
  This document defines the interface to which cryptographic service providers must conform in order to be accessible through IBM KeyWorks.

- *Key Recovery Service Provider Interface Specification*
  Document filename: kr_spi.pdf
  This document defines the interface to which key recovery service providers must conform in order to be accessible through IBM KeyWorks.

- *Key Recovery Server Installation and Usage Guide*
  Document filename: krs_gd.pdf
  This document describes how to install and use key recovery solutions using the components in the IBM Key Recovery Server.

- *IBM KeyWorks Toolkit Trust Policy Interface Specification*
  Document filename: kw_tp_spi.pdf
  This document defines the interface to which policy makers, such as certificate authorities, certificate issuers, and policy-making application developers, must conform in order to extend IBM KeyWorks with model or application-specific policies.

- *IBM KeyWorks Toolkit Certificate Library Interface Specification*
  Document filename: kw_cl_spi.pdf
  This document defines the interface to which library developers must conform to provide format-specific certificate manipulation services to numerous IBM KeyWorks applications and trust policy modules.

- *IBM KeyWorks Toolkit Data Storage Library Interface Specification*
  Document filename: kw_dl_spi.pdf
  This document defines the interface to which library developers must conform to provide format-specific or format-independent persistent storage of certificates.

1.4 References

Cryptography	*Applied Cryptography*, Schneier, Bruce**,** 2nd Edition, John Wiley and Sons, Inc., 1996.
	Handbook of Applied Cryptography, Menezes, A., Van Oorschot, P., and Vanstone, S., CRC Press, Inc., 1997.
	SDSI - A Simple Distributed Security Infrastructure, R. Rivest and B. Lampson, 1996.
	Microsoft CryptoAPI, Version 0.9, Microsoft Corporation, January 17, 1996.
CDSA Spec	*Common Data Security Architecture Specification*, Intel Architecture Labs, 1997.

CSSM API	*Common Security Services Manager Application Programming Interface Specification,* Intel Architecture Labs, 1997.
Key Escrow	*A Taxonomy for Key Escrow Encryption Systems,* Denning, Dorothy E. and Branstad, Dennis, Communications of the ACM, Vol. 39, No. 3, March 1996.
PKCS	*The Public-Key Cryptography Standards*, RSA Laboratories, Redwood City, CA: RSA Data Security, Inc.
IBM KeyWorks CLI	*Certificate Library Interface Specification,* Intel Architecture Labs, 1997.
IBM KeyWorks DLI	*Data Storage Library Interface Specification,* Intel Architecture Labs, 1997.
IBM KeyWorks KRI	*Key Recovery Service Provider Interface Specification,* Intel Architecture Labs, 1997.
IBM KeyWorks SPI	*Cryptographic Service Provider Interface Specification,* Intel Architecture Labs, 1997.
IBM KeyWorks TPI	*Trust Policy Interface Specification,* Intel Architecture Labs, 1997.
X.509	*CCITT. Recommendation X.509: The Directory – Authentication Framework*, 1988. CCITT stands for Comite Consultatif Internationale Telegraphique et Telephonique (International Telegraph and Telephone Consultative Committee)

Chapter 2. Certificate Library Interface

The primary purpose of a Certificate Library (CL) module is to perform syntactic operations on a specific certificate format, and its associated Certificate Revocation List (CRL) format. These manipulations encapsulate the complete life cycle of a certificate and the key pair associated with that certificate. Certificate and CRLs are related by the life cycle model and by the data formats used to represent them. For this reason, a single, cohesive library should manipulate these objects.

The CL encapsulates format-specific knowledge into a library that an application can access through IBM KeyWorks. These libraries allow applications and service provider modules to interact with Certificate Authorities (CAs) and to use certificates and CRLs for services such as signing, verification, creation and revocation without requiring knowledge of the certificate and CRL formats.

CLs manipulate memory-based objects only. The persistence of certificates, CRLs, and other security-related objects is an independent property of these objects. It is the responsibility of the application and/or the Trust Policy (TP) module to use data storage service provider modules to make objects persistent (if appropriate).

2.1 Certificate Life Cycle

The CL provides support for the certificate life cycle and for format-specific certificate or CRL manipulation, services that an application can access through KeyWorks. These libraries allow applications and service provider modules to create, sign, verify, and revoke certificates without requiring knowledge of certificate and CRL format and encoding.

A certificate is a form of credential. Under current certificate models, such as X.509, Simple Distributed Security Infrastructure (SDSI), Simple Public Key Infrastructure (SPKI), etc., a single certificate represents the identity of an entity (in the form of a binding between a name and a public key) and optionally associates authorizations with that entity. When a certificate is issued, the issuer includes a digital signature on the certificate. Verification of this signature is the mechanism used to establish trust in the identity and authorizations recorded in the certificate. Certificates can be signed by one or more other certificates. Root certificates are self-signed. The syntactic process of signing corresponds to establishing a trust relationship between the entities identified by the certificates.

Figure 2 presents the certificate life cycle. It begins with the registration process. During registration, the authenticity of a user's identity is verified. This can be a two-part process beginning with manual procedures requiring physical presence, followed by backoffice procedures to register results for use by the automated system. The level of verification associated with the identity of the individual will depend on the Security Policy and Certificate Management Practice Statements that apply to the individual who will receive a certificate, and the domain in which that certificate will be issued and used.

After registration, keying material is generated and a certificate is created. Once the private key material and public key certificate are issued to a user, and backed up if appropriate, the active phase of the certificate management life cycle begins. The active phase includes:

- Retrieval - Retrieves a certificate from a remote repository such as an X.500 directory.
- Verification - Verifies the validity dates and signatures on a certificate and revocation status.
- Revocation - Asserts that a previously legitimate certificate is no longer a valid certificate.
- Recovery - When an end user can no longer access encryption keys (e.g., forgotten password).
- Update - Issues a new public/private keypair when a legitimate pair has or will expire soon.

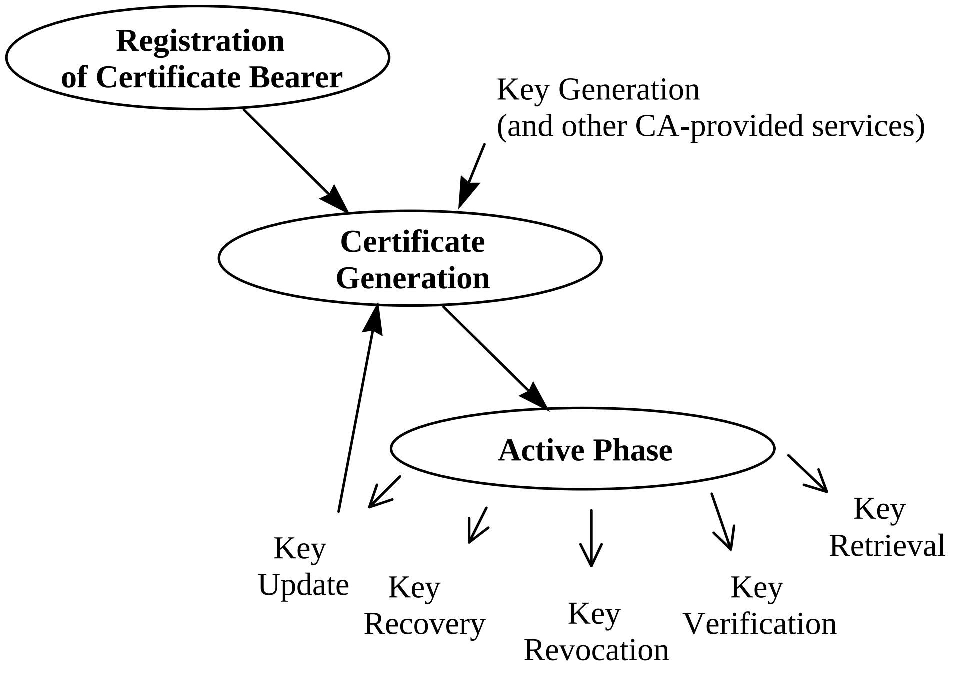

Figure 3. Certificate Life Cycle States and Actions

2.2 Certificate Library Interface Specification

The Certificate Library Interface (CLI) specifies the functions that a CL may make available to applications via KeyWorks in order to support a certificate and a CRL format. These functions mirror the KeyWorks API for certificates and CRLs. These functions include the basic areas of functionality expected of a CL, which include certificate operations, CRL operations, extensibility functions, and module management functions. The CL developer may choose to implement some or all of these CLI functions. The available functions are made known to KeyWorks at module attach time when it receives the CL's function table. In the function table, any unsupported function must have a NULL function pointer. The CL module developer is responsible for making the certificate format and general functionality known to application developers.

Certificate operations fall into three general areas, including:

- **Cryptographic Operations -** These operations include signing a certificate and verifying the signature on a certificate. It is expected that the CL will determine the certificate fields to be signed or verified, and will manage the interaction with a Cryptographic Service Provider (CSP) to perform the signing or verification.
- **Certificate Field Management -** Fields are added to a certificate when it is created. After the certificate is signed, the fields cannot be modified in any way. However, they can be queried for their values using the KeyWorks certificate interface.
- **Certificate Format Translation -** In the heterogeneous world of multiple certificate formats, CL modules may want to provide the service of translating between certificate formats. This translation would involve mapping the fields from one certificate format into another certificate format, while maintaining the original format for integrity verification purposes. For example, an X.509 Version 1 certificate may be exported to a Simple Distributed Security Infrastructure (SDSI) format or imported into an X.509 Version 3 certificate, but the original data and signature must somehow be maintained. The supported import and export types are registered with KeyWorks as part of CL installation.

To support new certificate types and new uses of certificates, the sign and verify operations in the CLI support a scope parameter. The scope parameter enables an application to sign a portion of the certificate, namely, the fields identified by the scope. This provides support for certificate models that permit field signing. CL modules that support existing certificate formats, such as X.509 Version 1, which sign and verify a predefined portion of the certificate, will ignore this parameter.

The CL module's certificate format is exposed via its fields. These fields will consist of tag/value pairs, where the tag is an object identifier (OID). These OIDs reference specific data types or data structures within the certificate or CRL. OIDs are defined by the CL developer at a granularity appropriate for the expected usage of the CL.

Operations on CRLs are comprised of cryptographic operations and field management operations on the CRL, as a whole, and on individual revocation records. The entire CRL can be signed or verified. This will ensure the integrity of the CRL's contents as it is passed between systems. Individual revocation records are signed when they are revoked and verified when they are queried. Certificates may be revoked and unrevoked by adding or removing them from the CRL at any time prior to its being signed. The contents of the CRL can be queried for all of its revocation records, specific certificates, or individual CRL fields.

A passthrough function is included in the CLI to allow CLs to expose additional services beyond what is currently defined in the KeyWorks API. These services should be syntactic in nature, meaning that they should be dependent on the data format of the certificates and CRLs manipulated by the library. KeyWorks will pass an operation identifier and input parameters from the application to the appropriate CL. Within the CL_PassThrough function in the CL, the input parameters will be interpreted and the appropriate operation performed. The CL developer is responsible for making known to the application the identity and parameters of the supported passthrough operations.

2.2.1 Certificate Operations

This section provides the detailed functions that compose the certificate operations in the CLI. It gives a high-level overview of each function's expected operation, its parameter definitions where necessary, and potential differences among CL module implementations.

CL_CertAbortQuery
This function releases the handle that was assigned by the CL_CertGetFirstFieldValue function to identify the results of a certificate query. It will only be supported by CL modules that allow multiple instances of an OID in a single certificate.

CL_CertCreateTemplate
This function creates a certificate in the CL module's native certificate format from the OID/value pairs provided by the application. The CL module makes its supported OIDs available to the application via the CertTemplate registered with KeyWorks and via the CL_CertDescribeFormat function. The CL module is responsible for indicating which fields are required to create a certificate. The returned certificate will not be a valid certificate until it has been signed.

CL_CertDescribeFormat
This function returns a list of OIDs corresponding to the data objects that compose the CL module's native certificate format.

CL_CertExport
This function translates a certificate from the native certificate type manipulated by the CL module into a foreign certificate type.

CL_CertGetAllFields

This function returns a list of all the fields in the input certificate, as described by their OID/value pairs.

CL_CertGetFirstFieldValue

This function returns the first field in the certificate that matches the input OID. If the certificate contains more than one instance of the requested OID, the CL module will return a handle to be used to obtain the additional instances, and a count of the total number of instances of this OID in the certificate. The application obtains the additional matching instances by repeated calls to the CL_CertGetNextFieldValue function.

CL_CertGetKeyInfo

This function retrieves the public key information stored in the certificate. In most certificate formats this includes multiple fields, but it may not include all of the fields defined by the CSSM_KEY data structure. Each CL module is responsible for making known which portions of the CSSM_KEY data structure will be returned.

CL_CertGetNextFieldValue

This function returns the next field that matched the OID given in the CL_CertGetFirstFieldValue function. It will be supported only by CL modules that allow multiple instances of an OID in a single certificate.

CL_CertImport

This function translates a certificate from a foreign certificate type to the native certificate type manipulated by the CL module.

CL_CertSign

This function will create a digital signature for the subject certificate by using the signer's certificate. The cryptographic context handle indicates the algorithm and parameters to be used for signing. Which field or fields should be signed will depend on the implementation of the CL module. A CL module that supports X.509 Version 1 certificates will sign all of the certificate fields and will ignore the SignScope parameter. A CL module that supports field signing would sign the subset of fields specified by the SignScope parameter.

CL_CertVerify

This function will verify the signer certificate's signature on the subject certificate. The cryptographic context handle indicates the algorithm and parameters to be used for verification. If the CL module supports field signing, the VerifyScope parameter may be used to identify the fields that were signed.

2.2.2 Certificate Revocation List Operations

This section provides a more detailed look at the functions that compose the CRL operations in the CLI. This section gives a high-level overview of each function's expected operation, its parameter definitions where necessary, and potential differences between CL module implementations.

CL_CrlAbortQuery

This function releases a handle that was assigned by the CL_CrlGetFirstFieldValue function to identify the results of a CRL query.

CL_CrlAddCert

This function revokes the input certificate by adding a record representing the certificate to the CRL. It then uses the revoker's certificate to sign the new record. The updated CRL is returned to the calling application.

CL_CrlCreateTemplate

This function creates a CRL in the CL module's native CRL format based on the OID/value pairs provided by the application. The CL module makes its supported OIDs available to the application via the CrlTemplate registered with KeyWorks and via the CL_CrlDescribeFormat function. The CL module is responsible for indicating which fields are required to create a CRL, or which fields cannot be set using this function. The returned CRL will not be a valid CRL until it has been signed.

CL_CrlDescribeFormat

This function returns a list of the OIDs that represent the fields in the CRL format manipulated by the CL module.

CL_CrlGetFirstFieldValue

This function returns the first field in the CRL that matches the input OID. It is likely that the CRL will support multiple instances of an OID that represents a revoked certificate record. If an application requests an OID that has multiple instances within the CRL, a results handle and a count of the number of matching instances will be returned along with the first instance of the OID. The application uses the results handle to obtain the additional matching instances by repeated calls to the CL_CrlGetNextFieldValue function. For example, given the OID for *revocation record*, this function would return the first revocation record in the CRL. The remaining revocation records could be obtained by successive calls to the CL_CrlGetNextFieldValue function.

CL_CrlGetNextFieldValue

This function returns the next field that matches the OID given in the CL_CrlGetFirstFieldValue function.

CL_CrlRemoveCert

This function unrevokes the input certificate by removing the record representing the certificate from the CRL. The updated CRL is returned to the calling application.

CL_CrlSetFields

This function sets the fields of an existing CRL to new values, based on the OID/value pairs provided by the application. The CL module is responsible for indicating any set of fields that must be or cannot be set using this function, and for specifying module-specific behavior such as overwriting existing fields, modifying extensions, or modifying CRL records. This operation is valid only if the CRL has not been closed by the process of signing the CRL (i.e., execution of the function CL_CrlSign). Once the CRL has been signed, fields cannot be changed.

CL_CrlSign

This function will create a digital signature for the entire CRL using the signer's certificate. The cryptographic context handle indicates the algorithm and parameters to be used for signing. The field or fields of the CRL that should be signed will depend on the implementation of the CL module. A CL module may choose to ignore the SignScope parameter if the fields to be signed are predefined. A CL module that supports field signing would sign the subset of fields specified by the SignScope parameter. Typically, this function will be used to sign the entire CRL prior to distributing it to other systems. The signature will be used to quickly detect tampering of the CRL. CRL queries may be performed on both signed and unsigned CRLs.

CL_CrlVerify

This function will check the signer's certificate signature on the subject CRL to determine whether any record in the CRL has been tampered with and whether the signer's certificate was actually used to sign the CRL. The cryptographic context handle indicates the algorithm and parameters to be used for verification. If the CL supports field signing on a CRL, the VerifyScope parameter may be used to identify the fields that were signed.

CL_IsCertInCrl

This function searches the CRL for a record corresponding to the input certificate.

2.2.3 Extensibility Functions

CL_PassThrough

This performs the CL module-specific function indicated by the operation ID. The operation ID specifies an operation that the CL has exported for use by an application or module. Such operations should be specific to the data format of the certificates and CRLs manipulated by the CL module.

2.3 Data Structures

This section describes the data structures that may be passed to or returned from a CL function. They will be used by applications to prepare data to be passed as input parameters into KeyWorks API function calls that will be passed without modification to the appropriate CL. The CL is then responsible for interpreting the data structures and returning the appropriate data structure to the calling application through the KeyWorks Framework. These data structures are defined in the header file, cssmtype.h, which is distributed with KeyWorks.

2.3.1 CSSM_BOOL

```
typedef uint32 CSSM_BOOL;

#define CSSM_TRUE   1
#define CSSM_FALSE  0
```

2.3.2 CSSM_CA_SERVICES

This bit-mask defines the additional certificate-creation-related services that an issuing CA (CA) can offer. Such services include (but are not limited to) archiving the certificate and keypair, publishing the certificate to one or more certificate directory services, and sending automatic, out-of-band notifications of the need to renew a certificate. A CA may offer any subset of these services. Additional services can be defined over time.

```
typedef uint32 CSSM_CA_SERVICES;
/* bit masks for additional CA services at cert enroll */
#define CSSM_CA_KEY_ARCHIVE 0x0001 /* archive cert & keys */
#define CSSM_CA_CERT_PUBLISH 0x0002 /* cert in directory service */
#define CSSM_CA_CERT_NOTIFY_RENEW 0x0004 /* notify at renewal time */
#define CSSM_CA_CRL_DISTRIBUTE 0x0010 /* push CRL to everyone */
```

2.3.3 CSSM_CERT_ENCODING

This variable specifies the certificate-encoding format supported by a CL.

```
typedef enum cssm_cert_encoding {
    CSSM_CERT_ENCODING_UNKNOWN =   0x00,
    CSSM_CERT_ENCODING_CUSTOM  =   0x01,
    CSSM_CERT_ENCODING_BER     =  0x02,
    CSSM_CERT_ENCODING_DER     =  0x03,
    CSSM_CERT_ENCODING_NDR     =  0x04
} CSSM_CERT_ENCODING, *CSSM_CERT_ENCODING_PTR;
```

2.3.4 CSSM_CERTGROUP

This structure contains a set of certificates. Typically, the certificates are related in some manner, but this is not required.

```
typedef struct cssm_certgroup {
    uint32 NumCerts;
    CSSM_DATA_PTR CertList;
    void *reserved;
} CSSM_CERTGROUP, *CSSM_CERTGROUP_PTR;
```

Definitions:

NumCerts - Number of certificates in the group.

CertList - List of certificates.

reserved - Reserved for future use.

2.3.5 CSSM_CERT_TYPE

This variable specifies the type of certificate format supported by a CL and the types of certificates understood for import and export. They are expected to define such well-known certificate formats as X.509 Version 3 and SDSI, as well as custom certificate formats. The list of enumerated values can be extended for new types by defining a label with an associated value greater than CSSM_CL_CUSTOM_CERT_TYPE.

```
typedef enum cssm_cert_type {
    CSSM_CERT_UNKNOWN =0x00,
    CSSM_CERT_X_509v1 =0x01,
    CSSM_CERT_X_509v2 =0x02,
    CSSM_CERT_X_509v3 =0x03,
    CSSM_CERT_Fortezza = 0x07,
    CSSM_CERT_PGP =0x04,
    CSSM_CERT_SPKI = 0x05,
    CSSM_CERT_SDSIv1 = 0x06,
    CSSM_CERT_Intel =0x08,
    CSSM_CERT_ATTRIBUTE_BER = 0x09, /* ber encoded X.509 attribute cert */
    CSSM_CERT_X509_CRL = 0x10,
    CSSM_CERT_LAST = 0x7FFF
} CSSM_CERT_TYPE, *CSSM_CERT_TYPE_PTR;

/* Applications wishing to define their own custom certificate
 * type should create a random uint32 whose value is greater than
 * the CSSM_CL_CUSTOM_CERT_TYPE */
    CSSM_CL_CUSTOM_CERT_TYPE 0x08000
```

2.3.6 CSSM_CL_CA_CERT_CLASSINFO

```
typedef struct cssm_cl_ca_cert_classinfo {
    CSSM_STRING CertClassName;
    CSSM_DATA CACert;
} CSSM_CL_CA_CERT_CLASSINFO, *CSSM_CL_CA_CERT_CLASSINFO_PTR;
```

Descriptions:

CertClassName - Name of a certificate class issued by this CA.

CACert - CA certificate for this cert class.

2.3.7 CSSM_CL_CA_PRODUCTINFO

This structure holds product information about a backend CA that is accessible to the CL module. The CL module vendor is not required to provide this information, but may choose to do so. For example, a CL module that implements upstream protocols to a particular type of commercial CA can record information about that CA service in this structure.

```
typedef struct cssm_cl_ca_productinfo {
    CSSM_VERSION StandardVersion;
    CSSM_STRING StandardDescription;
    CSSM_VERSION ProductVersion;
    CSSM_STRING ProductDescription;
    CSSM_STRING ProductVendor;
    CSSM_CERT_TYPE CertType;
    CSSM_CA_SERVICES AdditionalServiceFlags;
    uint32 NumberOfCertClasses;
    CSSM_CL_CA_CERT_CLASSINFO CertClassNames;
} CSSM_CL_CA_PRODUCTINFO, *CSSM_CL_CA_PRODUCTINFO_PTR;
```

Definitions:

StandardVersion - If this product conforms to an industry standard, this is the version number of that standard.

StandardDescription - If this product conforms to an industry standard, this is a description of that standard.

ProductVersion - Version number information for the actual product version used in this version of the CL module.

ProductDescription - A string describing the product.

ProductVendor - The name of the product vendor.

CertType - An enumerated value specifying the certificate and CRL type that the CA manages.

AdditionalServiceFlags - A bit-mask indicating the additional services a caller can request from a CA (as side effects and in conjunction with other service requests).

NumberOfCertClasses - The number of classes or levels of certificates managed by this CA.

CertClassNames - Names of the certificate classes issued by this CA.

2.3.8 CSSM_CL_ENCODER_PRODUCTINFO

This structure holds product information about embedded products that a CL module uses to provide its services. The CL module vendor is not required to provide this information, but may choose to do so. For example, a CL module that manipulates X.509 certificates may embed a third-party tool that parses, encodes, and decodes those certificates. The CL module vendor can describe such embedded products using this structure.

```
typedef struct cssm_cl_encoder_productinfo {
    CSSM_VERSION StandardVersion;
    CSSM_STRING StandardDescription;
    CSSM_VERSION ProductVersion;
    CSSM_STRING ProductDescription;
    CSSM_STRING ProductVendor;
    CSSM_CERT_TYPE CertType;
    uint32 ProductFlags;
} CSSM_CL_ENCODER_PRODUCTINFO, *CSSM_CL_ENCODER_PRODUCTINFO_PTR;
```

Definitions:

StandardVersion - If this product conforms to an industry standard, this is the version number of that standard.

StandardDescription - If this product conforms to an industry standard, this is a description of that standard.

ProductVersion **-** Version number information for the actual product version used in this version of the CL module.

ProductDescription **-** A string describing the product.

ProductVendor **-** The name of the product vendor.

CertType **-** An enumerated value specifying the certificate and CRL type that the CA manages.

ProductFlags **-** A bit-mask indicating any selectable features of the embedded product that the CL module selected for use.

2.3.9 CSSM_CL_HANDLE

The CSSM_CL_HANDLE is used to identify the association between an application thread and an instance of a CL module. CSSM_CL_HANDLE is assigned when an application causes KeyWorks to attach to a CL. It is freed when an application causes KeyWorks to detach from a CL. The application uses the CSSM_CL_HANDLE with every CL function call to identify the targeted CL. The CL module uses the CSSM_CL_HANDLE to identify the appropriate application's memory management routines when allocating memory on the application's behalf.

```
typedef uint32 CSSM_CL_HANDLE
```

2.3.10 CSSM_CLSUBSERVICE

Three structures are used to contain all of the static information that describes a CL module: cssm_moduleinfo, cssm_serviceinfo, and cssm_clsubservice. This descriptive information is securely stored in the KeyWorks registry when the CL module is installed with KeyWorks. A CL module may implement multiple types of services and organize them as subservices. For example, a CL module supporting X.509 encoded certificates may organize its implementation into three subservices: one for X.509 Version 1, a second for X.509 Version 2, and a third for X.509 Version 3. Most CL modules will implement exactly one subservice.

The descriptive information stored in these structures can be queried using the function CSSM_GetModuleInfo and specifying the CL module Globally Unique ID (GUID).

```
typedef struct cssm_clsubservice {
    uint32 SubServiceId;
    CSSM_STRING Description;
    CSSM_CERT_TYPE CertType;
    CSSM_CERT_ENCODING CertEncoding;
    CSSM_USER_AUTHENTICATION_MECHANISM AuthenticationMechanism;
    uint32 NumberOfTemplateFields;
    CSSM_OID_PTR CertTemplates;
    uint32 NumberOfTranslationTypes;
    CSSM_CERT_TYPE_PTR CertTranslationTypes;
    CSSM_CL_WRAPPEDPRODUCT_INFO WrappedProduct;
} CSSM_CLSUBSERVICE, *CSSM_CLSUBSERVICE_PTR;
```

Definitions:

SubServiceId **-** A unique, identifying number for the subservice described in this structure.

Description **-** A string containing a description name or title for this subservice.

CertType **-** An identifier for the type of certificate. This parameter is also used to determine the certificate data format.

CertEncoding - An identifier for the certificate-encoding format.

AuthenticationMechanism - An enumerated value defining the credential format accepted by the CL module. Authentication credential may be required when requesting certificate creation or other CL functions. Presented credentials must be of the required format.

NumberOfTemplateFields - The number of certificate fields. This number also indicates the length of the *CertTemplate* array.

CertTemplates - A pointer to an array of tag/value pairs which identify the field values of a certificate.

NumberOfTranslationTypes - The number of certificate types that this CL module can import and export. This number also indicates the length of the *CertTranslationTypes* array.

CertTranslationTypes - A pointer to an array of certificate types. This array indicates the certificate types that can be imported into and exported from this CL module's native certificate type.

WrappedProduct - A data structure describing the embedded products and CA service used by the CL module.

2.3.11 CSSM_CL_WRAPPEDPRODUCTINFO

This structure lists the set of embedded products and the CA service used by the CL module to implement its services. The CL module is not required to provide any of this information, but may choose to do so.

```
typedef struct cssm_cl_wrappedproductinfo {
    CSSM_CL_ENCODER_PRODUCTINFO_PTR EmbeddedEncoderProducts;
    uint32 NumberOfEncoderProducts;
    CSSM_CL_CA_PRODUCTINFO_PTR AccessibleCAProducts;
    uint32 NumberOfCAProducts;
} CSSM_CL_WRAPPEDPRODUCTINFO, *CSSM_CL_WRAPPEDPRODUCTINFO_PTR;
```

Definitions:

EmbeddedEncoderProducts - An array of structures that describe each embedded encoder product used in this CL module implementation.

NumberOfEncoderProducts - A count of the number of distinct embedded certificate encoder products used in the CL module implementation.

AccessibleCAProducts - An array of structures that describe each type of CA accessible through this CL module implementation.

NumberOfCAProducts - A count of the number of distinct CA products described in the array *AccessibleCAProducts*.

2.3.12 CSSM_DATA

The CSSM_DATA structure is used to associate a length, in bytes, with an arbitrary block of contiguous memory. This memory must be allocated and freed using the memory management routines provided by the calling application via KeyWorks.

```
typedef struct cssm_data {
    uint32 Length;
    uint8* Data;
} CSSM_DATA, *CSSM_DATA_PTR
```

Definitions:

Length **-** The length, in bytes, of the memory block pointed to by *Data*.

Data **-** A pointer to a contiguous block of memory.

2.3.13 CSSM_FIELD

This structure contains the OID/value pair for any item that can be identified by an OID. A CL module uses this structure to hold an OID/value pair for a field in a certificate or CRL.

```
typedef struct cssm_field {
    CSSM_OID FieldOid;
    CSSM_DATA FieldValue;
}CSSM_FIELD, *CSSM_FIELD_PTR
```

Definitions:

FieldOid **-** The OID that identifies the certificate or CRL data type or data structure.

FieldValue **-** A CSSM_DATA type which contains the value of the specified OID in a contiguous block of memory.

2.3.14 CSSM_HEADERVERSION

This data structure represents the version number of a key header structure. This version number is an integer that increments with each format revision of CSSM_KEYHEADER. The current revision number is represented by CSSM_KEYHEADER_VERSION, which equals 2 in this release of KeyWorks.

```
typedef uint32 CSSM_HEADERVERSION

#define CSSM_KEYHEADER_VERSION (2)
```

2.3.15 CSSM_KEY

This structure is used to represent keys in KeyWorks.

```
    typedef struct cssm_key{
        CSSM_KEYHEADER KeyHeader;
        CSSM_DATA KeyData;
    } CSSM_KEY, *CSSM_KEY_PTR;

    typedef CSSM_KEY CSSM_WRAP_KEY, *CSSM_WRAP_KEY_PTR;
```

Definitions:

KeyHeader **-** Header describing the key, fixed length.

KeyData **-** Data representation of the key, variable length.

2.3.16 CSSM_KEYHEADER

The key header contains meta-data about a key. It contains information used by a CSP or application when using the associated key data. The service provider module is responsible for setting the appropriate values.

```
typedef struct cssm_keyheader {
    CSSM_HEADERVERSION HeaderVersion;
    CSSM_GUID CspId;
    uint32 BlobType;
    uint32 Format;
    uint32 AlgorithmId;
    uint32 KeyClass;
    uint32 KeySizeInBits;
    uint32 KeyAttr;
    uint32 KeyUsage;
    CSSM_DATE StartDate;
    CSSM_DATE EndDate;
    uint32 WrapAlgorithmId;
    uint32 WrapMode;
    uint32 Reserved;
} CSSM_KEYHEADER, *CSSM_KEYHEADER_PTR;
```

Definitions:

HeaderVersion **-** This is the version of the keyheader structure.

CspId **-** If known, the GUID of the CSP that generated the key. This value will not be known if a key is received from a third party, or extracted from a certificate.

BlobType **-** Describes the basic format of the key data. It can be any one of the following values in Table 1.

Table 1. Keyblob Type Identifiers

Keyblob Type Identifier	**Description**
CSSM_KEYBLOB_RAW	The blob is a clear, raw key
CSSM_KEYBLOB_RAW_BERDER	The blob is a clear key, DER-encoded.
CSSM_KEYBLOB_REFERENCE	The blob is a reference to a key.
CSSM_KEYBLOB_WRAPPED	The blob is a wrapped RAW key.
CSSM_KEYBLOB_WRAPPED_BERDER	The blob is a wrapped DER-encoded key.
CSSM_KEYBLOB_OTHER	Other keyblob type.

Format **-** Describes the detailed format of the key data based on the value of the *BlobType* field. If the blob type has a nonreference basic type, then a CSSM_KEYBLOB_RAW_FORMAT identifier must be used, otherwise a CSSM_KEYBLOB_REF_FORMAT identifier is used. Any of the following values in Table 2 are valid as format identifiers.

Table 2. Keyblob Format Identifiers

Keyblob Format Identifier	**Description**
CSSM_KEYBLOB_RAW_FORMAT_NONE	No further conversion needs to be done.
CSSM_KEYBLOB_RAW_FORMAT_PKCS1	RSA PKCS1 V1.5
CSSM_KEYBLOB_RAW_FORMAT_PKCS3	RSA PKCS3 V1.5
CSSM_KEYBLOB_RAW_FORMAT_MSCAPI	Microsoft CAPI V2.0
CSSM_KEYBLOB_RAW_FORMAT_PGP	PGP
CSSM_KEYBLOB_RAW_FORMAT_FIPS186	U.S. Gov. FIPS 186 - DSS V
CSSM_KEYBLOB_RAW_FORMAT_BSAFE	RSA BSAFE V3.0

Keyblob Format Identifier	**Description**
CSSM_KEYBLOB_RAW_FORMAT_PKCS11	RSA PKCS11 V2.0
CSSM_KEYBLOB_RAW_FORMAT_CDSA	Intel CDSA
CSSM_KEYBLOB_RAW_FORMAT_OTHER	Other, CSP defined.
CSSM_KEYBLOB_REF_FORMAT_INTEGER	Reference is a number or handle.
CSSM_KEYBLOB_REF_FORMAT_STRING	Reference is a string or name.
CSSM_KEYBLOB_REF_FORMAT_OTHER	Other, CSP defined.

AlgorithmId **-** The algorithm for which the key was generated. This value does not change when the key is wrapped. Any of the defined KeyWorks algorithm IDs may be used.

KeyClass **-** Class of key contained in the key blob. Valid key classes are as follows in Table 3.

Table 3. Key Class Identifiers

Key Class Identifier	**Description**
CSSM_KEYCLASS_PUBLIC_KEY	Key is a public key.
CSSM_KEYCLASS_PRIVATE_KEY	Key is a private key.
CSSM_KEYCLASS_SESSION_KEY	Key is a session or symmetric key.
CSSM_KEYCLASS_SECRET_PART	Key is part of secret key.
CSSM_KEYCLASS_OTHER	Other.

KeySizeInBits **-** This is the logical size of the key in bits. The logical size is the value referred to when describing the length of the key. For instance, an RSA key would be described by the size of its modulus and a Digital Signature Algorithm (DSA) key would be represented by the size of its prime. Symmetric key sizes describe the actual number of bits in the key. For example, Data Encryption Standard (DES) keys would be 64 bits and an RC4 key could range from 1 to 128 bits.

KeyAttr **-** Attributes of the key represented by the data. These attributes are used by CSPs to convey information about stored or referenced keys. The attributes are represented as a bit-mask (see Table 4).

Table 4. Key Attribute Flags

Attribute	**Description**
CSSM_KEYATTR_PERMANENT	Key is stored persistently in the CSP, i.e., PKCS11 token object.
CSSM_KEYATTR_PRIVATE	Key is a private object and protected by either user login, a password, or both.
CSSM_KEYATTR_MODIFIABLE	The key or its attributes can be modified.
CSSM_KEYATTR_SENSITIVE	Key is sensitive. It may only be extracted from the CSP in a wrapped state. It will always be false for raw keys.
CSSM_KEYATTR_ALWAYS_SENSITIVE	Key has always been sensitive. It will always be false for raw keys.

Attribute	**Description**
CSSM_KEYATTR_EXTRACTABLE	Key is extractable from the CSP. If this bit is not set, the key is either not stored in the CSP or cannot be extracted from the CSP under any circumstances. It will always be false for raw keys.
CSSM_KEYATTR_NEVER_EXTRACTABLE	Key has never been extractable. It will always be false for raw keys.

KeyUsage **-** A bit-mask representing the valid uses of the key. Any of the following values in Table 5 are valid.

Table 5. Key Usage Flags

Usage Mask	**Description**
CSSM_KEYUSE_ANY	Key may be used for any purpose supported by the algorithm.
CSSM_KEYUSE_ENCRYPT	Key may be used for encryption.
CSSM_KEYUSE_DECRYPT	Key may be used for decryption.
CSSM_KEYUSE_SIGN	Key can be used to generate signatures. For symmetric keys this represents the ability to generate Message Authentication Codes (MACs).
CSSM_KEYUSE_VERIFY	Key can be used to verify signatures. For symmetric keys this represents the ability to verify MACs.
CSSM_KEYUSE_SIGN_RECOVER	Key can be used to perform signatures with message recovery. This form of a signature is generated using the CSSM_EncryptData API with the algorithm mode set to CSSM_ALGMODE_PRIVATE_KEY. This attribute is only valid for asymmetric algorithms.
CSSM_KEYUSE_VERIFY_RECOVER	Key can be used to verify signatures with message recovery. This form of a signature verified using the CSSM_DecryptData API with the algorithm mode set to CSSM_ALGMODE_PRIVATE_KEY. This attribute is only valid for asymmetric algorithms.
CSSM_KEYUSE_WRAP	Key can be used to wrap another key.
CSSM_KEYUSE_UNWRAP	Key can be used to unwrap a key.
CSSM_KEYUSE_DERIVE	Key can be used as the source for deriving other keys.

StartDate **-** Date from which the corresponding key is valid. All fields of the CSSM_DATA structure will be set to zero if the date is unspecified or unknown. This date is not enforced by the CSP.

EndDate **-** Data that the key expires and can no longer be used. All fields of the CSSM_DATA structure will be set to zero if the date is unspecified or unknown. This date is not enforced by the CSP.

WrapAlgorithmId **-** If the key data contains a wrapped key, this field contains the algorithm used to create the wrapped blob. This field will be set to CSSM_ALGID_NONE if the key is not wrapped.

WrapMode **-** If the wrapping algorithm supports multiple wrapping modes, this field contains the mode used to wrap the key. This field is ignored if the *WrapAlgorithmId* is CSSM_ALGID_NONE.

Reserved **-** This field is reserved for future use. It should always be set to zero.

2.3.17 CSSM_KEY_SIZE

This structure holds the key size and the effective key size for a given key. The metric used is bits. The number of effective bits is the number of key bits that can be used in a cryptographic operation compared with the number of bits that may be present in the key. When the number of effective bits is less than the number of actual bits, this is known as *dumbing down*.

```
typedef struct cssm_key_size {
    uint32 KeySizeInBits;/* Key size in bits */
    uint32 EffectiveKeySizeInBits; /* Effective key size in bits */
} CSSM_KEYSIZE, *CSSM_KEYSIZE_PTR
```

Definitions:

KeySizeInBits **-** The actual number of bits in a key.

EffectiveKeySizeInBits **-** The number of key bits that can be used for cryptographic operations.

2.3.18 CSSM_KEY_TYPE

```
typedef uint32 CSSM_KEY_TYPE, *CSSM_KEY_TYPE_PTR;
```

2.3.19 CSSM_SPI_MEMORY_FUNCS

This structure is used by KeyWorks to pass an application's memory function table to the service provider modules. The functions are used when memory needs to be allocated by the service provider module for returning data structures to the applications.

```
typedef struct cssm_spi_func_tbl {
    void *(*malloc_func) (CSSM_HANDLE AddInHandle, uint32 Size);
    void (*free_func) (CSSM_HANDLE AddInHandle, void *MemPtr);
    void  *(*realloc_func) (CSSM_HANDLE AddInHandle, void *MemPtr, uint32
Size);
    void  *(*calloc_func) (CSSM_HANDLE AddInHandle, uint32 Num, uint32 Size);
} CSSM_SPI_MEMORY_FUNCS, *CSSM_SPI_MEMORY_FUNCS_PTR;
```

Definitions:

malloc_func **-** Pointer to function that returns a void pointer to the allocated memory block of at least *size* bytes from heap *AllocRef.*

free_func **-** Pointer to function that deallocates a previously allocated memory block (*memblock*) from heap *AllocRef.*

realloc_func **-** Pointer to function that returns a void pointer to the reallocated memory block (*memblock*) of at least *size* bytes from heap *AllocRef.*

calloc_func **-** Pointer to function that returns a void pointer to an array of *num* elements of length *size* initialized to zero from heap *AllocRef.*

AllocRef **-** – Pointer that can be used at the discretion of the application developer to implement additional memory management features such as usage counters.

2.3.20 CSSM_OID

The OID is used to hold an identity for the data types and data structures that comprise the fields of a certificate or CRL. The underlying representation and meaning of the identifier is defined by the CL module.

For example, a CL module can choose to represent its identifiers in any of the following forms:

- A character string in a character set native to the platform.
- A DER-encoded X.509 OID that must be parsed.
- An S-expression that must be evaluated.
- An enumerated value that is defined in header files supplied by the CL module.

```
typedef CSSM_DATA CSSM_OID, *CSSM_OID_PTR;
```

2.3.21 CSSM_RETURN

```
typedef enum cssm_return {
    CSSM_OK = 0,
    CSSM_FAIL = -1
} CSSM_RETURN;
```

2.3.22 CSSM_REVOKE_REASON

This list defines the possible reasons why a certificate may be revoked.

```
typedef enum cssm_revoke_reason {
    CSSM_REVOKE_CUSTOM,
    CSSM_REVOKE_UNSPECIFIC,
    CSSM_REVOKE_KEYCOMPROMISE,
    CSSM_REVOKE_CACOMPROMISE,
    CSSM_REVOKE_AFFILIATIONCHANGED,
    CSSM_REVOKE_SUPERCEDED,
    CSSM_REVOKE_CESSATIONOFOPERATION,
    CSSM_REVOKE_CERTIFICATEHOLD,
    CSSM_REVOKE_CERTIFICATEHOLDRELEASE,
    CSSM_REVOKE_REMOVEFROMCRL
} CSSM_REVOKE_REASON
```

2.4 Certificate Operations

This section describes the function prototypes and error codes expected for the functions in the CLI. The functions will be exposed to KeyWorks via a function table, so the function names may vary at the discretion of the CL developer. However, the function parameter list and return type must match the prototypes given in this section in order to be used by applications.

2.4.1 CL_CertAbortQuery

CSSM_RETURN CSSMAPI CL_CertAbortQuery (CSSM_CL_HANDLE CLHandle,
CSSM_HANDLE ResultsHandle)

This function terminates the query initiated by CL_CertGetFirstFieldValue and allows the CL to release all intermediate state information associated with the query.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

ResultsHandle (input)
The handle that identifies the results of a certificate query.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred.
Use CSSM_GetError to obtain the error code.

See Also

CL_CertGetFirstFieldValue, CL_CertGetNextFieldValue

2.4.2 CL_CertCreateTemplate

CSSM_DATA_PTR CSSMAPI CL_CertCreateTemplate (CSSM_CL_HANDLE CLHandle,
const CSSM_FIELD_PTR CertTemplate
uint32 NumberOfFields)

This function allocates and initializes memory for a certificate based on the input OID/value pairs specified in the CertTemplate. The initialization process includes encoding all certificate field values according to the format required by the certificate representation. The function returns the initialized template containing encoded values. The memory is allocated using the calling application's memory management routines.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

CertTemplate (input)
A pointer to an array of OID/value pairs that identify the field values to initialize a new certificate.

NumberOfFields (input)

The number of certificate field values specified in the CertTemplate.

Return Value

A pointer to the CSSM_DATA structure containing the unsigned certificate template. If the return pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CL_CertRequest, CL_CertGetFirstFieldValue

2.4.3 CL_CertDescribeFormat

CSSM_OID_PTR CSSMAPI CL_CertDescribeFormat (CSSM_CL_HANDLE CLHandle,
uint32 *NumberOfFields)

This function returns a list of the OIDs used to describe the certificate format supported by the specified CL.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

NumberOfFields (output)
The length of the output OID array.

Return Value

A pointer to the array of CSSM_OID structures which are supported for certificate operations in the specified CL module. If the return pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CL_CertGetFirstFieldValue

2.4.4 CL_CertExport

CSSM_DATA_PTR CSSMAPI CL_CertExport (CSSM_CL_HANDLE CLHandle,
CSSM_CERT_TYPE TargetCertType,
const CSSM_DATA_PTR NativeCert)

This function exports a certificate from the native format of the specified CL into the specified target certificate format.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

TargetCertType (input)
A unique value that identifies the target type of the certificate being exported.

NativeCert (input)
A pointer to the CSSM_DATA structure containing the certificate to be exported.

Return Value

A pointer to the CSSM_DATA structure containing the target-type certificate exported from the native certificate. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CL_CertImport

2.4.5 CL_CertGetAllFields

CSSM_FIELD_PTR CSSMAPI CL_CertGetAllFields (CSSM_CL_HANDLE CLHandle,
const CSSM_DATA_PTR Cert,
uint32 *NumberOfFields)

This function returns a list of the fields in the input certificate, as described by their OID/value pairs.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate whose fields will be returned.

NumberOfFields (output)
The length of the output CSSM_FIELD array.

Return Value

A pointer to an array of CSSM_FIELD structures that describe the contents of the certificate using OID/value pairs. If the return pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CL_CertGetFirstFieldValue

2.4.6 CL_CertGetFirstFieldValue

CSSM_DATA_PTR CSSMAPI CL_CertGetFirstFieldValue (CSSM_CL_HANDLE CLHandle,
const CSSM_DATA_PTR Cert,
const CSSM_OID_PTR CertField,
CSSM_HANDLE_PTR ResultsHandle,
uint32 *NumberOfMatchedFields)

This function returns the value of the designated certificate field. If more than one field matches the *CertField* OID, the first matching field will be returned. The number of matching fields is an output parameter, as is the ResultsHandle to be used to retrieve the remaining matching fields.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate.

CertField (input)
A pointer to an OID that identifies the field value to be extracted from the *Cert*.

ResultsHandle (output)
A pointer to the CSSM_HANDLE that should be used to obtain any additional matching fields.

NumberOfMatchedFields (output)
The number of fields that match the *CertField* OID.

Return Value

A pointer to the CSSM_DATA structure containing the value of the requested field. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CL_CertGetNextFieldValue, CL_CertAbortQuery, CL_CertGetAllFields, CL_CertDescribeFormat

2.4.7 CL_CertGetKeyInfo

CSSM_KEY_PTR CSSMAPI CL_CertGetKeyInfo (CSSM_CL_HANDLE CLHandle,
const CSSM_DATA_PTR Cert)

This function obtains information about the certificate's public key. Ideally, this information comprises the key fields the application needs to create a cryptographic context that uses this certificate's key.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate from which to extract the public key information.

Return Value

A pointer to the CSSM_KEY structure containing the public key and possibly other key information. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

2.4.8 CL_CertGetNextFieldValue

CSSM_DATA_PTR CSSMAPI CL_CertGetNextFieldValue (CSSM_CL_HANDLE CLHandle,
CSSM_HANDLE ResultsHandle)

This function returns the next certificate field that matched the OID in a call to CL_CertGetFirstFieldValue.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

ResultsHandle (input)
The handle that identifies the results of a certificate query.

Return Value

A pointer to the CSSM_DATA structure containing the value of the requested field. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CL_CertGetFirstFieldValue, CL_CertAbortQuery

2.4.9 CL_CertImport

CSSM_DATA_PTR CSSMAPI CL_CertImport (CSSM_CL_HANDLE CLHandle,
CSSM_CERT_TYPE ForeignCertType,
const CSSM_DATA_PTR ForeignCert)

This function imports a certificate from the input format into the native format of the specified CL.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

ForeignCertType (input)
A unique value that identifies the type of the certificate being imported.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate to be imported into the native type.

Return Value

A pointer to the CSSM_DATA structure containing the native-type certificate imported from the foreign certificate. Use CSSM_GetError to obtain the error code.

See Also

CL_CertExport

2.4.10 CL_CertSign

```
CSSM_DATA_PTR CSSMAPI CL_CertSign  (CSSM_CL_HANDLE CLHandle,
                                    CSSM_CC_HANDLE CCHandle,
                                    const CSSM_DATA_PTR SubjectCert,
                                    const CSSM_DATA_PTR SignerCert,
                                    const CSSM_FIELD_PTR SignScope,
                                    uint32 ScopeSize)
```

This function signs the fields of the input certificate as indicated by the *SignScope* array.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

SubjectCert (input)
A pointer to the CSSM_DATA structure containing the certificate to be signed.

SignerCert (input)
A pointer to the CSSM_DATA structure containing the certificate to be used to sign the subject certificate.

SignScope (input)
A pointer to the CSSM_FIELD array containing the tag/value pairs of the fields to be signed. A NULL input signs all the fields in the certificate.

ScopeSize (input)
The number of entries in the sign scope list.

Return Value

A pointer to the CSSM_DATA structure containing the signed certificate. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CL_CertVerify

2.4.11 CL_CertVerify

```
CSSM_BOOL CSSMAPI CL_CertVerify  (CSSM_CL_HANDLE CLHandle,
                                  CSSM_CC_HANDLE CCHandle,
                                  const CSSM_DATA_PTR SubjectCert,
                                  const CSSM_DATA_PTR SignerCert,
                                  const CSSM_FIELD_PTR VerifyScope,
                                  uint32 ScopeSize)
```

This function verifies that the signed certificate has not been altered since it was signed by the designated signer. It does this by verifying the digital signature on the VerifyScope fields.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

SubjectCert (input)
A pointer to the CSSM_DATA structure containing the signed certificate.

SignerCert (input)
A pointer to the CSSM_DATA structure containing the certificate used to sign the subject certificate.

VerifyScope (input)
A pointer to the CSSM_FIELD array containing the tag/value pairs of the fields to be verified. A NULL input verifies all the fields in the certificate.

ScopeSize (input)
The number of entries in the verify scope list.

Return Value

CSSM_TRUE if the certificate verified. CSSM_FALSE if the certificate did not verify or an error condition occurred. Use CSSM_GetError to obtain the error code.

See Also

CL_CertSign

2.5 Certificate Revocation List Operations

This section describes the function prototypes supported by a CL module for operations on CRLs.
The functions will be exposed to KeyWorks through a function table, so the function names may vary at the discretion of the CL developer. However, the function parameter list and return type must match the prototypes given in this section in order to be used by applications.

2.5.1 CL_CrlAbortQuery

```
CSSM_RETURN CSSMAPI CL_CrlAbortQuery  (CSSM_CL_HANDLE CLHandle,
                                        CSSM_HANDLE ResultsHandle)
```

This function terminates the query initiated by CL_CrlGetFirstFieldValue and allows the CL to release all intermediate state information associated with the query.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

ResultsHandle (input)
The handle that identifies the results of a CRL query.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred.
Use CSSM_GetError to obtain the error code.

See Also

CL_CrlGetFirtsFieldValue, CL_CrlGetNextFieldValue

2.5.2 CL_CrlAddCert

```
CSSM_DATA_PTR CSSMAPI CL_CrlAddCert  (CSSM_CL_HANDLE CLHandle,
                                      CSSM_CC_HANDLE CCHandle,
                                      const CSSM_DATA_PTR Cert,
                                      const CSSM_DATA_PTR RevokerCert,
                                      CSSM_REVOKE_REASON RevokeReason,
                                      const CSSM_DATA_PTR OldCrl)
```

This function revokes the input certificate by adding a record representing the certificate to the CRL. It uses the revoker's certificate to sign the new record in the CRL. The reason for revoking the certificate may also be stored in the revocation record.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate to be revoked.

RevokerCert (input)
A pointer to the CSSM_DATA structure containing the revoker's certificate.

RevokeReason (input)
The reason for revoking the certificate.

OldCrl (input)
A pointer to the CSSM_DATA structure containing the CRL to which the newly revoked certificate will be added.

Return Value

A pointer to the CSSM_DATA structure containing the updated CRL. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CL_CrlRemoveCert

2.5.3 CL_CrlCreateTemplate

CSSM_DATA_PTR CSSMAPI CL_CrlCreateTemplate (CSSM_CL_HANDLE CLHandle,
const CSSM_FIELD_PRT CrlTemplate,
uint32 NumberOfFields)

This function creates an unsigned, memory-resident CRL. Fields in the CRL are initialized with the descriptive data specified by the OID/value input pairs. The specified OID/value pairs can initialize all or a subset of the general attribute fields in the new CRL, though the module developer may specify a set of fields that must be or cannot be set using this operation. Subsequent values may be set using the CL_CrlSetFields operation.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

CrlTemplate (input)
An array of OID/value pairs specifying the initial values for descriptive data fields of the new CRL.

NumberOfFields (input)
The number of OID/value pairs specified in the CrlTemplate input parameter.

Return Value

A pointer to the CSSM_DATA structure containing the new CRL. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

2.5.4 CL_CrlDescribeFormat

CSSM_OID_PTR CSSMAPI CL_CrlDescribeFormat (CSSM_CL_HANDLE CLHandle,
uint32 *NumberOfFields)

This function returns a list of the OIDs used to describe the CRL format supported by the specified CL.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

NumberOfFields (output)
The length of the output array.

Return Value

A pointer to the array of CSSM_OID structures which are supported for CRL operations in the specified CL module. If the return pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

2.5.5 CL_CrlGetFirstFieldValue

```
CSSM_DATA_PTR CSSMAPI CL_CrlGetFirstFieldValue  (CSSM_CL_HANDLE CLHandle,
                                                  const CSSM_DATA_PTR Crl,
                                                  const CSSM_OID_PTR CrlField,
                                                  CSSM_HANDLE_PTR ResultsHandle,
                                                  uint32 *NumberOfMatchedFields)
```

This function returns the value of the designated CRL field. If more than one field matches the *CrlField* OID, the first matching field will be returned. The number of matching fields is an output parameter, as is the ResultsHandle to be used to retrieve the remaining matching fields.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

Crl (input)
A pointer to the CSSM_DATA structure that contains the CRL from which the first revocation record will be retrieved.

CrlField (input)
A pointer to an OID that identifies the field value to be extracted from the *Crl*.

ResultsHandle (output)
A pointer to the CSSM_HANDLE, which should be used to obtain any additional matching fields.

NumberOfMatchedFields (output)
The number of fields that match the *CrlField* OID.

Return Value

Returns a pointer to a CSSM_DATA structure containing the first field that matched the *CrlField*. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CL_CrlGetNextFieldValue, CL_CrlAbortQuery

2.5.6 CL_CrlGetNextFieldValue

CSSM_DATA_PTR CSSMAPI CL_CrlGetNextFieldValue (CSSM_CL_HANDLE CLHandle,
CSSM_HANDLE ResultsHandle)

This function returns the next CRL field that matched the OID in a call to CL_CrlGetFirstFieldValue.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

ResultsHandle (input)
The handle that identifies the results of a CRL query.

Return Value

Returns a pointer to a CSSM_DATA structure containing the next field in the CRL, which matched the *CrlField* specified in the CL_CrlGetFirstFieldValue function. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CL_CrlGetFirstFieldValue, CL_CrlAbortQuery

2.5.7 CL_CrlRemoveCert

```
CSSM_DATA_PTR CSSMAPI CL_CrlRemoveCert  (CSSM_CL_HANDLE CLHandle,
                                          const CSSM_DATA_PTR Cert,
                                          const CSSM_DATA_PTR OldCrl)
```

This function unrevokes a certificate by removing it from the input CRL.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate to be unrevoked.

OldCrl (input)
A pointer to the CSSM_DATA structure containing the CRL from which the certificate will be removed.

Return Value

A pointer to the CSSM_DATA structure containing the updated CRL. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CL_CrlAddCert

2.5.8 CL_CrlSetFields

```
CSSM_DATA_PTR CSSMAPI CL_CrlSetFields  (CSSM_CL_HANDLE CLHandle,
                                        const CSSM_FIELD_PRT CrlTemplate,
                                        uint32 NumberOfFields,
                                        const CSSM_DATA_PTR OldCrl)
```

This function will set the fields of the input CRL to the new values specified by the input OID/value pairs. The module developer may specify a set of fields that must be or cannot be set using this operation. This operation is valid only if the CRL has not been closed by the process of signing the CRL (i.e., execution of the function CL_CrlSign). Once the CRL has been signed, fields cannot be changed.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

CrlTemplate (input)
Any array of field OID/value pairs containing the values to initialize the CRL attribute fields.

NumberOfFields (input)
The number of OID/value pairs specified in the CrlTemplate input parameter.

OldCrl (input)
The CRL to be updated with the new attribute values. The CRL must be unsigned and available for update.

Return Value

A pointer to the modified, unsigned CRL. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

2.5.9 CL_CrlSign

CSSM_DATA_PTR CSSMAPI CL_CrlSign (CSSM_CL_HANDLE CLHandle,
CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR UnsignedCrl,
const CSSM_DATA_PTR SignerCert,
const CSSM_FIELD_PTR SignScope,
uint32 ScopeSize)

This function signs, in accordance with the specified cryptographic context, the fields of the CRL indicated in the *SignScope* parameter.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

UnsignedCrl (input)
A pointer to the CSSM_DATA structure containing the CRL to be signed.

SignerCert (input)
A pointer to the CSSM_DATA structure containing the certificate to be used to sign the CRL.

SignScope (*input*)
A pointer to the CSSM_FIELD array containing the tag/value pairs of the fields to be signed. A NULL input signs all the fields in the CRL.

ScopeSize (*input*)
The number of entries in the sign scope list.

Return Value

A pointer to the CSSM_DATA structure containing the signed CRL. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CL_CrlVerify

2.5.10 CL_CrlVerify

```
CSSM_BOOL CSSMAPI CL_CrlVerify  (CSSM_CL_HANDLE CLHandle,
                                  CSSM_CC_HANDLE CCHandle,
                                  const CSSM_DATA_PTR SubjectCrl,
                                  const CSSM_DATA_PTR SignerCert,
                                  const CSSM_FIELD_PTR VerifyScope,
                                  uint32 ScopeSize)
```

This function verifies that the signed CRL has not been altered since it was signed by the designated signer. It does this by verifying the digital signature on the VerifyScope fields.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

SubjectCrl (input)
A pointer to the CSSM_DATA structure containing the CRL to be verified.

SignerCert (input)
A pointer to the CSSM_DATA structure containing the certificate used to sign the CRL.

VerifyScope (*input*)
A pointer to the CSSM_FIELD array containing the tag/value pairs of the fields to be verified. A NULL input verifies all the fields in the CRL.

ScopeSize (*input*)
The number of entries in the verify scope list.

Return Value

A CSSM_TRUE return value signifies that the CRL verifies successfully. When CSSM_FALSE is returned, either the CRL verified unsuccessfully or an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CL_CrlSign

2.5.11 CL_IsCertInCrl

```
CSSM_BOOL CSSMAPI CL_IsCertInCrl  (CSSM_CL_HANDLE CLHandle,
                                    const CSSM_DATA_PTR Cert,
                                    const CSSM_DATA_PTR Crl)
```

This function searches the CRL for a record corresponding to the certificate.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate to be located.

Crl (input)
A pointer to the CSSM_DATA structure containing the CRL to be searched.

Return Value

A CSSM_TRUE return value signifies that the certificate is in the CRL. When CSSM_FALSE is returned, either the certificate is not in the CRL or an error has occurred. Use CSSM_GetError to obtain the error code.

2.6 Extensibility Functions

The CL_PassThrough function is provided to allow CL developers to extend the certificate and CRL format-specific functionality of the KeyWorks API. Because it is only exposed to KeyWorks as a function pointer, its name internal to the CL can be assigned at the discretion of the CL module developer. However, its parameter list and return value must match what is shown below.

2.6.1 CL_PassThrough

```
void * CSSMAPI CL_PassThrough  (CSSM_CL_HANDLE CLHandle,
                                 CSSM_CC_HANDLE CCHandle,
                                 uint32 PassThroughId,
                                 const void * InputParams)
```

This function allows applications to call CL module-specific operations.

Parameters

CLHandle (input)
The handle that describes the service provider CL module used to perform this function.

CCHandle (input)
The handle that describes the context of the cryptographic operation.

PassThroughId (input)
An identifier assigned by the CL module to indicate the function to perform.

InputParams (input)
A pointer to a module, implementation-specific structure containing parameters to be interpreted in a function-specific manner by the requested CL module. This parameter can be used as a pointer to an array of void pointers.

Return Value

A pointer to a module, implementation-specific structure containing the output from the passthrough function. The output data must be interpreted by the calling application based on externally available information. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

Chapter 3. Certificate Library Function Examples

3.1 Attach/Detach Example

The Certificate Library (CL) module is responsible for performing certain operations when KeyWorks attaches to and detaches from it. These operations should be performed in a function called AddInAuthenticate, which must be exported by the module. The AddInAuthenticate function will be called by the framework when the module is loaded. The steps in Section 3.1.1 must be performed in order for the attach process to work properly.

In the code example in Section 3.1.1, it is assumed that the CSSM entry points, such as CSSM_RegisterServices, have been resolved at link time. If not, the module may call GetProcAddress to resolve the entry points.

3.1.1 AddInAuthenticate

```
#include “cssm.h”

/* global variables used for registration */
CSSM_REGISTRATION_INFO    reg_info;
CSSM_SPI_CL_FUNCS         cl_jmp_tbl;
CSSM_SPI_MEMORY_FUNCS     upcall_tbl = {NULL, NULL, NULL, NULL};
CSSM_MODULE_FUNCS         module_funcs;
CSSM_GUID cl_guid =
{ 0x83badc39, 0xfac1, 0x11da, { 0x81, 0x72, 0x0, 0xaa, 0x0, 0xb1, 0x99, 0xdd }
};

__declspec(dllexport) CSSM_RETURN AddInAuthenticate(char* cssmCredentialPath,
char* cssmSection)
{
      CSSM_RETURN ret_code;

      memset(&cl_jmp_tbl, 0, sizeof(cl_jmp_tbl) );

      cl_jmp_tbl.CertSign              = CertSign;
      cl_jmp_tbl.CertVerify            = CertVerify;
      cl_jmp_tbl.CertCreateTemplate    = CertCreateTemplate;
      cl_jmp_tbl.CertGetFirstFieldValue = CertGetFirstFieldValue;
      cl_jmp_tbl.CertGetKeyInfo        = CertGetKeyInfo;
      cl_jmp_tbl.CertGetAllFields            = CertGetAllFields;

      /* set up the module specific info for CSSM */
      memset(&module_funcs, 0, sizeof(module_funcs) );
      module_funcs.ServiceType = CSSM_SERVICE_CL;
      module_funcs.ClFuncs = &cl_jmp_tbl;

      /* set up the registration structure for CSSM */
      memset(&reg_info, 0, sizeof(reg_info) );
      reg_info.Initialize = Initialize;
      reg_info.Terminate =  Terminate;
      reg_info.ThreadSafe = CSSM_FALSE;
      reg_info.ServiceSummary = CSSM_SERVICE_CL;
      reg_info.NumberOfServiceTables = 1;
      reg_info.Services = &module_funcs;

      /* Register services with CSSM */
      ret_code = CSSM_RegisterServices(&cl_guid, &reg_info, &upcall_tbl,
NULL);
```

```
      return ret_code;
}
```

3.2 Certificate Operations Examples

This section contains sample implementations of certificate functions in the CL.

3.2.1 CL_CertCreateTemplate

```
/*-----------------------------------------------------------------------------
-
 * Name: CL_CertCreateTemplate
 *
 * Description:
 * This function allocates and initializes memory for a certificate
 * based on the input tag/values pairs. The returned certificate
 * must be signed using the CSSM_CL_CertSign function.
 *
 * Parameters:
 * CLHandle (input)       : A handle to a CL module.
 * CertTemplate (input)   : A pointer to an array of tag/value pairs
 *                          which identify the fields of the new certificate
 * NumberOfFields (input) : The length of the CertTemplate array
 *
 * Return value:
 * The new certificate
 *
 * Error Codes:
 * CSSM_CL_INVALID_CL_HANDLE
 * CSSM_CL_INVALID_FIELD_POINTER
 * CSSM_CL_INVALID_TEMPLATE
 * CSSM_CL_MEMORY_ERROR
 * CSSM_CL_UNSUPPORTED_OPERATION
 * CSSM_CL_CERT_CREATE_FAIL
*----------------------------------------------------------------------------*/
CSSM_DATA_PTR   CSSMAPI CL_CertCreateTemplate  (CSSM_CL_HANDLE CLHandle,
                                               const CSSM_FIELD_PTR
CertTemplate,
                                                uint32 NumberOfFields)
{
    /* Initializations */
    CSSM_CERTIFICATE_PTR cert_ptr = NULL;
    CSSM_DATA_PTR  packed_cert_ptr = NULL;
    CSSM_ERROR_PTR err_ptr = NULL;
    uint32 i=0;

    /* Check inputs */
      /* Check that this is a valid CLHandle */
    if (CLHandle == 0)
    {
        CSSM_SetError(&my_clm_guid, CSSM_CL_INVALID_CL_HANDLE);
        return NULL;
    }
        /* Check that the NumberOfFields is greater than 0
           and that the CertTemplate pointer is not NULL */
    if ( !NumberOfFields || !CertTemplate)
    {
        CSSM_SetError(&my_clm_guid, CSSM_CL_INVALID_TEMPLATE);
        return NULL;
    }
        /* Check that CertTemplate is a valid pointer */
    if (cssm_IsBadReadPtr (CertTemplate, NumberOfFields*sizeof(CSSM_FIELD)) ||
        cssm_IsBadReadPtr(CertTemplate[NumberOfFields-1].FieldValue.Data,
            CertTemplate[NumberOfFields-1].FieldValue.Length) ||
        cssm_IsBadReadPtr(CertTemplate[NumberOfFields-1].FieldOid.Data,
            CertTemplate[NumberOfFields-1].FieldOid.Length) )
    {
        CSSM_SetError(&my_clm_guid, CSSM_CL_INVALID_TEMPLATE);
        return NULL;
    }

    /* Allocate a new certificate structure */
    cert_ptr = UpcallTable.malloc_func(CLHandle, sizeof(CSSM_CERTIFICATE));
    if (cert_ptr == NULL)
    {
        CSSM_SetError(&my_clm_guid, CSSM_CL_MEMORY_ERROR);
        return NULL;
    }
    memset(cert_ptr, 0, sizeof(CSSM_CERTIFICATE));

    /* Loop through the CertTemplate array */
    for( i=0; i < NumberOfFields; i++ )
    {
        /* Check that this field contains a valid data pointer */
        if ( !cl_IsBadReadPtr (CertTemplate[i].FieldValue.Data,
                            CertTemplate[i].FieldValue.Length))
        {
            /* If so, copy the data into the certificate structure */
            /* Add CL module-specific code here */
        }
        else
        {
            CSSM_SetError(&my_clm_guid, CSSM_CL_INVALID_FIELD_POINTER);
            /* Free the certificate structure */
            return NULL;
        }
    }

    /* Add internal, CL-generated certificate information */
    /* Add CL module-specific code here */

    /* If there are signatures on this cert, delete them */
    /* A newly created cert is assumed to be unsigned */
    /* Add CL module-specific code here */

    /* Pack the new certificate */
    /* The pack routine will allocate memory for the new cert using the
       application’s memory allocation routines */
    packed_cert_ptr = cl_PackCertificate(cert_ptr);

    /* Cleanup */
        /* Free the certificate structure */

    /* Return the packed certificate */
    return packed_cert_ptr;
};
```

3.3 CRL Operations Examples

This section contains sample implementations of Certificate Revocation List (CRL) functions in the CL.

3.3.1 CL_CrlAddCert

```
/*------------------------------------------------------------------------------
-
* Name: CL_CrlAddCert
 *
 * Description:
 * This function revokes the input certificate by adding a record representing
 * the certificate to the CRL. It uses the revoker’s certificate to sign the
 * new record in the CRL.  The reason for revoking the certificate may also be
 * stored in the revocation record.
 *
 * Parameters:
 * CLHandle (input)       : Handle to the CL module
 * CCHandle (input)       : Handle to the cryptographic context
 * Cert (input)           : A pointer to the CSSM_DATA structure containing
                            the certificate to be revoked
 * RevokerCert (input)    : A pointer to the CSSM_DATA structure containing
                            the revoker’s certificate
 * RevokeReason (input)   : The reason for revoking the certificate
 * OldCrl (input)         : A pointer to the CSSM_DATA structure containing
                            the CRL to which the newly revoked certificate
                            will be added
 *
 * Return value:
 * The updated CRL
 *
 * Error Codes:
 * CSSM_CL_INVALID_CL_HANDLE
 * CSSM_CL_INVALID_CC_HANDLE
 * CSSM_CL_INVALID_CERTIFICATE_PTR
 * CSSM_CL_INVALID_CRL
 * CSSM_CL_MEMORY_ERROR
 * CSSM_CL_CRL_ADD_CERT_FAIL
*-----------------------------------------------------------------------------*/
CSSM_DATA_PTR   CSSMAPI CL_CrlAddCert    (CSSM_CL_HANDLE CLHandle,
                                         CSSM_CC_HANDLE CCHandle,
                                         const CSSM_DATA_PTR Cert,
                                         const CSSM_DATA_PTR RevokerCert,
                                          CSSM_REVOKE_REASON RevokeReason,
                                         const CSSM_DATA_PTR OldCrl)
{
      CSSM_REVOCATION_LIST_PTR new_crl_ptr = NULL;
      CSSM_DATA_PTR new_crl_data_ptr = NULL;
      CSSM_DATA_PTR sign_data_ptr = NULL;
      CSSM_REVOKED_CERT_PTR new_revoked_cert_ptr = NULL;
      CSSM_REVOKED_CERT_PTR temp_revoked_cert_ptr = NULL;
      CSSM_REVOKED_CERT_PTR prev_revoked_cert_ptr = NULL;

      CSSM_CERTIFICATE_PTR revoker_cert_ptr = NULL;
      CSSM_CERTIFICATE_PTR cert_ptr = NULL;
      uint32 signature_size;
      CSSM_DATA_PTR signature_data_ptr = NULL;
      CSSM_CONTEXT_PTR context_ptr = NULL;
      CSSM_RETURN ret;

      /* Check inputs */
```

```
        if(CLHandle == 0)
        {
              CSSM_SetError(&my_clm_guid,CSSM_CL_INVALID_CL_HANDLE);
              return NULL;
        }
        if(CCHandle == 0)
        {
              CSSM_SetError(&my_clm_guid,CSSM_CL_INVALID_CC_HANDLE);
              return NULL;
        }
        if(Cert == NULL)
        {
              CSSM_SetError(&my_clm_guid,CSSM_CL_INVALID_CERT_POINTER);
              return NULL;
        }
        if(Cert != NULL && cssm_IsBadReadPtr(Cert, sizeof(CSSM_DATA)) )
        {
              CSSM_SetError(&my_clm_guid, CSSM_CL_INVALID_DATA_POINTER);
              return NULL;
        }
        if(Cert->Length != 0 && cssm_IsBadReadPtr(Cert->Data,Cert->Length))
        {
              CSSM_SetError(&my_clm_guid, CSSM_CL_INVALID_CERT_POINTER);
              return NULL;
        }

        if(RevokerCert == NULL)
        {
              CSSM_SetError(&my_clm_guid,CSSM_CL_INVALID_REVOKER_CERT_PTR);
              return NULL;
        }
        if(RevokerCert->Length != 0 && cssm_IsBadReadPtr(RevokerCert->Data,RevokerCert-
>Length))
        {
              CSSM_SetError(&my_clm_guid, CSSM_CL_INVALID_REVOKER_CERT_PTR);
              return NULL;
        }
        if(OldCrl == NULL)
        {
              CSSM_SetError(&my_clm_guid, CSSM_CL_INVALID_CRL_PTR);
              return NULL;
        }
        if(cssm_IsBadReadPtr(OldCrl, sizeof(CSSM_DATA)))
        {
              CSSM_SetError(&my_clm_guid, CSSM_CL_INVALID_CRL_PTR);
              return NULL;
        }
        if(OldCrl->Length != 0 && !cssm_IsBadReadPtr(OldCrl->Data, OldCrl-
>Length))
        {
              /* Unpack the CRL */
              new_crl_ptr = cl_UnPackCrl(CLHandle,&MemoryFunctions,OldCrl);
              if(new_crl_ptr == NULL)
              {
                    CSSM_SetError(&my_clm_guid, CSSM_CL_MEMORY_ERROR);
                    return NULL;
              }

              /* remove the crl signature, if necessary */
              /* unpack the revoker's certificate */
              revoker_cert_ptr =
cl_UnpackCertificate(CLHandle,&MemoryFunctions,RevokerCert);
              if(revoker_cert_ptr == NULL)
```

```
            {
                  /* Cleanup */
                  CSSM_SetError(&my_clm_guid, CSSM_CL_MEMORY_ERROR);
                  return NULL;
            }
            /* unpack the certificate to be revoked */
            cert_ptr = cl_UnpackCertificate(CLHandle,&MemoryFunctions,Cert);;
            if(cert_ptr == NULL)
            {
                  /* Cleanup */
                  CSSM_SetError(&my_clm_guid, CSSM_CL_MEMORY_ERROR);
                  return NULL;
            }

            /* Create the revoked certificate structure to be placed in the CRL
      */
            /* Add any revocation record specific information,
               such as the time of revocation and the revocation reason */
            /* Sign the revoked certificate structure using the revoker’s certificate */
      }

      /* Add the new revocation record to the CRL */

      /* Pack the new CRL */
      new_crl_data_ptr = cl_PackCrl(CLHandle,&MemoryFunctions,new_crl_ptr);

      /* Cleanup & Return */
      return new_crl_data_ptr;
}
```

3.4 Extensibility Functions Examples

3.4.1 CL_PassThrough

In this example, the pack and unpack routines that are used internally to the CL module are exposed for use by applications through the passthrough mechanism.

```
typedef enum cl_custom_function_id {
    CL_CUSTOMID_PACK_CERTIFICATE = 0,
    CL_CUSTOMID_UNPACK_CERTIFICATE     = 1,
} CL_CUSTOM_FUNCTION_ID;

/*-----------------------------------------------------------------------------
-
 * Name: CL_PassThrough
 *
 * Description:
 * This function allows applications to call KeyWorks CL module-specific
 * operations. The KeyWorks CL module-specific operations include:
 *    cl_PackCertificate
 *    cl_UnpackCertificate
 *
 * Parameters:
 * CCHandle (input)       : Handle identifying a Cryptographic Context which
 *                          may be used by the passthrough function
 * PassThroughId (input)  : An identifier assigned by the KeyWorks CL module
 *                          to indicate the exported function to perform.
 * InputParams (input)    : Parameters to be interpreted in a
 *                          function-specific manner by the KeyWorks CL
module.
 *
 * Return value:
 * Output from the passthrough function.
 * The output data must be interpreted by the calling application
 * based on externally available information.
 *
 * Error Codes:
 * CSSM_CL_INVALID_CL_HANDLE
 * CSSM_CL_INVALID_CC_HANDLE
 * CSSM_CL_INVALID_DATA_POINTER
 * CSSM_CL_UNSUPPORTED_OPERATION
 * CSSM_CL_PASS_THROUGH_FAIL
 *----------------------------------------------------------------------------
*/
CSSM_DATA_PTR CSSMAPI CL_PassThrough   (CSSM_CL_HANDLE CLHandle,
                                        CSSM_CC_HANDLE CCHandle,
                                        uint32 PassThroughId,
                                        const CSSM_DATA_PTR InputParams)
{
    /* Initializations */
    /* Check inputs */
        /* Check that this is a recognized PassThroughId */
    /* Call the requested function */
    switch ( PassThroughId ) {
    case CL_CUSTOMID_PACK_CERTIFICATE:
      return cl_PackCertificate( InputParams );
    case CL_CUSTOMID_UNPACK_CERTIFICATE:
      return cl_UnpackCertificate( InputParams );
    default:

       CSSM_SetError(&my_clm_guid, CSSM_CL_UNSUPPORTED_OPERATION);
```

```
        return NULL;
    }
};
```

Appendix A. IBM KeyWorks Errors

The error codes in this section constitute the generic error codes that are defined by KeyWorks for use by all Certificate Libraries (CLs) in describing common error conditions. A CL may also define and return vendor-specific error codes. The error codes defined by KeyWorks are considered to be comprehensive and few if any vendor-specific codes should be required. Applications must consult vendor-supplied documentation for the specification and description of any error codes defined outside of this specification.

All CL service provider interface (SPI) functions return one of the following:

- CSSM_RETURN **-** An enumerated type consisting of CSSM_OK and CSSM_FAIL. If it is CSSM_FAIL, an error code indicating the reason for failure can be obtained by calling CSSM_GetError.
- CSSM_BOOL **-** KeyWorks functions returning this data type return either CSSM_TRUE or CSSM_FALSE. If the function returns CSSM_FALSE, an error code may be available (but not always) by calling CSSM_GetError.
- A pointer to a data structure, a handle, a file size, or whatever is logical for the function to return. An error code may be available (but not always) by calling CSSM_GetError.

The information returned from CSSM_GetError includes both the error number and a Globally Unique ID (GUID) that associates the error with the module that set it. Each module must have a mechanism for reporting their errors to the calling application. In general, there are two types of errors a module can return, including:

- Errors defined by KeyWorks that are common to a particular type of service provider module
- Errors reserved for use by individual service provider modules

Since some errors are predefined by KeyWorks, those errors have a set of predefined numeric values that are reserved by KeyWorks, and cannot be redefined by modules. For errors that are particular to a module, a different set of predefined values has been reserved for their use. Table 6 lists the range of error numbers defined by KeyWorks for CL modules and those available for use in individual Cryptographic Service Provider (CSP) modules.

Table 6. CL Module Error Numbers

Error Number Range	**Description**
3000 – 3999	CL errors defined by KeyWorks
4000 – 4999	CL errors reserved for individual CL modules

The calling application must determine how to handle the error returned by CSSM_GetError. Detailed descriptions of the error values will be available in the corresponding specification, the cssmerr.h header file, and the documentation for specific modules. If a routine does not know how to handle the error, it may choose to pass the error to its caller.

A.1. Certificate Library Module Errors

Table 7. Certificate Library Errors

Error Code	Error Name
3001	CSSM_CL_UNKNOWN_FORMAT
3002	CSSM_CL_UNKNOWN_TAG
3003	CSSM_CL_INVALID_CONTEXT
3004	CSSM_CL_INVALID_CL_HANDLE
3005	CSSM_CL_INVALID_CC_HANDLE
3006	CSSM_CL_INVALID_CERT_POINTER
3007	CSSM_CL_INVALID_FIELD_POINTER
3008	CSSM_CL_INVALID_TEMPLATE
3009	CSSM_CL_INVALID_DATA_POINTER
3010	CSSM_CL_INVALID_SCOPE
3012	CSSM_CL_CERT_CREATE_FAIL
3013	CSSM_CL_CERT_VIEW_FAIL
3014	CSSM_CL_CERT_GET_FIELD_VALUE_FAIL
3015	CSSM_CL_CERT_GET_KEY_INFO_FAIL
3016	CSSM_CL_CERT_IMPORT_FAIL
3017	CSSM_CL_CERT_EXPORT_FAIL
3018	CSSM_CL_PASS_THROUGH_FAIL
3019	CSSM_CL_CERT_DESCRIBE_FORMAT_FAIL
3020	CSSM_CL_UNSUPPORTED_OPERATION
3021	CSSM_CL_MEMORY_ERROR
3022	CSSM_CL_CERT_SIGN_FAIL
3023	CSSM_CL_CERT_UNSIGN_FAIL
3024	CSSM_CL_CERT_VERIFY_FAIL
3025	CSSM_CL_RESULTS_HANDLE
3026	CSSM_CL_INVALID_SIGNER_CERTIFICATE
3027	CSSM_CL_NO_FIELD_VALUES
3028	CSSM_CL_INVALID_CRL_PTR
3029	CSSM_CL_CERT_ABORT_QUERY_FAIL
3030	CSSM_CL_CRL_CREATE_FAIL
3031	CSSM_CL_CRL_SET_FAIL
3032	CSSM_CL_CRL_ADD_CERT_FAIL
3033	CSSM_CL_CRL_REMOVE_CERT_FAIL
3034	CSSM_CL_CRL_SIGN_FAIL
3035	CSSM_CL_CRL_VERIFY_FAIL
3036	CSSM_CL_IS_CERT_IN_CRL_FAIL
3037	CSSM_CL_CRL_GET_FIELD_VALUE_FAIL
3038	CSSM_CL_CRL_ABORT_QUERY_FAIL
3039	CSSM_CL_CRL_DESCRIBE_FORMAT_FAIL
3040	CSSM_CL_INVALID_POINTER
3041	CSSM_CL_INVALID_DATA
3042	CSSM_CL_INITIALIZE_FAIL
3100	CSSM_CL_SIG_NOT_IN_CERT
3101	CSSM_CL_INVALID_REVOKER_CERT_PTR
3102	CSSM_CL_NO_REVOKED_CERTS_IN_CRL
3103	CSSM_CL_CERT_NOT_FOUND_IN_CRL
3104	CSSM_CL_CRL_SIGNSCOPE_NOT_SUPPORTED
3105	CSSM_CL_CRL_VERIFYSCOPE_NOT_SUPPORTED
3106	CSSM_CL_CRL_NOT_SIGNEDBY_SIGNER

Error Code	**Error Name**
3107	CSSM_CL_CRL_NO_FIELD_OID
3108	CSSM_CL_INVALID_REVOKED_CERT_PTR
3109	CSSM_CL_INVALID_INPUT_PTR
3110	CSSM_CL_KEY_ALGID_NOT_SUPPORTED
3111	CSSM_CL_GET_KEY_ATTRIBUTE_FAIL
3112	CSSM_CL_CERT_ENCODE_FAIL
3113	CSSM_CL_CERT_DECODE_FAIL
3114	CSSM_CL_SIGNATURE_ALGID_NOT_SUPPORTED
3115	CSSM_CL_KEY_FORMAT_UNKNOWN
3116	CSSM_CL_INVALID_CERT_ISSUER_NAME
3117	CSSM_CL_INVALID_CERT_SUBJECT_NAME
3118	CSSM_CL_MISSING_CERT_SUBJECT_NAME
3119	CSSM_CL_MISSING_CERT_ISSUER_NAME
3120	CSSM_CL_MISSING_CERT_VALIDITY
3121	CSSM_CL_MISSING_SUBJECT_PUB_KEY
3122	CSSM_CL_FIELD_NOT_PRESENT

Appendix B.   List of Acronyms

API	Application Programming Interface
CA	Certificate Authority
CL	Certificate Library
CLI	Certificate Library Interface
CRL	Certificate Revocation List
CSP	Cryptographic Service Provider
DES	Data Encryption Standard
DL	Data Storage Library
DLL	Dynamically Linked Library
DSA	Digital Signature Algorithm
GUID	Globally Unique ID
ISV	Independent Software Vendor
KRF	Key Recovery Field
KRSP	Key Recovery Service Provider
MAC	Message Authentication Code
OID	Object Identifier
SDSI	Simple Distributed Security Infrastructure
SPI	Service Provider Interface
TP	Trust Policy

Appendix C. 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.
Authentication Information	Information that is verified for authentication. For example, a Key Recovery Officer (KRO) selects a password which will be used for authentication with the Key Recovery Coordinator (KRC). A KRO operator who has identification information must search the Authentication Information (AI) database to locate an AI value that corresponds to the individual who generated the information.
Certificate	See Digital certificate.
Certificate Authority	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 (CA). CAs issue, verify, and revoke certificates.
Certificate chain	The hierarchical chain of all the other certificates used to sign the current certificate. This includes the 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 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 CA may issue a new certificate.
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. IBM KeyWorks accommodates Data Encryption Standard (DES), RC2, RC4, International Data Encryption Algorithm (IDEA), and other encryption algorithms.
Cryptographic Service Provider	Cryptographic Service Providers (CSPs) are 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
Cryptography	The science of keeping data secure. Cryptography provides the ability to store information or to communicate between parties in such a way that prevents other non-involved parties from understanding the stored information or accessing and understanding the communication. The encryption process takes understandable text and transforms it into an unintelligible piece of data (called ciphertext); the decryption process restores the understandable text from the unintelligible data. Both involve a mathematical formula or algorithm and a secret sequence of data called a key. Cryptographic services provide confidentiality (keeping data secret), integrity (preventing data from being modified), authentication (proving the identity of a resource or a user), and non-repudiation (providing proof that a message or transaction was sent and/or received). There are two types of cryptography: • In shared/secret key (symmetric) cryptography there is only one key that is a shared secret between the two communicating parties. The same key is used for encryption and decryption. • In public key (asymmetric) cryptography different keys are used for encryption and decryption. A party has two keys: a public key and a private key. The two keys are mathematically related, but it is virtually impossible to derive the private key from the public key. A message that is encrypted with someone's public key (obtained from some public directory) can be decrypted with the associated private key. Alternately, the private key can be used to "sign" a document; the public key can be used as verification of the source of the document.
Cryptoki	Short for cryptographic token interface. See Token.
Data Encryption Standard	In computer security, the National Institute of Standards and Technology (NITS) Data Encryption Standard (DES), adopted by the U.S. Government as Federal Information Processing Standard (FIPS) Publication 46, which allows only hardware implementations of the data encryption algorithm.
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 unforgettable 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 has not 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.
Enterprise	A company or individual who is authorized to submit on-line requests to the Key Recovery Officer (KRO). In the enterprise key recovery scenario, a process at the enterprise called the KRO is responsible for preparing key recovery requests and communicating them to the KRC. The KRO, acting on behalf of an enterprise or individual, sends an on-line request to the Key Recovery Coordinator (KRC) to recover a key from a Key Recovery Block (KRB).
Graphical User Interface	A type of display format that enables the user to choose commands, start programs, and see lists of files and other options by pointing to pictorial representations (icons) and lists of menu items on the screen. Graphical User Interfaces (GUIs) are used by the Microsoft Windows program for IBM-compatible microcomputers and by other systems.
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.
IBM KeyWorks Architecture	A set of layered security services that address communications and data security problems in the emerging PC business space.
IBM KeyWorks Framework	The IBM KeyWorks Framework 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 IBM KeyWorks binds together all the security services required by PC applications. In particular, it facilitates linking digital certificates to cryptographic actions and trust protocols.
Key Escrow	The storing of a key (or parts of a key) with a trusted party or trusted parties in case of loss or destruction of the key.
Key Recovery Agent	The Key Recovery Agent (KRA) acts as the back end for a key recovery operation. The KRA can only be accessed through an on-line communication protocol via the Key Recovery Coordinator (KRC). KRAs are considered outside parties involved in the key recovery process; they are analogous to the neighbors who each hold one digit of the combination of the lock box

	containing the key. The authorized parties (i.e., enterprise or law enforcement) have the freedom to choose the number of specific KRAs that they want to use. The authorized party requests that each KRA decrypt its section of the Key Recovery Fields (KRFs) that is associated with the transmission. Then those pieces of information are used in the process that derives the session key. The KRA 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. The number of KRAs on each end of the communication does not have to be equal.
Key Recovery Block	The Key Recovery Block (KRB) is a piece of encrypted information that is contained within a block. The KRS components (i.e., KRO, KRC, KRA) work collectively to recover a session key from a provided KRB. In the enterprise scenario, the KRO has both the KRB and the credentials that authenticate it to receive the recovered key. This information will be transmitted over the network to the KRC. In the law enforcement scenario, the KRB is presented on a 3.5-inch diskette, and the credentials are in the physical form of a legal warrant. This warrant will specify any information available to the law enforcement agents which can be used to tie the warrant to the identity of the user for whom KRBs were generated (i.e., username, hostname, IP address). The KRC has the ability to check credentials and derive the original encryption key from the KRB with the help of its KRAs.
Key Recovery Coordinator	The Key Recovery Coordinator (KRC) acts as the front end for the key recovery operation. The KRO, acting on behalf of an enterprise or individual, sends an on-line request to the KRC to recover a key from a KRB. The KRC receives the on-line request and services it by interacting with the appropriate set of KRAs as specified within the KRB. The recovered key is then sent back to the KRO by the KRC using an on-line protocol. The KRC consists of one main application which, when started, behaves as a server process. The system, which serves as the KRC, may be configured to start the KRC application as part of system services; alternatively, the KRC operator can start up the KRC application manually. The KRC application performs the following operations: • Listens for on-line recovery requests from KRO • Can be used to launch an embedded application that allows manual key recovery for law enforcement • Monitors and displays the status of the recovery requests being serviced
Key Recovery Field	A Key Recovery Field (KRF) is a block of data that is created from a symmetric key and key recovery profile information. The Key Recovery Service Provider (KRSP) is invoked from the IBM KeyWorks framework to create the KRFs. There are two major pieces of the KRFs: block 1 contains information that is unrelated to the session key of the transmitted message, and encrypted with the public keys of the selected key recovery agents; block 2 contains information that is related to the session key of the transmission. The KRSP generates the KRFs for the session key. This information is *not* the key or any portion of the key, but is information that can be used to recover the key. The KRSP has access to location-unique jurisdiction policy information that controls and modifies some of the steps in the generation of the KRFs. Only once the KRFs are generated, and both the client and server sides have access to them, can the encrypted message flow begin. KRFs are generated so that they can be used by a 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.
Key Recovery Module Manager	The Key Recovery Module Manager enables key recovery for cryptographic services obtained through the IBM KeyWorks. It mediates all cryptographic services provided by the IBM KeyWorks and applies the appropriate key recovery policy on all such operations. The Key Recovery Module Manager contains a Key Recovery Policy Table (KRPT) that defines the applicable key recovery policy for all cryptographic products. The Key Recovery Module Manager routes the KR-API function calls made by an application to the appropriate KR-SPI functions. The Key Recovery Module Manager also enforces the key recovery policy on all cryptographic operations that are obtained through the IBM KeyWorks. It maintains key recovery state in the form of key recovery contexts.
Key Recovery Officer	An entity called the Key Recovery Officer (KRO) is the focal point of the key recovery process. In the enterprise key recovery scenario, the KRO is responsible for preparing key recovery requests and communicating them to the KRC. The KRO has both the KRB and the credentials that authenticate it to receive the recovered key. The KRO is the entity that acts on behalf of an enterprise to initiate a key recovery request operation. An employee within an enterprise who desires key recovery will send a request to the KRO with the KRB that is to be recovered. The actual key recovery phase begins when the KRO operator uses the KRO application to initiate a key recovery request to the appropriate KRC. At this time, the operator selects a KRB to be sent for recovery, enters the Authentication Information (AI) information that can be used to authenticate the request to the KRC, and submits the request.
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 interoperability policies.* Key recovery enablement policies specify the exact cryptographic protocol suites (e.g., 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 interoperability policies specify to what degree a key recovery enabled cryptographic product is allowed to interoperate with other cryptographic products.

Key Recovery Server	The Key Recovery Server (KRS) consists of three major entities: Key Recovery Coordinator (KRC), Key Recovery Agent (KRA), and Key Recovery Officer (KRO). The KRS is intended to be used by enterprise employees and security personnel, law enforcement personnel, and KRSF personnel. The KRS interacts with one or more local or remote KRAs to reconstruct the secret key that can be used to decrypt the ciphertext.
Key Recovery Server Facility	The Key Recovery Server Facility (KRSF) is a facility room that houses the KRS component facilities, ensuring they operate within a secure environment that is highly resistant to penetration and compromise. Several physical and administrative security procedures must be followed at the KRSF such as a combination keyed lock, limited personnel, standalone system, operating system with security features (Microsoft NT Workstation 4.0), NTFS (Windows NT Filesystem), and account and auditing policies.
Key Recovery Service Provider	Key Recovery Service Providers (KRSPs) are modules that provide key recovery enablement functions. The cryptographic functions provided may include: • Key recovery field generation • Key recovery field processing
Law Enforcement	A type of scenario where key recovery is mandated by the jurisdictional law enforcement authorities in the interest of national security and law enforcement. In the law enforcement scenario, the Key Recovery Block (KRB) is presented on a 3.5-inch diskette, and the credentials are in the physical form of a legal warrant. This warrant will specify any information available to the law enforcement agents which can be used to tie the warrant to the identity of the user for whom KRBs were generated (i.e., username, hostname, IP address).
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.
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 is used to decipher messages in public-key cryptography. This key is kept secret by its owner.
Public key	The cryptographic key is used to encrypt messages in public-key cryptography. The public key is available to multiple users (i.e., the public).
Random number generator	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 (CA) has a self-signed root certificate. The root certificate's public key is the foundation of signature verification in its domain.
Secure Electronic Transaction	A mechanism for securely and automatically routing payment information among users, merchants, and their banks. Secure Electronic Transaction (SET) is a protocol for securing bankcard transactions on the Internet or other open networks using cryptographic services. SET is a specification designed to utilize technology for authenticating parties involved in payment card purchases on any type of on-line network, including the Internet. SET was developed by Visa and MasterCard, with participation from leading technology companies, including Microsoft, IBM, Netscape, SAIC, GTE, RSA, Terisa Systems, and VeriSign. By using sophisticated cryptographic techniques, SET will make cyberspace a safer place for conducting business and is expected to boost consumer confidence in electronic commerce. SET focuses on maintaining confidentiality of information, ensuring message integrity, and authenticating the parties involved in a transaction. The significance of SET, over existing Internet security protocols, is found in the use of digital certificates, Digital certificates will be used to authenticate all the parties involved in a transaction. SET will provide those in the virtual world with the same level of trust and confidence a consumer has today when making a purchase at any of the 13 million Visa-acceptance locations in the physical world. The SET specification is open and free to anyone who wishes to use it to develop SET-compliant software for buying or selling in cyberspace.
Security Context	A control structure that retains state information shared between a CSP 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.

Smart Card	A device (usually similar in size to a credit card) that contains an embedded microprocessor that could be used to store information. Smart cards can store credentials used to authenticate the holder.
S/MIME	Secure/Multipurpose Internet Mail Extensions (S/MIME) is a protocol that adds digital signatures and encryption to Internet MIME messages. MIME is the official proposed standard format for extended Internet electronic mail. Internet e-mail messages consist of two parts, the header and the body. The header forms a collection of field/value pairs structured to provide information essential for the transmission of the message. The body is normally unstructured unless the e-mail is in MIME format. MIME defines how the body of an e-mail message is structured. The MIME format permits e-mail to include enhanced text, graphics, audio, and more in a standardized manner via MIME-compliant mail systems. However, MIME itself does not provide any security services. The purpose of S/MIME is to define such services, following the syntax given in PKCS #7 for digital signatures and encryption. The MIME body part carries a PKCS #7 message, which itself is the result of cryptographic processing on other MIME body parts.
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 Data Encryption Standard (DES) and International Data Encryption Algorithm (IDEA). The U.S. Government endorsed DES as a standard in 1977. It is 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, 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 smart cards and Personal Computer Memory Card International Association (PCMCIA) 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.