IBM

IBM KeyWorks Toolkit

Data Storage Library Interface (DLI) Specification

Table of Contents

CHAPTER 1. INTRODUCTION 1

1.1 SERVICE PROVIDER MODULES 1
1.2 INTENDED AUDIENCE 2
1.3 DOCUMENTATION SET 2
1.4 REFERENCES 3

CHAPTER 2. DATA STORAGE LIBRARY INTERFACE 5

2.1 CATEGORIES OF OPERATIONS 6
2.2 DATA STORAGE LIBRARY OPERATIONS 7
2.3 DATA STORAGE OPERATIONS 7
2.4 DATA RECORD OPERATIONS 8
2.5 EXTENSIBILITY FUNCTIONS 9
2.6 DATA STRUCTURES 9
2.6.1 CSSM_BOOL 9
2.6.2 CSSM_DATA 9
2.6.3 CSSM_DB_ACCESS_TYPE 9
2.6.4 CSSM_DB_ATTRIBUTE_DATA 10
2.6.5 CSSM_DB_ATTRIBUTE_INFO 10
2.6.6 CSSM_DB_ATTRIBUTE_NAME_FORMAT 11
2.6.7 CSSM_DB_CERTRECORD_SEMANTICS 11
2.6.8 CSSM_DB_CONJUNCTIVE 11
2.6.9 CSSM_DB_HANDLE 11
2.6.10 CSSM_DB_INDEX_INFO 11
2.6.11 CSSM_DB_INDEX_TYPE 12
2.6.12 CSSM_DBINFO 12
2.6.13 CSSM_DB_OPERATOR 13
2.6.14 CSSM_DB_PARSING_MODULE_INFO 14
2.6.15 CSSM_DB_RECORD_ATTRIBUTE_DATA 14
2.6.16 CSSM_DB_RECORD_ATTRIBUTE_INFO 14
2.6.17 CSSM_DB_RECORD_INDEX_INFO 15
2.6.18 CSSM_DB_RECORD_PARSING_FNTABLE 15
2.6.19 CSSM_DB_RECORDTYPE 16
2.6.20 CSSM_DB_UNIQUE_RECORD 16
2.6.21 CSSM_DL_DB_HANDLE 16
2.6.22 CSSM_DL_DB_LIST 16
2.6.23 CSSM_DL_CUSTOM_ATTRIBUTES 17
2.6.24 CSSM_DL_FFS_ATTRIBUTES 17
2.6.25 CSSM_DL_HANDLE 17
2.6.26 CSSM_DL_LDAP_ATTRIBUTES 17
2.6.27 CSSM_DL_ODBC_ATTRIBUTES 17
2.6.28 CSSM_DL_PKCS11_ATTRIBUTES 17
2.6.29 CSSM_DLSUBSERVICE 18
2.6.30 CSSM_DLTYPE 19
2.6.31 CSSM_DL_WRAPPEDPRODUCTINFO 20
2.6.32 CSSM_NAME_LIST 20
2.6.33 CSSM_QUERY 20
2.6.34 CSSM_QUERY_LIMITS 21
2.6.35 CSSM_SELECTION_PREDICATE 21
2.7 DATA STORAGE OPERATIONS 22
2.7.1 DL_Authenticate 22
2.7.2 DL_DbClose 23
2.7.3 DL_DbCreate 24

2.7.4 DL_DbDelete *25*
2.7.5 DL_DbExport *26*
2.7.6 DL_GetDbNameFromHandle *27*
2.7.7 DL_DbGetRecordParsingFunctions *28*
2.7.8 DL_DbImport *29*
2.7.9 DL_DbOpen *31*
2.7.10 DL_DbSetRecordParsingFunctions *32*
2.8 DATA RECORD OPERATIONS 33
2.8.1 DL_DataAbortQuery *33*
2.8.2 DL_DataDelete *34*
2.8.3 DL_DataGetFirst *35*
2.8.4 DL_DataGetNext *37*
2.8.5 DL_DataInsert *38*
2.8.6 DL_FreeUniqueRecord *39*
2.9 EXTENSIBILITY FUNCTIONS 40
2.9.1 DL_PassThrough *40*

CHAPTER 3. DATA STORAGE LIBRARY FUNCTION EXAMPLES 41

3.1 ATTACH/DETACH EXAMPLE 41
3.1.1 AddInAuthenticate *41*
3.2 DATA STORE OPERATIONS EXAMPLE 43

APPENDIX A. IBM KEYWORKS ERRORS 44

APPENDIX B. LIST OF ACRONYMS 47

APPENDIX C. GLOSSARY 48

List of Figures

Figure 1. IBM KeyWorks Toolkit Architecture 2

List of Tables

Table 1. DL Module Error Numbers 44
Table 2. Data Storage Errors 45

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. These service providers will be provided by Independent Software Vendors (ISVs) and hardware vendors. 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.

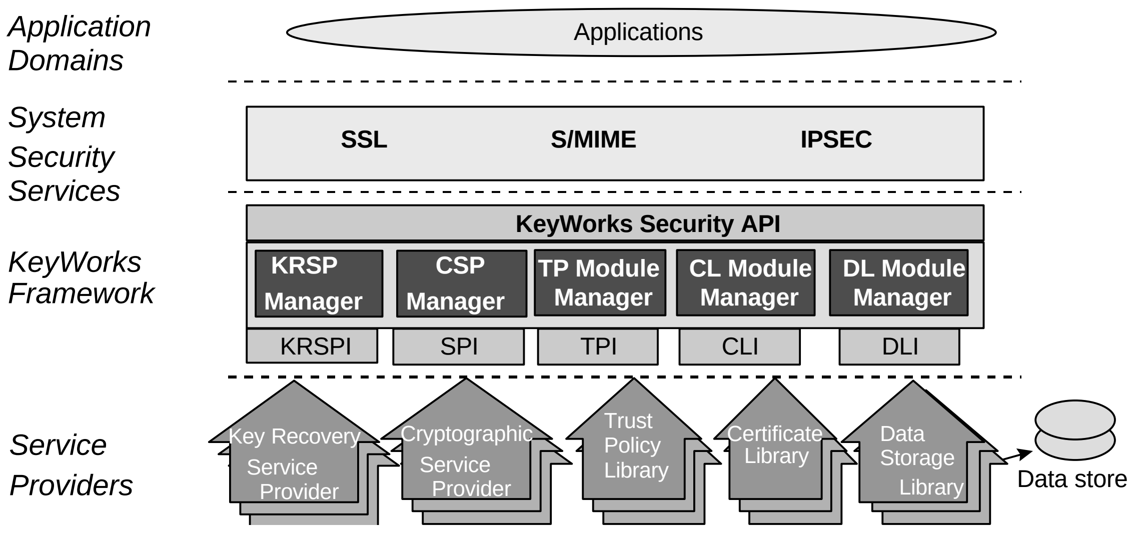

Figure 1. IBM KeyWorks Toolkit Architecture

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 is provided in the *IBM KeyWorks Service Provider Module Structure & Administration Specification*.

1.2 Intended Audience

This document should be used by ISVs who want to develop their own TP service provider modules. 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
  This 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. Data Storage Library Interface

A module with Data Storage Library (DL) services provides access to persistent data stores of certificates, Certificate Revocation Lists (CRLs), keys, policies, and other security-related objects. Stable storage can be provided by a:

- Commercially available database management system (DBMS) product
- Directory service
- Custom hardware-based storage device
- Native file system

The implementation of DL operations should be semantically free. For example, a DL operation that inserts a trusted X.509 certificate into a data store should not be responsible for verifying the trust on that certificate. The semantic interpretation of security objects should be implemented in Trust Policy (TP) services, layered services, and applications.

The DL provides access to persistent stores of security-related objects by translating calls from the Data Storage Library Interface (DLI) into the native interface of the data store. The native interface of the data store may be that of a DBMS package, a directory service, a custom storage device, or a traditional local or remote file system. Applications are able to obtain information about the available DL services by using the CSSM_GetModuleInfo function to query the IBM KeyWorks registry. The information about the DL service includes the following:

- Vendor information - Information about the module vendor, a text description of the DL and the module version number.
- Types of supported data stores - The module may support one or more types of persistent data stores as separate subservices. For each type of data store, the DL provides information on the supported query operators and optionally provides specific information on the accessible data stores.

The DL may choose to provide information about the data stores that it has access to. Applications can obtain information about these data stores by using the CSSM_GetModuleInfo function call. The information about the data store includes the following:

- Types of persistent security objects - The types of security objects that may be stored include certificates, CRLs, keys, policy objects, and generic data objects. A single data store can contain a single object type in one format, a single object type in multiple formats, or multiple object types.
- Attributes of persistent security objects - The stored security object may have attributes which must be included by the calling application on data insertion, and which are returned by the DL on data retrieval.
- Data store indexes - These indexes are high-performance query paths constructed as part of data store creation and maintained by the data store.
- Secure access mechanisms - A data store may restrict a user’s ability to perform certain actions on the data store or on the data store’s contents. This structure exposes the mechanism required to authenticate to the data store.
- Record integrity capabilities - Some data stores will insure the integrity of the data store’s contents. To insure the integrity of the data store’s contents, the data store is expected to sign and verify each record.
- Data store location - The persistent repository can be local or remote.

To build indexes or to satisfy an application's request for record retrieval, the data store may need to parse the stored security objects. If the application has invoked CSSM_DL_DbSetRecordParsingFunctions for a given security object type, those functions will be used to parse that security object as the need arises. If the application has not explicitly set record-parsing functions, the default service provider modules set by the data store creator will be used for parsing.

Secured access to the data store and to the data store's contents may be enforced by the DL, the data store, or both. The partitioning of authentication responsibility is exposed via the DL and data store authentication mechanisms.

Data stores may be added to a DL in one of three ways:

- Using DL_DbCreate - This creates and opens a new, empty data store with the specified schema.
- Using DL_DbImport with information and data - If the specified data store does not exist, a new data store is created with the specified schema and the exported data records.
- Using DL_DbImport with information only - In this case, the data store's native format is the same as that managed by the DL service. Importing its information makes it accessible via this DL service.

In all cases, it is the responsibility of the DL service to update the KeyWorks registry with information about the new data store. This can be accomplished by making use of the CSSM_GetModuleInfo and CSSM_SetModuleInfo functions.

2.1 Categories of Operations

The DL service provider interface (SPI) defines four categories of operations:

- DL operations
- Data store operations
- Data record operations
- Extensibility operations.

DL operations are used to control access to the DL library. They include:

- Authentication to the DL Module - A user may be required to present valid credentials to the DL prior to accessing any of the data stores embedded in the DL module. The DL module will be responsible for insuring that the access privileges of the user are not exceeded.

The data store functions operate on a data store as a single unit. These operations include:

- Opening and closing data stores - A DL service manages the mapping of logical data store names to the storage mechanisms it uses to provide persistence. The caller uses logical names to reference persistent data stores. The open operation prepares an existing data store for future access by the caller. The close operation terminates current access to the data store by the caller.
- Creating and deleting data stores - A DL creates a new, empty data store and opens it for future access by the caller. An existing data store may be deleted. Deletion discards all data contained in the data store.
- Importing and exporting data stores - Occasionally a data store must be moved from one system to another, or a DL service may need to provide access to an existing data store. The import and export operations may be used in conjunction to support the transfer of an entire data store. The export operation prepares a snapshot of a data store. (Export does not delete the data store it snapshots.)

The import operation accepts a snapshot (generated by the export operation) and includes it in a new or existing data store managed by a DL. Alternately, the import operation may be used independently to register an existing data store with a DL.

The data record operations operate on a single record of a data store. They include:

- Adding new data objects **-** A DL adds a persistent copy of data object to an open data store. This operation may or may not include the creation of index entries. The mechanisms used to store and retrieve persistent data objects are private to the implementation of a DL module.
- Deleting data objects **-** A DL removes single data object from the data store.
- Retrieving data objects **-** A DL provides a search mechanism for selectively retrieving a copy of persistent security objects. Selection is based on a selection criterion.

Data store extensibility operations include:

- Pass through for unique, module-specific operations **-** A passthrough function is included in the DLI to allow data store libraries to expose additional services beyond what is currently defined in the KeyWorks API. KeyWorks passes an operation identifier and input parameters from the application to the appropriate DL. Within the DL_PassThrough function in the DL, the input parameters are interpreted and the appropriate operation performed. The DL developer is responsible for making known to the application the identity and parameters of the supported passthrough operations.

2.2 Data Storage Library Operations

DL_Authenticate
This function authenticates a user's ability to use this DL for accessing the underlying data stores.

2.3 Data Storage Operations

DL_DbOpen
For authorized users, this opens a data store with the specified logical name in the requested access mode. Returns a handle to the data store.

DL_DbClose
Closes a previously opened data store.

DL_DbCreate
This function creates a new, empty data store with the specified logical name and the specified schema. A DL may implement this function by opening the data store if it already exists, or creating the data store if it does not exist. The DL may also create the data store schema as part of the implementation of this function. The data store should be opened after this operation. As a side effect, the DL updates the KeyWorks Registry to expose information about the new data store.

DL_DbDelete
For authorized users, this deletes all records from the specified data store and removes current state information associated with that data store.

DL_DbImport
Accepts as input a flag for what to import, a filename, a logical name, and a schema for a data store. If information about the data store is being imported, then the DL updates its list of accessible data stores to include this new data store with the specified schema.
If the contents of the data store are being imported, then the file contains an exported copy of an existing data store. The data records contained in the file must be in the native format of a data

store. The DL imports all security objects in the file (such as certificates and CRLs), creating a new data record for each. If the specified logical name is that of an existing data store, the new records will be added to the data store. Otherwise, a new data store will be created with the specified schema to hold the new records.
Note: This mechanism can be used to copy data stores among systems or to restore a persistent data store from a backup copy. It could also be used to import data stores that were created and managed by other DLs, but this is not the typical implementation and use of this interface.

DL_DbExport

Accepts as input the logical name of a data store and the name of a target output file. The specified data store contains persistent data records. A representation of the schema for the data store being exported is written to the file, along with a copy of each data record in the data store.
Note: This mechanism can be used to copy data stores among systems or to create a backup of persistent data stores.

DL_DbSetRecordParsingFunctions

Sets the functions to be used for parsing the specified type of security object.

DL_DbGetRecordParsingFunctions

Returns the function pointers in use for parsing the specified type of security object.

DL_GetDbNameFromHandle

Retrieves the data source name corresponding to an opened database handle.

2.4 Data Record Operations

DL_DataInsert

Accepts as input a handle to a data store, the type of the security object, the attributes of the object and the object itself. The data object and its attributes are made persistent in the specified data store. This may or may not include the creation of index entries, etc. The DL module will return to the calling application a unique identifier for the input record, which may be used to rapidly retrieve the security object. The mechanisms used to store and retrieve persistent security objects are private to the implementation of the DL.

DL_DataDelete

Accepts as input a handle to a data store and a unique identifier of the security object. The object is removed from the data store. If the object is not found in the specified data store, or if the user does not have deletion permissions, the operation fails.

DL_DataGetFirst

Accepts as input a handle to a data store and a query. The query is composed of the type of data record to be retrieved, a selection predicate, and any limits on the query. Selection predicates are represented as a set of (relational operator, attribute) pairs that are connected by a conjunctive operator. Query limits provide a mechanism for the user to specify upper bounds on the search time and/or the number of records retrieved. Not all DL modules will support query limits. The specified data store is searched for data objects of the specified type that match the selection criteria. This function returns the first data object matching the criteria together with its attributes and a unique identifier for use in future references. If additional objects are matched, a selection handle is returned that may be used to retrieve the subsequent objects. A DL may limit the number of concurrently managed selection handles to exactly one. The library developer must document all such restrictions and application developers should proceed accordingly.

DL_DataGetNext

Accepts as input a selection results handle that was returned by an invocation of the function CSSM_DL_DataGetFirst. In response, a DL module returns the next data record, its attributes, and its unique identifier from the set specified by the selection results handle. If this is the last data record, the EndOfDataStore flag is set to CSSM_TRUE. A DL may limit the number of concurrently managed selection result handles to exactly one. The library developer must document such restrictions and application developers should proceed accordingly.

DL_DataAbortQuery
Cancels the query initiated by CSSM_DL_DataGetFirst function and resets the selection results handle.

DL_FreeUniqueRecord
Frees the memory associated with the input unique record structure.

2.5 Extensibility Functions

DL_PassThrough
Accepts as input an operation ID and a set of arbitrary input parameters. The operation ID may specify any type of operation a DL wishes to export for use by an application or by another module. Such operations may include queries or services that are specific to certain types of security objects or specific types of data stores managed by a DL module. It is the responsibility of the DL developer to make information on the availability and usage of passthrough operations available to application developers.

2.6 Data Structures

This section describes the data structures that may be passed to or returned from a DL function. Applications use these data structures to prepare and then pass input parameters into KeyWorks API function calls, which are passed without modification to the appropriate DL. The DL is responsible for interpreting them and returning the appropriate data structure to the calling application via KeyWorks. These data structures are defined in the header file, cssmtype.h, which is distributed with KeyWorks.

2.6.1 CSSM_BOOL

```
typedef uint32 CSSM_BOOL;

#define CSSM_TRUE   1
#define CSSM_FALSE  0
```

2.6.2 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[0];
} CSSM_DATA, *CSSM_DATA_PTR
```

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

Data **-** A byte array of size 0. Acts as a placeholder for a contiguous block of memory.

2.6.3 CSSM_DB_ACCESS_TYPE

This structure indicates a user's desired level of access to a data store.

```
typedef struct cssm_db_access_type {
    CSSM_BOOL ReadAccess;
    CSSM_BOOL WriteAccess;
    CSSM_BOOL PrivilegedMode; /* versus user mode */
    CSSM_BOOL Asynchronous;   /* versus synchronous */
} CSSM_DB_ACCESS_TYPE, *CSSM_DB_ACCESS_TYPE_PTR;
```

Definitions:

ReadAccess - A Boolean indicating that the user requests read access.

WriteAccess - A Boolean indicating that the user requests write access.

PrivilegedMode - A Boolean indicating that the user requests privileged operations.

Asynchronous - A Boolean indicating that the user requests asynchronous access.

2.6.4 CSSM_DB_ATTRIBUTE_DATA

This data structure holds an attribute value that can be stored in an attribute field of a persistent record. The structure contains a value for the data item and a reference to the meta-information (typing information and schema information) associated with the attribute.

```
typedef struct cssm_db_attribute_data {
    CSSM_DB_ATTRIBUTE_INFO Info;
    CSSM_DATA Value;
} CSSM_DB_ATTRIBUTE_DATA, *CSSM_DB_ATTRIBUTE_DATA_PTR;
```

Definitions:

Info - A reference to the meta-information (schema) describing this attribute in relationship to the data store at large.

Value - The data-present value assigned to the attribute.

2.6.5 CSSM_DB_ATTRIBUTE_INFO

This data structure describes an attribute of a persistent record. The description is part of the schema information describing the structure of records in a data store. The description includes the format of the attribute name and the attribute name itself. The attribute name implies the underlying data type of a value that may be assigned to that attribute.

```
typedef struct cssm_db_attribute_info {
    CSSM_DB_ATTRIBUTE_NAME_FORMAT AttributeNameFormat;
    union {
        char * AttributeName;            /* eg. "record label" */
        CSSM_OID AttributeID;            /* eg. CSSMOID_RECORDLABEL */
        uint32 AttributeNumber;
    };
} CSSM_DB_ATTRIBUTE_INFO, *CSSM_DB_ATTRIBUTE_INFO_PTR;
```

Definitions:

AttributeNameFormat - Indicates which of the three formats was selected to represent the attribute name.

AttributeName - A character string representation of the attribute name.

AttributeID - A DER-encoded Object Identifier (OID) representation of the attribute name.

AttributeNumber - An unsigned integer representation of the attribute name.

2.6.6 CSSM_DB_ATTRIBUTE_NAME_FORMAT

This enumerated list defines three formats used to represent an attribute name. The name can be represented by a character string in the native string encoding of the platform, by a number, or the name can be represented by an opaque OID structure that is interpreted by the DL module.

```
typedef enum cssm_db_attribute_name_format {
    CSSM_DB_ATTRIBUTE_NAME_AS_STRING = 0,
    CSSM_DB_ATTRIBUTE_NAME_AS_OID = 1,
    CSSM_DB_ATTRIBUTE_NAME_AS_NUMBER = 2
} CSSM_DB_ATTRIBUTE_NAME_FORMAT, *CSSM_DB_ATTRIBUTE_NAME_FORMAT_PTR;
```

2.6.7 CSSM_DB_CERTRECORD_SEMANTICS

These bit-masks define a list of usage semantics for how certificates may be used. It is anticipated that additional sets of bit-masks will be defined listing the usage semantics of how other record types can be used, such as CRL record semantics, key record semantics, policy record semantics, etc.

```
#define CSSM_DB_CERT_USE_ROOT    0x00000001 /* a self-signed root cert */
#define CSSM_DB_CERT_USE_TRUSTED 0x00000002 /* re-issued locally */
#define CSSM_DB_CERT_USE_SYSTEM  0x00000004 /* contains CSSM system cert */
#define CSSM_DB_CERT_USE_OWNER   0x00000008 /* private key is owned by the
                                               system user */
#define CSSM_DB_CERT_USE_REVOKED 0x00000010 /* revoked cert - used w\ CRL APIs */
#define CSSM_DB_CERT_SIGNING     0x00000011 /* use cert for signing only */
#define CSSM_DB_CERT_PRIVACY     0x00000012 /* use cert for encryption only */
```

2.6.8 CSSM_DB_CONJUNCTIVE

These are the conjunctive operations that can be used when specifying a selection criterion.

```
typedef enum cssm_db_conjunctive{
    CSSM_DB_NONE = 0,
    CSSM_DB_AND = 1,
    CSSM_DB_OR = 2
} CSSM_DB_CONJUNCTIVE, *CSSM_DB_CONJUNCTIVE_PTR;
```

2.6.9 CSSM_DB_HANDLE

```
typedef uint32 CSSM_DB_HANDLE    /* data store Handle */
```

2.6.10 CSSM_DB_INDEX_INFO

This structure contains the meta-information or schema description of an index defined on an attribute. The description includes the type of index (e.g., unique key or nonunique key), the logical location of the indexed attribute in the KeyWorks record (e.g., an attribute, a field within the opaque object in the record, or unknown), and the meta-information on the attribute itself.

```
typedef struct cssm_db_index_info {
    CSSM_DB_INDEX_TYPE IndexType;
    CSSM_DB_INDEXED_DATA_LOCATION IndexedDataLocation;
    CSSM_DB_ATTRIBUTE_INFO Info;
} CSSM_DB_INDEX_INFO, *CSSM_DB_INDEX_INFO_PTR;
```

Definitions:

IndexType **-** A CSSM_DB_INDEX_TYPE.

IndexedDataLocation **-** A CSSM_DB_INDEXED_DATA_LOCATION.

Info **-** The meta-information description of the attribute being indexed.

2.6.11 CSSM_DB_INDEX_TYPE

This enumerated list defines two types of indexes: indexes with unique values (i.e., primary database keys) and indexes with non-unique values. These values are used when creating a new data store and defining the schema for that data store.

```
typedef enum cssm_db_index_type {
    CSSM_DB_INDEX_UNIQUE = 0,
    CSSM_DB_INDEX_NONUNIQUE = 1
} CSSM_DB_INDEX_TYPE;
```

2.6.12 CSSM_DBINFO

This structure contains the meta-information about an entire data store. The description includes the types of records stored in the data store, the attribute schema for each record type, the index schema for all indexes over records in the data store, the type of authentication mechanism used to gain access to the data store, and other miscellaneous information used by the DL module to manage the data store in a secure manner.

```
typedef struct cssm_dbInfo {
    uint32 NumberOfRecordTypes;
    CSSM_DB_PARSING_MODULE_INFO_PTR DefaultParsingModules;
    CSSM_DB_RECORD_ATTRIBUTE_INFO_PTR RecordAttributeNames;
    CSSM_DB_RECORD_INDEX_INFO_PTR RecordIndexes;

    /* access restrictions for opening this data store */
    CSSM_USER_AUTHENTICATION_MECHANISM AuthenticationMechanism;

    /* transparent integrity checking options for this data store */
    CSSM_BOOL RecordSigningImplemented;
    CSSM_DATA SigningCertificate;
    CSSM_GUID SigningCsp;

    /* additional information */
    CSSM_BOOL IsLocal;
    char *AccessPath; /* URL, dir path, etc */
    void *Reserved;
} CSSM_DBINFO, *CSSM_DBINFO_PTR;
```

Definitions:

NumberOfRecordTypes - The number of distinct record types stored in this data store.

DefaultParsingModules - A pointer to a list of pairs (record-type, GUID) which define the default-parsing module for each record type.

RecordAttributeNames - The meta-information (schema) about the attributes associated with each record type that can be stored in this data store.

RecordIndexes - The meta- information (schema) about the indexes that are defined over each of the record types that can be stored in this data store.

AuthenticationMechanism - Defines the authentication mechanism required when accessing this data store.

RecordSigningImplemented - A flag indicating whether or not the DL module provides record integrity service based on digital signaturing of the data store records.

SigningCertificate - The certificate used to sign data store records when the transparent record integrity option is in effect.

SigningCsp - The GUID for the Cryptographic Service Provider (CSP) to be used to sign data store records when the transparent record integrity option is in effect.

IsLocal - Indicates whether the physical data store is local.

AccessPath - A character string describing the access path to the data store, such as a Universal Resource Locator (URL), a file system path name, a remote directory service name, etc.

Reserved - Reserved for future use.

2.6.13 CSSM_DB_OPERATOR

These are the logical operators that can be used when specifying a selection predicate.

```
typedef enum cssm_db_operator {
    CSSM_DB_EQUAL = 0,
    CSSM_DB_NOT_EQUAL = 1,
    CSSM_DB_APPROX_EQUAL = 2,
    CSSM_DB_LESS_THAN = 3,
    CSSM_DB_GREATER_THAN = 4,
    CSSM_DB_EQUALS_INITIAL_SUBSTRING = 5,
    CSSM_DB_EQUALS_ANY_SUBSTRING = 6,
    CSSM_DB_EQUALS_FINAL_SUBSTRING = 7,
    CSSM_DB_EXISTS = 8
} CSSM_DB_OPERATOR, *CSSM_DB_OPERATOR_PTR;
```

2.6.14 CSSM_DB_PARSING_MODULE_INFO

This structure aggregates the GUID of a default-parsing module with the record type that it parses. A parsing module can parse multiple record types. The same GUID would be repeated with each record type parsed by the module.

```
typedef struct cssm_db_parsing_module_info {
    CSSM_DB_RECORDTYPE RecordType;
    CSSM_GUID Module;
} CSSM_DB_PARSING_MODULE_INFO, *CSSM_DB_PARSING_MODULE_INFO_PTR;
```

Definitions:

RecordType - The type of record parsed by the module specified by GUID.

Module - A GUID identifying the default parsing module for the specified record type.

2.6.15 CSSM_DB_RECORD_ATTRIBUTE_DATA

This structure aggregates the actual data values for all of the attributes in a single record.

```
typedef struct cssm_db_record_attribute_data {
    CSSM_DB_RECORDTYPE DataRecordType;
    uint32 SemanticInformation;
    uint32 NumberOfAttributes;
    CSSM_DB_ATTRIBUTE_DATA_PTR AttributeData;
} CSSM_DB_RECORD_ATTRIBUTE_DATA, *CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR;
```

Definitions:

DataRecordType **-** A CSSM_DB_RECORDTYPE.

SemanticInformation **-** A bit-mask of type CSSM_XXXRECORD_SEMANTICS defining how the record can be used. Currently, these bit-masks are defined only for certificate records (CSSM_CERTRECORD_SEMANTICS). For all other record types, a bit-mask of zero must be used or a set of semantically meaningful masks must be defined.

NumberOfAttributes **-** The number of attributes in the record of the specified type.

AttributeData **-** A list of attribute name/value pairs.

2.6.16 CSSM_DB_RECORD_ATTRIBUTE_INFO

This structure contains the meta-information or schema information about all of the attributes in a particular record type. The description specifies the record type, the number of attributes in the record type, and a type information for each attribute.

```
typedef struct cssm_db_record_attribute_info {
    CSSM_DB_RECORDTYPE DataRecordType;
    uint32 NumberOfAttributes;
    CSSM_DB_ATTRIBUTE_INFO_PTR AttributeInfo;
} CSSM_DB_RECORD_ATTRIBUTE_INFO, *CSSM_DB_RECORD_ATTRIBUTE_INFO_PTR;
```

Definitions:

DataRecordType **-** A CSSM_DB_RECORDTYPE.

NumberOfAttributes **-** The number of attributes in a record of the specified type.

AttributeInfo **-** A list of pointers to the type information (schema) for each of the attributes.

2.6.17 CSSM_DB_RECORD_INDEX_INFO

This structure contains the meta-information or schema description of the set of indexes defined on a single record type. The description includes the type of the record, the number of indexes and the meta-information describing each index.

```
typedef struct cssm_db_record_index_info {
    CSSM_DB_RECORDTYPE DataRecordType;
    uint32 NumberOfIndexes;
    CSSM_DB_INDEX_INFO_PTR IndexInfo;
} CSSM_DB_RECORD_INDEX_INFO, *CSSM_DB_RECORD_INDEX_INFO_PTR;
```

Definitions:

DataRecordType - A CSSM_DB_RECORDTYPE.

NumberOfIndexes - The number of indexes defined on the records of the given type.

IndexInfo - An array of pointer to the meta-description of each index defined over the specified record type.

2.6.18 CSSM_DB_RECORD_PARSING_FNTABLE

This structure defines the three prototypes for functions that can parse the opaque data object stored in a record. It is used in the CSSM_DbSetRecordParsingFunctions function to override the default-parsing module for a given record type. The DL module developer designates the default-parsing module for each record type stored in the data store.

```
typedef struct cssm_db_record_parsing_fntable {
    CSSM_DATA_PTR (CSSMAPI *RecordGetFirstFieldValue)
        (CSSM_HANDLE Handle,
        CSSM_DB_RECORDTYPE RecordType,
        const CSSM_DATA_PTR Data,
        const CSSM_OID_PTR DataField,
        CSSM_HANDLE_PTR ResultsHandle,
        uint32 *NumberOfMatchedFields);
    CSSM_DATA_PTR (CSSMAPI *RecordGetNextFieldValue)
        (CSSM_HANDLE Handle,
        CSSM_HANDLE ResultsHandle);
    CSSM_RETURN (CSSMAPI *RecordAbortQuery)
        (CSSM_HANDLE Handle,
        CSSM_HANDLE ResultsHandle);
} CSSM_DB_RECORD_PARSING_FNTABLE, *CSSM_DB_RECORD_PARSING_FNTABLE_PTR;
```

Definitions:

**RecordGetFirstFieldValue* - A function to retrieve the value of a field in the opaque object. The field is specified by attribute name. The results handle holds the state information required to retrieve subsequent values having the same attribute name.

**RecordGetNextFieldValue* - A function to retrieve subsequent values having the same attribute name from a record parsed by the first function in this table.

**RecordAbortQuery* - Stop subsequent retrieval of values having the same attribute name from within the opaque object.

2.6.19 CSSM_DB_RECORDTYPE

This enumerated list defines the categories of persistent security-related objects that can be managed by a DL module. These categories are in one-to-one correspondence with types of records that can be managed by a DL module.

```
typedef enum cssm_db_recordtype {
    CSSM_DL_DB_RECORD_GENERIC = 0,
    CSSM_DL_DB_RECORD_CERT = 1,
    CSSM_DL_DB_RECORD_CRL = 2,
    CSSM_DL_DB_RECORD_PUBLIC_KEY = 3,
    CSSM_DL_DB_RECORD_PRIVATE_KEY = 4,
    CSSM_DL_DB_RECORD_SYMMETRIC_KEY = 5,
    CSSM_DL_DB_RECORD_POLICY = 6
} CSSM_DB_RECORDTYPE;
```

2.6.20 CSSM_DB_UNIQUE_RECORD

This structure contains an index descriptor and a module-defined value. The index descriptor may be used by the module to enhance the performance when locating the record. The module-defined value must uniquely identify the record. For a DBMS, this may be the record data. For a Public-Key Cryptographic Standard (PKCS#11) DL, this may be an object handle. Alternately, the DL may have a module-specific scheme for identifying data that has been inserted or retrieved.

```
typedef struct cssm_db_unique_record {
    CSSM_DB_INDEX_INFO RecordLocator;
    CSSM_DATA RecordIdentifier;
} CSSM_DB_UNIQUE_RECORD, *CSSM_DB_UNIQUE_RECORD_PTR;
```

Definitions:

RecordLocator -The information describing how to locate the record efficiently.

RecordIdentifier - A module-specific identifier which will allow the DL to locate this record.

2.6.21 CSSM_DL_DB_HANDLE

This data structure holds a pair of handles, one for a DL and another for a data store opened and being managed by the DL.

```
typedef struct cssm_dl_db_handle {
    CSSM_DL_HANDLE DLHandle;
    CSSM_DB_HANDLE DBHandle;
} CSSM_DL_DB_HANDLE, *CSSM_DL_DB_HANDLE_PTR;
```

Definitions:

DLHandle - Handle of an attached module that provides DL services.

2.6.22 CSSM_DL_DB_LIST

This data structure defines a list of handle pairs (DL handle, data store handle).

```
typedef struct cssm_dl_db_list {
    uint32 NumHandles;
    CSSM_DL_DB_HANDLE_PTR DLDBHandle;
} CSSM_DL_DB_LIST, *CSSM_DL_DB_LIST_PTR;
```

Definitions:

NumHandles - Number of (DL handle, data store handle) pairs in the list.

DLDBHandle - List of (DL handle, data store handle) pairs.

2.6.23 CSSM_DL_CUSTOM_ATTRIBUTES

This structure can be used by DL module developers to define a set of attributes for a custom data store format.

```
typedef void *CSSM_DL_CUSTOM_ATTRIBUTES;
```

2.6.24 CSSM_DL_FFS_ATTRIBUTES

This structure can be used by DL module developers to define a set of attributes for a flat file system data store format.

```
typedef void *CSSM_DL_FFS_ATTRIBUTES;
```

2.6.25 CSSM_DL_HANDLE

A unique identifier for an attached module that provides DL services.

```
typedef uint32 CSSM_DL_HANDLE/* Data Storage Library Handle */
```

2.6.26 CSSM_DL_LDAP_ATTRIBUTES

This structure can be used by DL module developers to define a set of attributes for an Lightweight Directory Access Protocol (LDAP) data store format.

```
typedef void *CSSM_DL_LDAP_ATTRIBUTES;
```

2.6.27 CSSM_DL_ODBC_ATTRIBUTES

This structure can be used by DL module developers to define a set of attributes for an Open Database Connectivity (ODBC) data store format.

typedef void *CSSM_DL_ODBC_ATTRIBUTES;

2.6.28 CSSM_DL_PKCS11_ATTRIBUTES

Each type of DL module can define its own set of type-specific attributes. This structure contains the attributes that are specific to a PKCS#11-compliant data storage device.

```
typedef struct cssm_dl_pkcs11_attributes {
    uint32 DeviceAccessFlags;
} *CSSM_DL_PKCS11_ATTRIBUTES;
```

Definitions:

DeviceAccessFlags - Specifies the PKCS#11-specific access modes applicable for accessing persistent objects in a PKCS#11 data store.

2.6.29 CSSM_DLSUBSERVICE

Three structures are used to contain all of the static information that describes a DL module: cssm_moduleinfo, cssm_serviceinfo, and cssm_dlsubservice. This descriptive information is securely stored in the KeyWorks registry when the DL module is installed with KeyWorks. A DL module may implement multiple types of services and organize them as subservices. For example, a DL module supporting two types of remote directory services may organize its implementation into two subservices: one for an X.509 certificate directory and a second for custom enterprise policy data store. Most DL modules will implement exactly one subservice.

Not all DL modules can maintain a summary of managed data stores. In this case, the DL module reports its number of data stores as CSSM_DB_DATASTORES_UNKNOWN. Data stores can (and probably do) exist, but the DL module cannot provide a list of them.

```
#define       CSSM_DB_DATASTORES_UNKNOWN (-1)
```

The descriptive information stored in these structures can be queried using the function CSSM_GetModuleInfo and specifying the DL module GUID.

```
typedef struct cssm_dlsubservice {
    uint32 SubServiceId;
    CSSM_STRING Description;
    CSSM_DLTYPE Type;
    union {
        CSSM_DL_CUSTOM_ATTRIBUTES CustomAttributes;
        CSSM_DL_LDAP_ATTRIBUTES LdapAttributes;
        CSSM_DL_ODBC_ATTRIBUTES OdbcAttributes;
        CSSM_DL_PKCS11_ATTRIBUTES Pkcs11Attributes;
        CSSM_DL_FFS_ATTRIBUTES FfsAttributes;
    } Attributes;

    CSSM_DL_WRAPPEDPRODUCT_INFO WrappedProduct;
    CSSM_USER_AUTHENTICATION_MECHANISM AuthenticationMechanism;
    /* meta-information about the query support provided by the module */
    uint32 NumberOfRelOperatorTypes;
    CSSM_DB_OPERATOR_PTR RelOperatorTypes;
    uint32 NumberOfConjOperatorTypes;
    CSSM_DB_CONJUNCTIVE_PTR ConjOperatorTypes;
    CSSM_BOOL QueryLimitsSupported;

    /* meta-information about the encapsulated data stores (if known) */
    uint32 NumberOfDataStores;
    CSSM_NAME_LIST_PTR DataStoreNames;
    CSSM_DBINFO_PTR DataStoreInfo;

    /* additional information */
    void *Reserved;
} CSSM_DLSUBSERVICE, *CSSM_DLSUBSERVICE_PTR;
```

Definitions:

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

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

Type **-** An identifier for the type of underlying data store the DL module uses to provide persistent storage.

Attributes - A structure containing attributes that define additional parameter values specific to the DL module type.

WrappedProduct - Pointer to a CSSM_DL_WRAPPEDPRODUCT_INFO structure describing a product that is wrapped by the DL module.

AuthenticationMechanism - Defines the authentication mechanism required when using this DL module. This authentication mechanism is distinct from the authentication mechanism (specified in a cssm_dbInfo structure) required to access a specific data store.

NumberOfRelOperatorTypes - The number of distinct relational operators the DL module accepts in selection queries for retrieving records from its managed data stores.

RelOperatorTypes - The list of specific relational operators that can be used to formulate selection predicates for queries on a data store. The list contains NumberOfRelOperatorTypes operators.

NumberOfConjOperatorTypes - The number of distinct conjunctive operators the DL module accepts in selection queries for retrieving records from its managed data stores.

ConjOperatorTypes - A list of specific conjunctive operators that can be used to formulate selection predicates for queries on a data store. The list contains NumberOfConjOperatorTypes operators.

QueryLimitsSupported - A Boolean indicating whether query limits are effective when the DL module executes a query.

NumberOfDataStores - The number of data stores managed by the DL module. This information may not be known by the DL module, in which case this value will equal CSSM_DB_DATASTORES_UNKNOWN.

DataStoreNames - A list of names of the data stores managed by the DL module. This information may not be known by the DL module and hence may not be available. The list contains NumberOfDataStores entries.

DataStoreInfo - A list of pointers to the meta-information (schema) for each data store managed by the DL module. This information may not be known in advance by the DL module and hence may not be available through this structure. The list contains NumberOfDataStores entries.

Reserved - Reserved for future use.

2.6.30 CSSM_DLTYPE

This enumerated list defines the types of underlying DBMSs that can be used by the DL module to provide services. It is the option of the DL module to disclose this information.

```
typedef enum cssm_dltype {
    CSSM_DL_UNKNOWN = 0,
    CSSM_DL_CUSTOM = 1,
    CSSM_DL_LDAP = 2,
    CSSM_DL_ODBC = 3,
    CSSM_DL_PKCS11 = 4,
    CSSM_DL_FFS = 5,/* flat file systemor fast file system */
    CSSM_DL_MEMORY = 6,
    CSSM_DL_REMOTEDIR = 7
} CSSM_DLTYPE, *CSSM_DLTYPE_PTR;
```

2.6.31 CSSM_DL_WRAPPEDPRODUCTINFO

This structure lists the set of data store services used by the DL module to implement its services. The DL module vendor is not required to provide this information, but may choose to do so. For example, a DL module that uses a commercial DBMS can record information about that product in this structure. Another example is a DL module that supports certificate storage through an X.500 certificate directory server. The DL module can describe the X.500 directory service in this structure.

```
typedef struct cssm_dl_wrappedproductinfo {
    CSSM_VERSION StandardVersion;
    CSSM_STRING StandardDescription;
    CSSM_VERSION ProductVersion;
    CSSM_STRING ProductDescription;
    CSSM_STRING ProductVendor;
    uint32 ProductFlags;
} CSSM_DL_WRAPPEDPRODUCT_INFO, *CSSM_DL_WRAPPEDPRODUCT_INFO_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 DL module.

ProductDescription - A string describing the product.

ProductVendor - The name of the product vendor.

ProductFlags - A bit-mask enumerating selectable features of the database service that the DL module uses in its implementation.

2.6.32 CSSM_NAME_LIST

```
typedef struct cssm_name_list {
    uint32 NumStrings;
    char** String;
} CSSM_NAME_LIST, *CSSM_NAME_LIST_PTR;
```

2.6.33 CSSM_QUERY

This structure holds a complete specification of a query to select records from a data store.

```
typedef struct cssm_query {
    CSSM_DB_RECORDTYPE RecordType;
    CSSM_DB_CONJUNCTIVE Conjunctive;
    uint32 NumSelectionPredicates;
    CSSM_SELECTION_PREDICATE_PTR SelectionPredicate;
    CSSM_QUERY_LIMITS QueryLimits;
    CSSM_QUERY_FLAGS QueryFlags;
} CSSM_QUERY, *CSSM_QUERY_PTR;
```

Definitions:

RecordType **-** Specifies the type of record to be retrieved from the data store.

Conjunctive **-** The conjunctive operator to be used in constructing the selection predicate for the query.

NumSelectionPredicates **-** The number of selection predicates to be connected by the specified conjunctive operator to form the query.

SelectionPredicate **-** The list of selection predicates to be combined by the conjunctive operator to form the data store query.

QueryLimits **-** Defines the time and space limits for processing the selection query. The constant values CSSM_QUERY_TIMELIMIT_NONE and CSM_QUERY_SIZELIMIT_NONE should be used to specify no limit on the resources used in processing the query.

QueryFlags **-** An integer that indicates the return format of the key data. This integer is represented by CSSM_QUERY_RETURN_DATA. When CSSM_QUERY_RETURN_DATA is 1, the key record is returned in KeyWorks format. When CSSM_QUERY_RETURN_DATA is 0, the information is returned in raw format (a format native to the individual module, BSAFE, or PKCS11).

2.6.34 CSSM_QUERY_LIMITS

This structure defines the time and space limits a caller can set to control early termination of the execution of a data store query. The constant values CSSM_QUERY_TIMELIMIT_NONE and CSM_QUERY_SIZELIMIT_NONE should be used to specify no limit on the resources used in processing the query. These limits are advisory. Not all DL modules recognize and act upon the query limits set by a caller.

```
#define       CSSM_QUERY_TIMELIMIT_NONE  0
#define       CSSM_QUERY_SIZELIMIT_NONE  0

typedef struct cssm_query_limits {
    uint32 TimeLimit;
    uint32 SizeLimit;
} CSSM_QUERY_LIMITS, *CSSM_QUERY_LIMITS_PTR;
```

Definitions:

TimeLimit **-** Defines the maximum number of seconds of resource time that should be expended performing a query operation. The constant value CSSM_QUERY_TIMELIMIT_NONE means no time limit is specified.

SizeLimit **-** Defines the maximum number of records that should be retrieved in response to a single query. The constant value CSSM_QUERY_SIZELIMIT_NONE means no space limit is specified.

2.6.35 CSSM_SELECTION_PREDICATE

This structure defines the selection predicate to be used for database queries.

```
typedef struct cssm_selection_predicate {
    CSSM_DB_OPERATOR DbOperator;
    CSSM_DB_ATTRIBUTE_DATA Attribute;
} CSSM_SELECTION_PREDICATE, *CSSM_SELECTION_PREDICATE_PTR;
```

Definitions:

DbOperator - The relational operator to be used when comparing a value to the values stored in the specified attribute in the data store.

Attribute - The meta-information about the attribute to be searched and the attribute value to be used for comparison with values in the data store.

2.7 Data Storage Operations

This section describes the function prototypes and error codes defined for the data source operations in the DLI. The functions are exposed to KeyWorks through a function table, so the function names may vary at the discretion of the DL developer. However, the function parameter list and return type must match the prototypes given in this section in order to be used by applications.

2.7.1 DL_Authenticate

```
CSSM_RETURN  DL_Authenticate  (const CSSM_DL_DB_HANDLE DLDBHandle,
                               const CSSM_DB_ACCESS_TYPE_PTR AccessRequest,
                               const CSSM_USER_AUTHENTICATION_PTR
                               UserAuthentication)
```

This function allows the caller to provide authentication credentials to the DL module at a time other than data store creation, deletion, open, import, and export. *AccessRequest* defines the type of access to be associated with the caller. If the authentication credential applies to access and use of a DL module in general, then the data store handle specified in the *DLDBHandle* must be NULL. When the authorization credential is to applied to a specific data store, the handle for that data store must be specified in the *DLDBHandle* pair.

Parameters

DLDBHandle (input)
The handle pair that describes the DL module used to perform this function and the data store to which access is being requested. If the form of authentication being requested is authentication to the DL module in general, then the data store handle must be NULL.

AccessRequest (input)
An indicator of the requested access mode for the data store or DL module in general.

UserAuthentication (input)
The caller's credential as required for obtaining authorized access to the data store or to the DL module in general.

Return Value

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

2.7.2 DL_DbClose

CSSM_RETURN DL_DbClose (CSSM_DL_DB_HANDLE DLDBHandle)

This function closes an open data store.

Parameters

DLDBHandle (input)
A handle structure containing the DL handle for the attached DL module and the database (DB) handle for an open data store managed by the DL. This specifies the open data store to be closed.

Return Value

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

DL_DbOpen

2.7.3 DL_DbCreate

```
CSSM_DB_HANDLE  DL_DbCreate
                    (CSSM_DL_HANDLE DLHandle,
                     const char *DbName,
                     const CSSM_DBINFO_PTR DBInfo,
                     const CSSM_DB_ACCESS_TYPE_PTR AccessRequest,
                     const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
                     const void *OpenParameters)
```

This function creates a new, empty data store with the specified logical name.

Parameters

DLHandle(input)
The handle that describes the DL module to be used to perform this function.

DBInfo (input)
A pointer to a structure describing the format/schema of each record type that will be stored in the new data store.

AccessRequest (input)
An indicator of the requested access mode for the data store, such as read-only or read/write.

UserAuthentication (input/optional)
The caller's credential as required for obtaining access to the data store. If no credentials are required for the specified data store, then user authentication must be NULL.

OpenParameters (input/optional)
A pointer to a module-specific set of parameters required to open the data store.

Return Value

Returns the CSSM_DB_HANDLE of the newly created data store. If the handle is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

DL_DbOpen, DL_DbClose, DL_DbDelete

2.7.4 DL_DbDelete

```
CSSM_RETURN  DL_DbDelete
        (CSSM_DL_HANDLE DLHandle,
         const char *DbName,
         const CSSM_USER_AUTHENTICATION_PTR UserAuthentication)
```

This function deletes all records from the specified data store and removes all state information associated with that data store.

Parameters

DLHandle(input)
The handle that describes the DL module to be used to perform this function.

DbName(input)
A pointer to the string containing the logical name of the data store.

UserAuthentication (input/optional)
The caller's credential as required for obtaining access (and consequently deletion capability) to the data store. If no credentials are required for the specified data store, then user authentication must be NULL

Return Value

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

DL_DbCreate, DL_DbOpen, DL_DbClose

2.7.5 DL_DbExport

```
CSSM_RETURN  DL_DbExport  (CSSM_DL_HANDLE DLHandle,
                           const char *DbDestinationName,
                           const char *DbSourceName,
                           const CSSM_BOOL InfoOnly,
                           const CSSM_USER_AUTHENTICATION_PTR
                           UserAuthentication)
```

This function exports a copy of the data store records from the source data store to a data container that can be used as the input data source for the DL_DbImport function. The DL module may require additional user authentication to determine authorization to snapshot a copy of an existing data store.

Parameters

DLHandle(input)
The handle that describes the DL module to be used to perform this function.

DbSourceName (input)
The name of the data store from which the records are to be exported.

DbDestinationName (*input)*
The name of the destination data container which will contain a copy of the source data store's records.

InfoOnly (input)
A Boolean value indicating what to export. If CSSM_TRUE, export only the *DBInfo* that describes the data store. If CSSM_FALSE, export both the *DBInfo* and all of the records in the specified data store.

UserAuthentication (input/optional)
The caller's credential as required for authorization to snapshot/copy a data store. If the DL module requires no additional credentials to perform this operation, then user authentication can be NULL.

Return Value

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

DL_DbImport

2.7.6 DL_GetDbNameFromHandle

char * DL_GetDbNameFromHandle (CSSM_DL_DB_HANDLE DLDBHandle)

This function retrieves the data source name corresponding to an opened database handle. A DL module is responsible for allocating the memory required for the list.

Parameters

DLDBHandle (input)
The handle pair that describes the DL module used to perform this function and the data store to which access is being requested.

Return Value

Returns a string that contains a data store name. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

2.7.7 DL_DbGetRecordParsingFunctions

```
CSSM_DB_RECORD_PARSING_FNTABLE_PTR  DL_DbGetRecordParsingFunctions
                                        (CSSM_DL_HANDLE DLHandle,
                                        const char* DbName,
                                        CSSM_DB_RECORDTYPE RecordType)
```

This function gets the records parsing function table, that operates on records of the specified type, in the specified data store. Three record-parsing functions can be returned in the table. The functions can be implemented to parse multiple record types. In this case, multiple calls to DL_DbGetRecordParsingFunctions must be made, once for each record type whose parsing functions are required by the caller. The DL module uses these functions to parse the opaque data object stored in a data store record. If no parsing function table has been set for a given record type, then a NULL value is returned.

Parameters

DLHandle (input)
The handle that describes the DL module to be used to perform this function.

DbName (input)
The name of the data store with which the parsing functions are associated.

RecordType (input)
The record type whose parsing functions are requested by the caller.

Return Value

A pointer to a function table for the parsing function appropriate to the specified record type. When CSSM_NULL is returned, either no function table has been set for the specified record type or an error has occurred. Use CSSM_GetError to obtain the error code and determine the reason for the NULL result.

See Also

DL_SetRecordParsingFunctions

2.7.8 DL_DbImport

```
CSSM_RETURN  DL_DbImport
                    (CSSM_DL_HANDLE DLHandle,
                     const char *DbDestinationName,
                     const char *DbSourceName,
                     const CSSM_DBINFO_PTR DBInfo,
                     const CSSM_BOOL InfoOnly,
                     const CSSM_USER_AUTHENTICATION_PTR UserAuthentication)
```

This function creates a new data store, or adds to an existing data store, by importing records from the specified data source. It is assumed that the data source contains records exported from a data store using the function DL_DbExport.

The *DbDestinationName* specifies the name of a new or existing data store. If a new data store is being created, the *DBInfo* structure provides the meta-information (schema) for the new data store. This structure describes the record attributes and the index schema for the new data store. If the data store already exists, then the existing meta-information (schema) is used. (Dynamic schema evolution is not supported.)

Typically, user authentication is required to create a new data store or to write to an existing data store. An authentication credential is presented to the DL module in the form required by the module. The required form is documented in the capabilities and feature descriptions for this module. The resulting data store is not opened as a result of this operation.

Parameters

DLHandle(input)
The handle that describes the DL module to be used to perform this function.

DbDestinationName (input)
The name of the destination data store in which to insert the records.

DbSourceName (*input)*
The name of the data source from which to obtain the records that are added to the data store.

DBInfo (*input/optional)*
A data structure containing a detailed description of the meta-information (schema) for the new data store. If a new data store is being created, then the caller must specify the meta-information (schema), or the data source must include the meta-information required for proper import of the records. If meta-information is supplied by the caller and specified in the data source, then the meta-information provided by the caller overrides the meta-information recorded in the data source. If the data store exists and records are being added, then this pointer must be NULL. The existing meta-information will be used and the schema cannot be evolved.

InfoOnly (input)
A Boolean value indicating what to import. If CSSM_TRUE, import only the *DBInfo* that describes the a data store. If CSSM_FALSE, import both the *DBInfo* and all of the records exported from a data store.

UserAuthentication (input/optional)
The caller's credential as required for authorization to create a data store. If the DL module requires no additional credentials to create a new data store, then user authentication can be NULL.

Return Value

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

DL_DbExport

2.7.9 DL_DbOpen

CSSM_DB_HANDLE DL_DbOpen (CSSM_DL_HANDLE DLHandle,
const char *DbName
const CSSM_DB_ACCESS_TYPE_PTR AccessRequest,
const CSSM_USER_AUTHENTICATION_PTR
UserAuthentication,
const void *OpenParameters)

This function opens the data store with the specified logical name under the specified access mode. If user authentication credentials are required, they must be provided. In addition, additional open parameters may be required to open a given data store and are supplied in the *OpenParameters*.

Parameters

DLHandle(input)
The handle that describes the DL module to be used to perform this function.

DbName(input)
A pointer to the string containing the logical name of the data store.

AccessRequest (input)
An indicator of the requested access mode for the data store, such as read-only or read/write.

UserAuthentication (input/optional)
The caller's credential as required for obtaining access to the data store. If no credentials are required for the specified data store, then user authentication must be NULL.

OpenParameters (input/optional)
A pointer to a module-specific set of parameters required to open the data store.

Return Value

Returns the CSSM_DB_HANDLE of the opened data store. If the handle is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

DL_DbClose

2.7.10 DL_DbSetRecordParsingFunctions

CSSM_RETURN DL_DbSetRecordParsingFunctions

```
(CSSM_DL_HANDLE DLHandle,
const char* DbName,
CSSM_DB_RECORDTYPE RecordType,
const
CSSM_DB_RECORD_PARSING_FNTABLE_PTR
FunctionTable)
```

This function sets the records parsing function table, overriding the default-parsing module for records of the specified type in the specified data store. Three record-parsing functions can be specified in the table. The functions can be implemented to parse multiple record types. In this case, multiple calls to DL_DbSetRecordParsingFunctions must be made, once for each record type that should be parsed using these functions. The DL module uses these functions to parse the opaque data object stored in a data store record. If no parsing function table has been set for a given record type, then the default-parsing module is invoked for that record type.

Parameters

DLHandle (input)
The handle that describes the DL module to be used to perform this function.

DbName (input)
The name of the data store with which to associate the parsing functions.

RecordType (input)
One of the record types parsed by the functions specified in the function table.

FunctionTable (input)
The function table referencing the three parsing functions to be used with the data store specified by *DbName*.

Return Value

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

DL_GetRecordParsingFunctions

2.8 Data Record Operations

2.8.1 DL_DataAbortQuery

CSSM_RETURN DL_DataAbortQuery (CSSM_DL_DB_HANDLE DLDBHandle,
CSSM_HANDLE ResultsHandle)

This function terminates the query initiated by CSSM_DL_DataGetFirst or CSSM_DL_DataGetNext, and allows a DL to release all intermediate state information associated with the query.

Parameters

DLDBHandle (input)
The handle pair that describes the DL module to be used to perform this function and the open data store from which records were selected by the initiating query.

ResultsHandle (input)
The selection handle returned from the initial query function.

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

DL_DataGetFirst, DL_DataGetNext

2.8.2 DL_DataDelete

```
CSSM_RETURN  DL_DataDelete
                              (CSSM_DL__DB_HANDLE DLDBHandle,
                               CSSM_DB_RECORDTYPE RecordType,
                               const CSSM_DB_UNIQUE_RECORD_PTR UniqueRecordIdentifier)
```

This function removes from the specified data store the data record specified by the unique record identifier.

Parameters

DLDBHandle (input)
The handle pair that describes the DL module to be used to perform this function and the open data store from which to delete the specified data record.

RecordType (input/optional)
An indicator of the type of record to be deleted from the data store. The *UniqueRecordIdentifier* may be unique only among records of the same type. If the data store contains only one record type or the unique identifiers managed are globally unique, then the record type need not be specified.

UniqueRecordIdentifier (input)
A pointer to a CSSM_DB_UNIQUE_RECORD identifier containing unique identification of the data record to be deleted from the data store. The identifier may be unique only among records of a given type. Once the associated record has been deleted, this unique record identifier cannot be used in future references.

Return Value

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

DL_DataInsert

2.8.3 DL_DataGetFirst

```
CSSM_DB_UNIQUE_RECORD_PTR  DL_DataGetFirst
                                    (CSSM_DL_DB_HANDLE DLDBHandle,
                                     const CSSM_QUERY_PTR Query,
                                     CSSM_HANDLE_PTR  ResultsHandle,
                                     CSSM_BOOL  *EndOfDataStore,
                                     CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
                                     CSSM_DATA_PTR  Data)
```

This function retrieves the first data record in the data store that matches the selection criteria. The selection criteria (including selection predicate and comparison values) is specified in the *Query* structure. The DL module can use internally managed indexing structures to enhance the performance of the retrieval operation. This function returns the first record, satisfying the query in the list of *Attributes* and the opaque *Data* object. This function also returns a flag indicating whether additional records also satisfied the query, and a results handle to be used when retrieving subsequent records satisfying the query. Finally, this function returns a unique record identifier associated with the retrieved record. This structure can be used in future references to the retrieved data record.

Parameters

DLDBHandle (input)
The handle pair that describes the DL module to be used to perform this function and the open data store to search for records satisfying the query.

Query (input/optional)
The query structure specifying the selection predicates used to query the data store. The structure contains meta-information about the search fields and the relational and conjunctive operators forming the selection predicate. The comparison values to be used in the search are specified in the Attributes and Data parameter. If no query is specified, the DL module can return the first record in the data store (i.e., perform sequential retrieval) or return an error.

ResultsHandle (output)
This handle should be used to retrieve subsequent records that satisfied this query.

EndOfDataStore (output)
A flag indicating whether a record satisfying this query was available to be retrieved in the current operation. If CSSM_TRUE, then a record was available and was retrieved unless an error condition occurred. If CSSM_FALSE, then all records satisfying the query have been previously retrieved and no record has been returned by this operation.

Attributes (output)
A list of attributes values (and corresponding meta-information) from the retrieved record.

Data (output)
The opaque object stored in the retrieved record.

Return Value

If successful and *EndOfDataStore* is CSSM_FALSE, this function returns a pointer to a CSSM_UNIQUE_RECORD structure containing a unique record locator and the record. If the pointer is NULL and *EndOfDataStore* is CSSM_TRUE, then a normal termination condition has occurred. If the pointer is NULL and *EndOfDataStore* is CSSM_FALSE, then an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

DL_DataGetNext, DL_DataAbortQuery.

2.8.4 DL_DataGetNext

```
CSSM_DB_UNIQUE_RECORD_PTR  DL_DataGetNext
                                        (CSSM_DL_DB_HANDLE DLDBHandle,
                                         CSSM_HANDLE ResultsHandle,
                                         CSSM_BOOL *EndOfDataStore,
                                         CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
                                         CSSM_DATA_PTR Data)
```

This function returns the next data record referenced by the *ResultsHandle*. The *ResultsHandle* parameter references a set of records selected by an invocation of the DL_DataGetFirst function. The record values are returned in the *Attributes* and *Data* parameters. A flag indicates whether additional records satisfying the original query remain to be retrieved. The function also returns a unique record identifier for the return record.

Parameters

DLDBHandle (input)
The handle pair that describes the DL module to be used to perform this function and the open data store from which records were selected by the initiating query.

ResultsHandle (output)
The handle identifying a set of records retrieved by a query executed by the DL_DataGetFirst function.

EndOfDataStore (output)
A flag indicating whether a record satisfying this query was available to be retrieved in the current operation. If CSSM_TRUE, then a record was available and was retrieved unless an error condition occurred. If CSSM_FALSE, then all records satisfying the query have been previously retrieved and no record has been returned by this operation.

Attributes (output)
A list of attributes values (and corresponding meta-information) from the retrieved record.

Data (output)
The opaque object stored in the retrieved record.

Return Value

If successful and EndOfDataStore is CSSM_FALSE, this function returns a pointer to a CSSM_UNIQUE_RECORD structure containing a unique record locator and the record. If the pointer is NULL and EndOfDataStore is CSSM_TRUE, then a normal termination condition has occurred. If the pointer is NULL and EndOfDataStore is CSSM_FALSE, then an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

DL_DataGetFirst, DL_DataAbortQuery

2.8.5 DL_DataInsert

```
CSSM_DB_UNIQUE_RECORD_PTR  DL_DataInsert
                              (CSSM_DL_DB_HANDLE DLDBHandle,
                               const CSSM_DB_RECORDTYPE RecordType,
                               const CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
                               const CSSM_DATA_PTR Data)
```

This function creates a new persistent data record of the specified type by inserting it into the specified data store. The values contained in the new data record are specified by the *Attributes* and the *Data* parameters. The attribute value list contains zero or more attribute values. The DL modules can assume default values for unspecified attribute values or can return an error condition when required attributes values are not specified by the caller. The *Data* parameter is an opaque object to be stored in the new data record.

Parameters

DLDBHandle (input)
The handle pair that describes the DL module to be used to perform this function and the open data store in which to insert the new data record.

RecordType (input)
Indicates the type of data record being added to the data store.

Attributes (input/optional)
A list of structures containing the attribute values to be stored in that attribute and the meta-information (schema) describing those attributes. The list contains, at most, one entry per attribute in the specified record type. The DL module can assume default values for those attributes that are not assigned values by the caller or may return an error. If the specified record type does not contain any attributes, this parameter must be NULL.

DataRecord (input/optional)
A pointer to the CSSM_DATA structure that contains the opaque data object to be stored in the new data record. If the specified record type does not contain an opaque data object, this parameter must be NULL.

Return Value

A pointer to a CSSM_DB_UNIQUE_RECORD_POINTER containing a unique identifier associated with the new record. This unique identifier structure can be used in future references to this record. When NULL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

DL_DataDelete

2.8.6 DL_FreeUniqueRecord

```
CSSM_RETURN  DL_FreeUniqueRecord
                    (CSSM_DL_DB_HANDLE DLDBHandle,
                     CSSM_DB_UNIQUE_RECORD_PTR UniqueRecord)
```

This function frees the memory associated with the data store unique record structure.

Parameters

DLDBHandle (input)
The handle pair that describes the DL module to be used to perform this function.

UniqueRecord (input)
The pointer to the memory that describes the data store unique record structure.

Return Value

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

DL_DataInsert, DL_DataGetFirst, DL_DataGetNext

2.9 Extensibility Functions

The DL_PassThrough function is provided to allow DL developers to extend the certificate and CRL format-specific storage functionality of the KeyWorks API. Because it is exposed to KeyWorks as only a function pointer, its name internal to the DL can be assigned at the discretion of the DL module developer. However, its parameter list and return value must match what is shown below. The error codes listed in this section are the generic codes all data storage libraries may use to describe common error conditions.

2.9.1 DL_PassThrough

```
void *  DL_PassThrough  (CSSM_DL_DB_HANDLE DLDBHandle,
                          uint32 PassThroughId,
                          const void *InputParams)
```

This function allows applications to call additional module-specific operations that have been exported by the DL. Such operations may include queries or services specific to the domain represented by the DL module.

Parameters

DLDBHandle (input)
The handle pair that describes the DL module to be used to perform this function and the open data store upon which the function is to be performed.

PassThroughId (input)
An identifier assigned by a DL module to indicate the exported function to be performed.

InputParams (input)
A pointer to a module, implementation-specific structure containing parameters to be interpreted in a function-specific manner by the requested DL 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. Data Storage Library Function Examples

3.1 Attach/Detach Example

The DL 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 DL 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 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”
CSSM_GUID dl_guid =
{ 0x5fc43dc1, 0x732, 0x11d0, { 0xbb, 0x14, 0x0, 0xaa, 0x0, 0x36, 0x67, 0x2d } };
CSSM_FUNCTIONTABLE FunctionTable;
CSSM_SPI_FUNC_TBL_PTR UpcallTable;

/* global variables used for registration */
CSSM_REGISTRATION_INFO     reg_info;
CSSM_SPI_DL_FUNCS          dl_jmp_tbl;
CSSM_SPI_MEMORY_FUNCS      upcall_tbl = {NULL, NULL, NULL, NULL};
CSSM_MODULE_FUNCS          module_funcs;

CSSM_RETURN CSSMAPI AddInAuthenticate(char* cssmCredentialPath, char*
cssmSection) {

       CSSM_RETURN   ret_code;

       /* first set up the DL jump table */
       memset(&dl_jmp_tbl, 0, sizeof(dl_jmp_tbl) );

       /* Fill in FunctionTable with function pointers */
       FunctionTable.Authenticate = DL_Authenticate;
       FunctionTable.DbOpen = DL_DbOpen;
       FunctionTable.DbClose = DL_DbClose;
       FunctionTable.DbCreate = DL_DbCreate;
       FunctionTable.DbDelete = DL_DbDelete;
       FunctionTable.DbImport = DL_DbImport;
       FunctionTable.DbExport = DL_DbExport;
       FunctionTable.DbSetRecordParsingFunctions =
              DL_DbSetRecordParsingFunctions;
       FunctionTable.DbGetRecordParsingFunctions =
               DL_DbGetRecordParsingFunctions;
       FunctionTable.GetDbNameFromHandle = DL_GetDbNameFromHandle;
       FunctionTable.DataInsert = DL_DataInsert;
       FunctionTable.DataDelete = DL_DataDelete;
       FunctionTable.DataGetFirst = DL_DataGetFirst;
       FunctionTable.DataGetNext = DL_DataGetNext;
       FunctionTable.DataAbortQuery = DL_DataAbortQuery;
       FunctionTable.FreeUniqueRecord = DL_FreeUniqueRecord;
       FunctionTable.PassThrough = DL_PassThrough;
```

```
        /* set up the module specific info for CSSM */
        memset(&module_funcs, 0, sizeof(module_funcs));
        module_funcs.ServiceType = CSSM_SERVICE_DL;
        module_funcs.DlFuncs = &FunctionTable;

        /* ok, now set up the registration structure for CSSM */
        memset(&reg_info, 0, sizeof(reg_info) );

        reg_info.Initialize        = Initialize;
        reg_info.Terminate         =  Uninitialize;
        reg_info.ThreadSafe        = CSSM_FALSE;
        reg_info.ServiceSummary    = CSSM_SERVICE_DL;
        reg_info.NumberOfServiceTables = 1;
        reg_info.Services          = &module_funcs;

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

        return ret_code;

}
```

3.2 Data Store Operations Example

This section contains a template for the DL_DbOpen function.

```
/*-----------------------------------------------------------------------------
 * Name: DL_DbOpen
 *
 * Description:
 * This function opens a Data store and returns a handle back to the
 * caller which should be used for further access to the data store.
 *
 * Parameters:
 * DLHandle(input)        : Handle identifying the DL module.
 * DbName                 : String containing the logical Data store name.
 * AccessRequest          : Requested access mode for the data store
 * UserAuthentication     : Caller's credentials
 * OpenParameters         : Module-specific parameters
 *
 * Return value:
 * Handle to the Opened Data store.
 * If NULL, use CSSM_GetError to get the follwing return codes
 *
 * Error Codes:
 * CSSM_DL_INVALID_DL_HANDLE
 * CSSM_DL_DATASTORE_NOT_EXISTS
 * CSSM_DL_INVALID_AUTHENTICATION
 * CSSM_DL_MEMORY_ERROR
 * CSSM_DL_DB_OPEN_FAIL
*-----------------------------------------------------------------------------*/
CSSM_DB_HANDLE  DL_DbOpen (CSSM_DL_HANDLE DLHandle,
                           const char *DbName,
                           const CSSM_DB_ACCESS_TYPE_PTR AccessRequest,
                           const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
                           const void * OpenParameters)
{
        if(DLHandle == NULL)
        {
                CSSM_SetError(&dl_guid, CSSM_DL_INVALID_DL_HANDLE);
                return NULL;
        }
        if(DbName == NULL)
        {
                CSSM_SetError(&dl_guid, CSSM_DL_INVALID_DATASTORE_NAME);
                return NULL;
        }
        if(!dl_IfDataStoreExists(DLHandle, DbName))
        {
                CSSM_SetError(&dl_guid, CSSM_DL_DATASTORE_NOT_EXISTS);
                return NULL;
        }

        /*DL specific internal implementation of DbOpen*/

        CSSM_DB_Handle Handle = dl_OpenDataStore(DbName);
        return Handle;
}
```

Appendix A. IBM KeyWorks Errors

This section describes the error handling features in KeyWorks that provide a consistent mechanism across all layers of KeyWorks for returning errors to the caller. All Data Storage Library (DL) 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:

- 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 1 lists the range of error numbers defined by KeyWorks for DL modules and those available for use individual DL modules.

Table 1. DL Module Error Numbers

Error Number Range	**Description**
5000 – 5999	DL errors defined by KeyWorks
6000 – 6999	DL errors reserved for individual DL 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 Data Storage Library Module Errors

Table 2. Data Storage Errors

Error Code	Error Name
5001	CSSM_DL_NOT_LOADED
5002	CSSM_DL_INVALID_DL_HANDLE
5003	CSSM_DL_DATASTORE_NOT_EXISTS
5004	CSSM_DL_MEMORY_ERROR
5005	CSSM_DL_DB_OPEN_FAIL
5006	CSSM_DL_INVALID_DB_HANDLE
5007	CSSM_DL_DB_CLOSE_FAIL
5008	CSSM_DL_DB_CREATE_FAIL
5009	CSSM_DL_DB_DELETE_FAIL
5010	CSSM_DL_INVALID_PTR
5011	CSSM_DL_DB_IMPORT_FAIL
5012	CSSM_DL_DB_EXPORT_FAIL
5013	CSSM_DL_INVALID_CERTIFICATE_PTR
5014	CSSM_DL_CERT_INSERT_FAIL
5015	CSSM_DL_CERTIFICATE_NOT_IN_DB
5016	CSSM_DL_CERT_DELETE_FAIL
5017	CSSM_DL_CERT_REVOKE_FAIL
5018	CSSM_DL_INVALID_SELECTION_PTR
5019	CSSM_DL_NO_CERTIFICATE_FOUND
5020	CSSM_DL_CERT_GETFIRST_FAIL
5021	CSSM_DL_NO_MORE_CERTS
5022	CSSM_DL_CERT_GET_NEXT_FAIL
5023	CSSM_DL_CERT_ABORT_QUERY_FAIL
5024	CSSM_DL_INVALID_CRL_PTR
5025	CSSM_DL_CRL_INSERT_FAIL
5026	CSSM_DL_CRL_NOT_IN_DB
5027	CSSM_DL_CRL_DELETE_FAIL
5028	CSSM_DL_NO_CRL_FOUND
5029	CSSM_DL_CRL_GET_FIRST_FAIL
5030	CSSM_DL_NO_MORE_CRLS
5031	CSSM_DL_CRL_GET_NEXT_FAIL
5032	CSSM_DL_CRL_ABORT_QUERY_FAIL
5033	CSSM_DL_GET_DB_NAMES_FAIL
5034	CSSM_DL_INVALID_PASSTHROUGH_ID
5035	CSSM_DL_PASS_THROUGH_FAIL
5036	CSSM_DL_INVALID_POINTER
5037	CSSM_DL_NO_DATASOURCES
5038	CSSM_DL_INCOMPATIBLE_VERSION
5039	CSSM_DL_INVALID_FIELD_INFO
5040	CSSM_DL_INVALID_ATTRIBUTE_NAME_FORMAT
5041	CSSM_DL_CONJUNCTIVE_NOT_SUPPORTED
5042	CSSM_DL_OPERATOR_NOT_SUPPORTED
5043	CSSM_DL_NO_MORE_OBJECT
5044	CSSM_DL_INVALID_RESULTS_HANDLE
5045	CSSM_DL_INVALID_ATTRIBUTE_NAME
5046	CSSM_DL_INVALID_ATTRIBUTE
5047	CSSM_DL_UNKNOWN_KEY_TYPE
5048	CSSM_DL_BUFFER_TOO_SMALL
5100	CSSM_DL_INVALID_DATA_POINTER

Error Code	Error Name
5101	CSSM_DL_INVALID_DLINFO_POINTER
5102	CSSM_DL_INSTALL_FAIL
5103	CSSM_DL_INVALID_GUID
5104	CSSM_DL_UNINSTALL_FAIL
5105	CSSM_DL_LIST_MODULES_FAIL
5107	CSSM_DL_ATTACH_FAIL
5108	CSSM_DL_DETACH_FAIL
5109	CSSM_DL_GET_INFO_FAIL
5110	CSSM_DL_FREE_INFO_FAIL
5111	CSSM_DL_INVALID_DLINFO_PTR
5112	CSSM_DL_INVALID_CL_HANDLE
5113	CSSM_DL_INVALID_CERTIFICATE_PTR
5114	CSSM_DL_INVALID_CRL
5115	CSSM_DL_INVALID_CRL_POINTER
5116	CSSM_DL_INVALID_RECORD_TYPE
5117	CSSM_DL_DATA_INSERT_FAIL
5118	CSSM_DL_DATA_GETFIRST_FAIL
5119	CSSM_DL_DATA_GETNEXT_FAIL
5120	CSSM_DL_NO_DATA_FOUND
5121	CSSM_DL_INVALID_AUTHENTICATION
5122	CSSM_DL_DATA_ABORT_QUERY_FAIL
5123	CSSM_DL_DATA_DELETE_FAIL

Appendix B. List of Acronyms

API	Application Programming Interface
CA	Certificate Authority
CL	Certificate Library
CRL	Certificate Revocation List
CSP	Cryptographic Service Provider
DB	Database
DBMS	Database Management System
DES	Data Encryption Standard
DL	Data Storage Library
DLI	Data Storage Library Interface
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
LDAP	Lightweight Directory Access Protocol
MAC	Message Authentication Code
ODBC	Open Database Connectivity
OID	Object Identifier
PKCS	Public-Key Cryptographic Standard
SDSI	Simple Distributed Security Infrastructure
SPI	Service Provider Interface
TP	Trust Policy
URL	Universal Resource Locator

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 for 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 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 only 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 (NIST) 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 which 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 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 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 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 ecovery 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 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 Cryptographic Service Provider (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 networking 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 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 smartcards 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.