An Operational View of IBM Keyworks Product

Version 1

Saturday, February 27, 1999

Shabnam Erfani, Michael E. Muresan, Sekar Chandersekaran

1. Introduction

This paper examines the IBM Keyworks framework from an operational point of view while assuming that the reader is already familiar with Common Data Security Architecture (CDSA) terminology. The Keyworks product is based on the CDSA standard from The Open Group and provides complex cryptographic, certificate services and intricate policy enforcement functions transparent to the exploiting application. These functions are motivated by various requirements imposed by architectural issues and import/export regulations for strong encryption. This paper presents a description of these requirements followed by an overview of the architectural decisions in IBM Keyworks product to satisfy these requirements. The first few sections provide some background material and motivation for the later discussions in the paper. The important features in the Keyworks product are key recovery policy enforcement, integrity verification and privilege model that in conjunction address the needs of the market. In addition, to place the discussed features in an appropriate context, the life cycle of a simple encryption application is discussed. The life cycle of a simple encryption application is composed of all the internal events that occur internally at different stages of operation such as framework initialization, CSP module attach, CSP module operation and finally CSP module detach. Each stage of the life cycle is expanded further to discuss the internal details of how various functions are achieved in the framework and how they satisfy the specified requirements in terms of key recovery.

2. Overview of Keyworks Architecture and Requirements

The Keyworks product implementation is based on Common Data Security Architecture (CDSA) standard from The Open Group. The Keyworks Architecture consists of a set of layered security services and associated programming interfaces designed to furnish an integrated set of security capabilities for PKI applications. Each layer builds on the more fundamental services of the layer directly below it.

These layers start with fundamental components such as cryptographic algorithms, random numbers, and unique identification information in the lower layers, and build up to digital certificates, key management and recovery mechanisms, and secure transaction protocols in higher layers. The Keyworks architecture is intended to be a multi-platform security architecture that is both horizontally broad and vertically robust. Figure 1 below shows a simplified view of the layered architecture of a system built on top of Keyworks. There are four major layers in the architecture: Application Domains, System Security Services, Keyworks Framework and Service Providers.

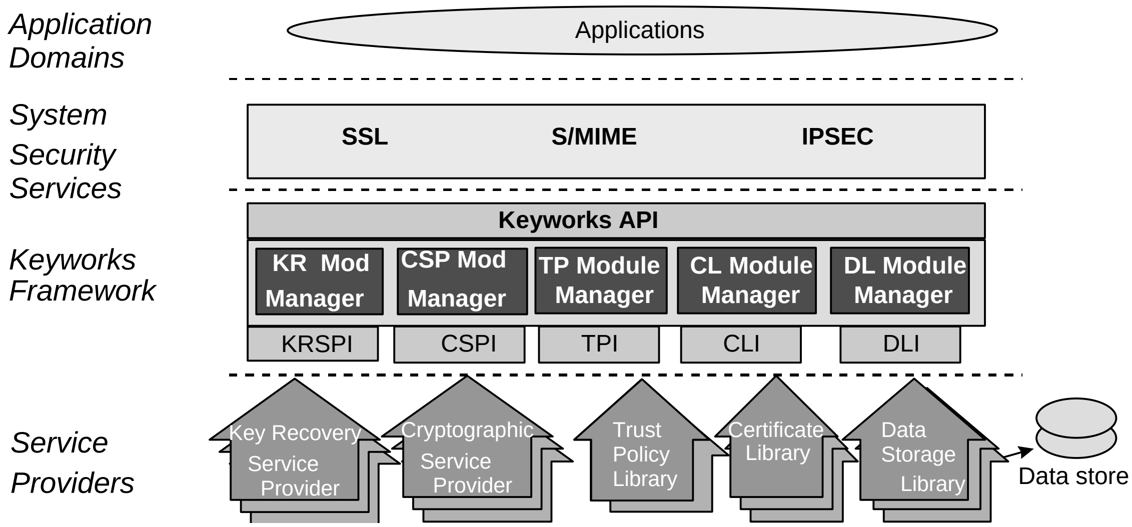


Figure 1. *Keyworks based application architecture*

The Application Domains layer implements the application domain services, such as Secure Electronic Transaction (SET) and E-Wallet, E-mail, or file archival services. The System Security Services layer is between the Application Domains layer and the Keyworks Framework layer. It implements security protocols that are used by the Application Domains layer. Software at this layer may implement cryptographic system security services such as Secure Sockets Layer (SSL), Internet Protocol Security (IPSEC), Secure/Multipurpose Internet Mail Extensions (S/MIME) and Electronic Data Interchange (EDI). The System Security Services layer also includes tools and utilities for installing, configuring, and maintaining the Keyworks Framework and service provider modules. This layer plays an important role in providing secure policy enforcement for different protocols, in particular key recovery. If this layer and the layer underneath it (Keyworks) ensure that key recovery protocol is properly enforced and executed and the key recovery blocks are delivered to their intended destination through some channel, the applications can treat them as trusted protocol handlers. As a result, the application is free from the responsibility of providing key recovery. Furthermore, the functionality can be reused under many applications with little code impact, as this layer can be plugged in underneath the application layer.

The framework component in Keyworks product is a central component of this extensible architecture that provides mechanisms to dynamically manage service provider modules. The framework defines a common security application-programming interface (API) that must be used by the applications to access services of service provider modules. Applications request security services through the API or through system security services implemented over the API. The framework also defines a service provider interface (SPI) through which API calls are dispatched to the service providers that perform the requested function. The framework embodies a number of module managers that perform function call dispatch, module management and policy enforcement functionality for each category of services as shown in Figure 1. In particular, the role of Key Recovery Module Manager (KRMM) and CSP Module Manager (CMM) will be discussed in detail later in the paper.

There are many advantages to using a framework-based architecture for providing security services such as cryptography. One important advantage is the decoupling of applications from the Cryptographic Service Providers. The introduction of the framework layer allows different CSPs to be plugged underneath the framework while complying with a common interface, hence shielding the application from specific CSP dependencies. In addition, the framework layer provides a medium for policy enforcement and tight control of application privileges. This aspect was used to architect a solution that satisfies the requirements needed by strong encryption export/import regulations while maintaining modularity and encapsulation in the architecture. As a result, the Keyworks product can offer a strong cryptography solution that is easy to export and import, customizable for different jurisdictions and does not require the application or service providers to be changed for policy enforcement.

The Bureau of Export Administrations (BXA) which is part of the US Department of Commerce defines the U.S. requirements for the export of strong encryption products. The US government has defined strong encryption based on the number of bits used as the key for various encryption algorithms as shown in Table 1.

Algorithm	U.S. Key Size Greater than:	France Key Size Greater than:
DES	56	40
Triple-DES	56	40
RC2	56	40
RC4	56	40
RC5	56 /12 rounds	40/12 rounds
RSA	512	--

Table 1. *U.S and France Strong Encryption Requirements*

Products offering strong encryption developed inside the US can not be exported without supporting some form of data recoverability, either using key recovery or other means, for law enforcement organizations. The complete list of US export requirements for these products is as follows:

1. The key needed to decrypt a ciphertext shall be accessible through a key recovery method via Key Recovery Agents (KRA) acceptable to the Department of Commerce
2. Strong cryptographic functions shall be inoperable till the key is recoverable.
3. The key shall be accompanied with the identity of the Key Recovery Agent (KRA) that is able to recover the key, and should be sent over the wire reasonably frequently.
4. The key recovery feature should allow access to the key regardless of whether ciphertext was generated or received.
5. Key Recovery function shall be allowed during a period of authorized access without repeated presentations to the KRA.
6. Key recovery enabled products shall not interoperate with products that have been tampered with, bypassed or disabled for key recovery.
7. Key recovery enabled products can interoperate with a non-key recovery enabled product by providing access to keys used for strong encryption.
8. The product shall be resistant to tampering, disablement or circumvention of the KR feature

In addition, countries that use or import the strong encryption products have requirements or definition of strong encryption, which are different from what is mandated by the U.S. government as shown in Table 1 for France. The countries where the product is used also need the flexibility to define their own key recovery policies and criteria and possibly be more restrictive than the exporting country's requirements. Therefore, a product that is offering strong encryption shall satisfy not only the US exports requirements for key recovery but also the usage requirements of the importing countries. Since a country typically designates a geographical location that may not be applicable to legal matters, we use the term jurisdiction in the rest of this paper to designate areas where a set of legal regulations is enforced. The manufacturing jurisdiction designates the country where the product is manufactured, and the usage jurisdiction is where the product is being used.

Furthermore, in some jurisdictions classes of applications are exempt from key recovery or equivalent requirements. For example, financial applications are exempt from the US export regulations. In other words financial applications can exploit strong encryption without generating key recovery blocks. The solution is required to cater to both manufacture and usage jurisdiction policies and requirements, as well as allow exempt applications to bypass the policy enforcement in a safe manner.

The architectural requirement on the solution mandates that it shall provide a flexible, configurable mechanism for key recovery that complies with both manufacturing and jurisdiction policies. The policy configuration and enforcement mechanism should be independent from both the application and the cryptographic service provider to minimize the code impact if the policies change. Moreover, the solution should be such that all existing CSPs can be approved for export/import with minimal change. All components that embed the key recovery functions shall be trusted based on strong integrity checking. These components shall interact with each other only after bilateral integrity verification and authentication.

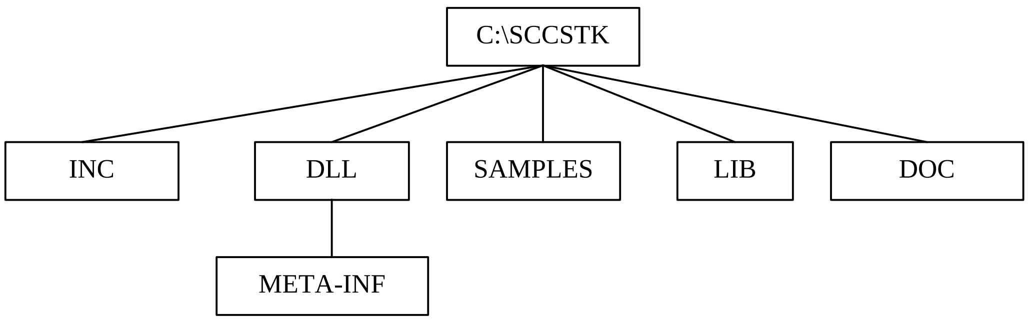
In the following sections of the paper, the components of the Keyworks product architecture that address the above requirements are discussed in detail.

3. IBM Keyworks Installation

The IBM Keyworks product comes in three separate packages. The first package contains all the components needed for installation of the framework and the service provider modules in the product, which collectively are refereed to as the Keyworks toolkit. The second package contains the installation components of the IBM SKR Key Recovery Service Provider (KRSP) and the corresponding configuration files. The third package provides an installation image for the Key Recovery Server, which is not discussed in this paper. A policy customization disk also accompanies the export Keyworks toolkit package. These components are installed in the following order:

- The Keyworks toolkit image is installed first
- The customization disk is used to install the local key recovery policies in the system. The framework will not be functional till policy files are properly installed in the system.
- If desired, the KRSP image can be installed in the system.

Each installation package contains the DLLs for the corresponding components. All the DLLs are accompanied by a set of self-protecting credentials generated when the module is signed. The installation procedure is the process of copying the modules from the install package to the desired directory on disk and registering the paths and other relevant information in the system registry, or equivalent of system registry on non-Windows platforms. By default, the product is installed in C:\sccstk directory on Windows and special directories are created for include files, libraries, samples and documents relative to the package installation directory. For example, if the default installation directory (c:\sccstk) is used, the included files will be in c:\sscstk\inc. The installed modules (DLL's) are located in the \DLL directory, and their credentials are copied to a subdirectory called meta-inf. The installation path is entered in the system registry and by default, all signed modules access their credentials by appending meta-inf to their registered path. After installation on Windows the following directory structure shall be present on disk:



The installation package also copies cssm32.dll (the main DLL that contains the framework code) and the policy modules cssmmanp.dll (manufacturing jurisdiction policy module) and cssmusep.dll (usage jurisdiction policy module) to the windows\system directory, so they can be found by the operating system at start of the application. The corresponding meta-inf (credentials) directory is also copied. At this point, the copying is done and all the paths for various components are updated in the registry.

The IBM Key Recovery Service Provider also needs to have a set of configuration files and the corresponding credentials to be installed in the system. These configuration files contain approved anchor certificates as recommended by the key recovery jurisdictions and KRA certificates as well as mandatory jurisdiction types for key recovery that are needed for generation of key recovery blocks as described in Appendix 2. By default, the configuration files and their credentials will be installed under \skrcfg and skrcfg\meta-inf directories relative to the package installation directory on disk. This path however can be reconfigured in the registry with a manual installation.

4. Key Recovery Management and Configuration

The key recovery (KR) feature in the IBM Keyworks is implemented based on the API/SPI definitions in the CDSA standard. The API functions for KR allow the application to request generation and configuration of key recovery blocks (KRB) by the KRSP. The framework acts as a controller that ensures proper generation of KRBs before strong encryption according to two policies: *manufacturing jurisdiction law enforcement (LE_MAN)* policy, and *usage jurisdiction law enforcement (LE_USE)* policy. In other words, the framework ensures that if either of these policies requires KR, application access to strong encryption is disabled till the appropriate KRB is generated successfully. The key recovery policies also mandate what fields should be present in the key recovery block. Normally, if KR is needed, law enforcement organizations of manufacturing and usage jurisdictions always can have the ability to recover the key from the KRB through the Key Recovery Server (KRS). However, the usage jurisdiction can effectively prevent manufacturing jurisdiction's access to key recovery by manipulating KRSP configuration files as described in Appendix 2. The usage jurisdiction can supply KRA certificates for both manufacturing and usage jurisdiction, hence preventing the LE_MAN from key recovery without cooperation of LE_USE. Based on the configuration of the KRSP, the KRB can be generated as to provide access to the key for the enterprise that uses the software as well. This functionality is achieved based on the algorithm used in the KRSP. The KRSP that accompanies IBM Keyworks implements the IBM SKR algorithm and is described in Appendix 2. In this section we mainly focus on how key recovery policies are created, installed, and enforced within the framework while treating the internals of KRSP and KRB generation abstractly.

The IBM Keyworks toolkit implements a customizable set of key recovery policies based on the requirements of the legal jurisdictions which apply to the software and the requirements of any enterprise in which the software is used as illustrated in figure 2. The key recovery policy has three primary components, LE_MAN, the law enforcement requirements for the jurisdiction of manufacture, LE_USE, the law enforcement requirements for the jurisdiction of usage, and ENT, the requirements for the enterprise. Depending on the usage jurisdiction preferences, LE_MAN and LE_USE can have different policy modules. On the other hand, it is possible for LE_USE to follow the LE_MAN policy and vice versa with prior agreements between the two jurisdictions.

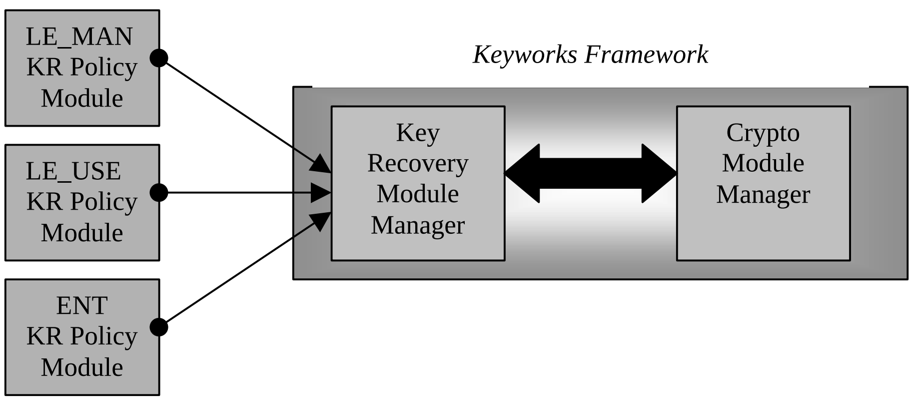


Figure 2. *Key Recovery Policy Tables in the framework*

Key recovery policies are implemented as DLLs containing a series of rules that define exactly what key lengths and algorithm parameter combinations are considered strong encryption for each jurisdiction. During framework initialization, the framework Key Recovery Module Manager (KRMM) loads at least the LE_MAN and LE_USE DLL's and their credentials into a framework internal policy table called the Key Recovery Policy Table (KRPT). Based on the particular requirements of the enterprise using the product, the enterprise policy module can also be loaded into the KRPT at the same time. The framework queries the system registry for the path to where the enterprise policy module is installed. If the path is present, the enterprise policy module DLL is loaded; otherwise the framework implicitly assumes that there is no policy module for enterprise. The KRMM needs to ensure that the policy modules are not modified and tampered with and signed by the IBM code signing anchor key. The discussion of how the integrity and trustworthiness of the policy modules are established is postponed to the next section, where the topic is addressed in depth.

Each policy DLL exports a single function called by the framework during its initialization. The exported function is called CSSM_Man_Krpt and CSSM_Use_Krpt respectively for LE_MAN and LE_USE modules and is defined as:

```
CSSM_RETURN CSSM_Man_Krpt (          KRPOLICYADDFUNC cssm_kr_policy_add,
                                     char *policy_type);
```

The first parameter, cssm_kr_policy_add, is a function which allows the KR policy module to add rules to the framework's KRPT. The second parameter, policy_type is a string to hold the name of the current KR policy module, such as "us_domestic" for the United States policy for domestic usage, or "fr_export" for the French policy for exported software.

The KRPOLICYADDFUNC type is defined as a function pointer by the framework, and has the following form:

```
typedef CSSM_RETURN(CSSMAPI *KRPOLICYADDFUNC)(uint32 AlgorithmId,
                                              uint32  Mode,
                                              uint32  MaxKeyLength,
                                              uint32  MaxRounds,
                                              uint8   WorkFactor,
                                              uint8   PolicyFlags,
                                              uint32    AlgClass);
```

The parameters thus provide all of the information necessary to create single rules to define the limits of strong encryption for each jurisdiction. The parameters are by the framework as follows:

- *AlgorithmId*: The encryption algorithm to which the rule applies, defined by specifying the CSSM algorithm identifier, e.g. CSSM_ALGID_DES.

- *Mode*: The mode to which the rule applies, defined either by specifying a CSSM mode identifier such as CSSM_ALGMODE_CBC or by specifying the wildcard mode identifier CSSM_ALGMODE_WILDCARD which means that the rule applies to all applicable modes of the given algorithm.

- *MaxKeyLength*: This is the maximum key length to be considered as weak encryption. In other words, specifying 56 for MaxKeyLength would mean that key lengths of 0-56 bits would be permitted as weak encryption, and key lengths greater than this number are considered strong encryption, hence requiring key recovery.

- *MaxRounds*: This is the maximum number of rounds permitted for an algorithm to be considered weak encryption. This is only applicable to algorithms which have rounds as a parameter such as RC5. If a given rule defines both *MaxKeyLength* and *MaxRounds*, a cryptographic operation exceeding **either one** of the limits will be considered strong encryption. For example, suppose a rule for RC5 specifies a *MaxKeyLength* of 56 and a *MaxRounds* of 12. In that case, encryption with RC5 is only allowed for operations with a key length of 0-56 and a number or rounds of 0-12. Key size of 64 **or** rounds of 15 would make the operation into a strong encryption that would require key recovery.

- *PolicyFlags*: Defined as either KR_LE_MAN or KR_LE_USE, this tells which policy it was which provided this particular rule. This is useful because if a cryptographic operation is defined as strong encryption by the LE_MAN policy, but only by the LE_USE policy, then key recovery blocks must be generated which meet the requirements of the jurisdiction of manufacture, but not the jurisdiction of usage.

- *AlgClass*: This parameter specifies whether the algorithm to which the rule pertains is symmetric or asymmetric and is set to a CSSM algorithm class identifier such as CSSM_ALGCLASS_SYMMETRIC or CSSM_ALGCLASS_ASYMMETRIC.

The following is an example of code which implements the key recovery policy currently required by the US government to meet export approval as shown in table 1:

```
CSSM_RETURN CSSM_Man_Krpt (       KRPOLICYADDFUNC cssm_kr_policy_add,
                                  char *policy_type)
{
        /* set the policy type */
        strcpy(policy_type, “us_export”);

        /* the US export policy */
        /* KR is required for DES with key length greater than 56, all applicable modes */
        cssm_kr_policy_add(     CSSM_ALGID_DES,
                                CSSM_ALGMODE_WILDCARD, 56, 0, 0,
                                KR_LE_MAN, CSSM_ALGCLASS_SYMMETRIC);

        /* KR is required for triple DES with key length greater than 56, all applicable
        modes */
        cssm_kr_policy_add(     CSSM_ALGID_3DES_3KEY,
                                CSSM_ALGMODE_WILDCARD, 56, 0, 0,
                                KR_LE_MAN, CSSM_ALGCLASS_SYMMETRIC);

        /* KR is required for RC2 with key length greater than 56, all applicable modes */
        cssm_kr_policy_add(     CSSM_ALGID_RC2,
                                CSSM_ALGMODE_WILDCARD, 56, 0, 0,
                                KR_LE_MAN, CSSM_ALGCLASS_SYMMETRIC);

        /* KR is required for RC4 with key length greater than 56, all applicable modes */
        cssm_kr_policy_add(     CSSM_ALGID_RC4,
                                CSSM_ALGMODE_WILDCARD, 56, 0, 0,
                                KR_LE_MAN, CSSM_ALGCLASS_SYMMETRIC);

        /* KR is required for RC5 with key length greater than 56, all applicable modes
        and number of rounds greater than 12 */
        cssm_kr_policy_add(     CSSM_ALGID_RC5,
                                CSSM_ALGMODE_WILDCARD, 56, 12, 0,
                                KR_LE_MAN, CSSM_ALGCLASS_SYMMETRIC);

        /* KR is required for RSA with key length greater than 512 */
        cssm_kr_policy_add(     CSSM_ALGID_RSA,
                                CSSM_ALGMODE_WILDCARD, 512, 0, 0,
                                KR_LE_MAN, CSSM_ALGCLASS_ASYMMETRIC);

         return CSSM_OK;
}
```

In the case of new algorithms that will appear in the future, as support is added to the framework the policy files need to be modified to define a rule for the treatment of the new algorithm.

As mentioned before, depending on the requirements of the enterprise that uses the software, there could be an enterprise policy module containing the rules under which the enterprise desires key recovery block generation. The enterprise KR policy is implemented as a DLL whose path and filename are stored in the system registry (or an equivalent system utility that keeps track of various installations in the system). If this DLL exists, it is loaded during the initialization of the framework, and is kept loaded for the duration of the framework's operation. The DLL exports a single function called EnterpriseRecoveryPolicy. The function definition is as follows:

```
CSSM_BOOL EnterpriseRecoveryPolicy(CSSM_CONTEXT_PTR Context);
```

The function takes as its argument a copy of the cryptographic context, and based on whatever rules the enterprise chooses to implement, determines whether key recovery is required for that context or not. If KR fields are required, the function is to return CSSM_TRUE. If the cryptographic operation can proceed as is, the function returns CSSM_FALSE.

It is worthwhile to mention that an individual using the product can also request generation of key recovery block. The current implementation does not require a policy module for individuals, as the KRB generation can also be requested using the API options for Individual and the other three jurisdiction types.

The KR policy files are signed and stored along with their credentials on disk. The framework will pass the initialization stage only if the policy files and credentials for LE_MAN and LE_USE are present and the signature in the credentials is trusted and verified by the framework, otherwise the framework will not be functional. The ENT policy file may or may not be present; however, if present it should also be accompanied by credentials that are verifiable.

So far, we have discussed how the key recovery policy is defined, loaded and configured in the framework. Once the applicable policies are determined and loaded into KRPT at initialization, for each cryptographic operation the KRPT is consulted to find out whether KR is required or not at run-time. The rest of this section discusses the policy enforcement mechanism used by the framework to ensure proper generation of key recovery blocks.

Key Recovery Policy Enforcement

The responsibility of enforcing the key recovery policies in the framework is divided between the Key Recovery Module Manager (KRMM) and the Cryptographic Module Manager (CMM) within the framework. KRMM encapsulates the KRPT and provides internal query functions for accessing the contents of the KRPT which are treated as read-only. The CMM on the other hand uses these query functions to determine whether it should allow the current cryptographic operation proceed.

In the Keyworks design all cryptographic operations need a cryptographic context. The context contains all the necessary parameters for completion of a given operation such as the key (or a reference to the key), algorithm identifier, mode, rounds, etc. For example, to perform a symmetric encryption, the application needs to first generate or retrieve the key from a secure storage, create a symmetric context, use the context to encrypt the data and finally delete the context. The context actually is created by the framework context management code and is accessible to the application solely via a handle. Subsequently, the application uses this handle as an argument to the rest of the API function calls and does not have direct write access to the context anymore. The attributes of the contexts can be updated only through API functions that control policy application to the context. This design forms the basis for the key recovery enforcement. The decision to enforce key recovery can be made using the contents of the contexts, and since the context can not be modified directly after creation, it can be marked for key recovery policy enforcement.

The cryptographic context created for encryption/decryption operations in the framework contains all the information necessary to determine whether KR is necessary for the current encryption operation. When the application calls the API function CSSM_CSP_CreateSymmetricContext(), the context management module in the framework creates a structure for the context. Then a KRMM function is called that applies the LE_MAN, LE_USE and ENT (if present) policies to the context. If it is determined that these policies mandate key recovery for this context, the context is annotated. This annotation explicitly signals the CMM that the encryption operation can not proceed by setting two values in the context structure: Usability field for key recovery and Work factor for law enforcement key recovery. These two fields are not available for modification outside the framework since the context is available to the application only through a handle. The framework strictly controls all modifications and updates to the context structure, so the key recovery annotations are protected. Once the annotation is performed successfully, the context manager returns the handle of the newly generated context to the application. If the application tries to encrypt its data using this context handle, CSP Module Manager (CMM) first checks to see whether the corresponding context is annotated. If so, the API function call returns with an appropriate error code, otherwise the call is dispatched to the CSP. This mechanism effectively disables access to strong encryption until a key recovery block is generated successfully by the application.

Once the application finds out that it needs to generate key recovery blocks before performing the encryption operation, it places calls to KR API functions that generate the key recovery block with the appropriate fields. Note that similar to an encryption operation, the KRSP needs to be attached first, a key recovery enablement context created and then the call to the KRB generation API placed. The actual call where key recovery block generation is complete takes the cryptographic context handle as an argument from the application. Therefore, it can internally access the context and clear the KR annotation in the context. As a result, the next time the application tries to encrypt data using that context, the CMM allows the operation to proceed since the KRMM has cleared the annotations after ensuring that the KRB is generated successfully. Figure 2 illustrates the algorithm used for the KR policy enforcement. The KRMM also provides an API call that the application can use directly to find out if it needs to generate the KRB before encryption.

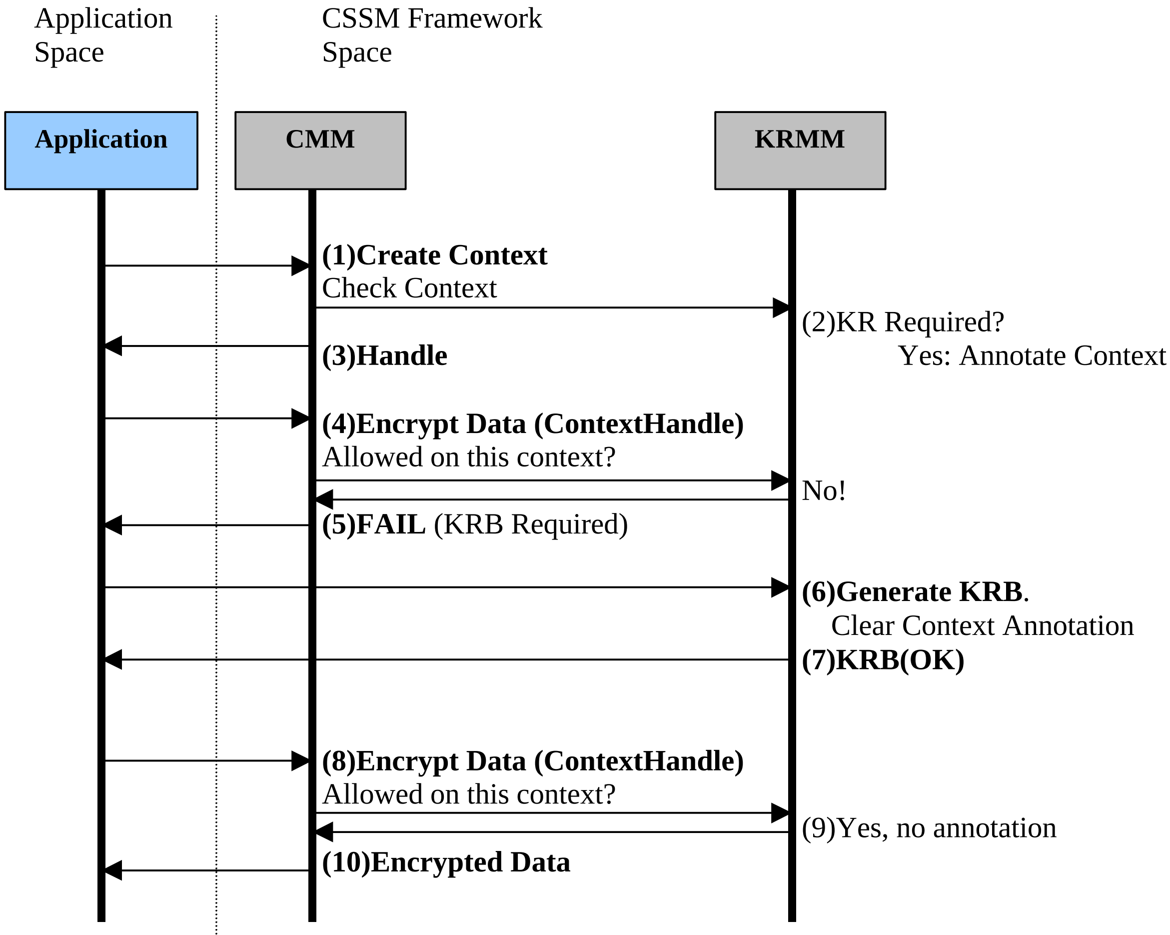


Figure 2. *KR Enforcement protocol*

A possible way for the application to bypass the key recovery policy checks is to perform multiple encryption using weak keys. If the application chains several weak encryption operations together with different keys, it can increase the effective key size to what is considered strong encryption. This idea indeed is used to strengthen DES into triple-DES algorithm, where encryption and decryption operation are chained together with different keys to produce a much stronger encryption than DES. To prevent multiple encryption, the framework has implemented a mechanism that detects encryption requests for previously encrypted data, and prevents the request from proceeding. This mechanism disables multiple encryption attempts that potentially could bypass key recovery enforcement.

Now that we have developed an understanding of the key recovery policy configuration and enforcement in the Keyworks framework, we can look back at the requirements in section 2 and discuss how the current design satisfies them. The method used for policy configuration in Keyworks provides ample flexibility for both usage and manufacturing jurisdictions to enforce local policies. Once the policy files are set in place, the key recovery blocks are generated by the application, otherwise the framework effectively disables the encryption operation for the application. The design is such that once KR is enforced, KRB is generated regardless of the success of the encryption operation and there is no intervention from the KRA each time (refer to

Appendix 2 for more details). The design also takes advantage of an integrity and authentication mechanism, which will be described in section 5. Moreover, the framework provides a general-purpose privilege model that can be customized for applications that can be granted special privileges such as key recovery block generation. The implemented key recovery mechanism satisfies the entire list of specified export requirements, as well as requirements mandated by importing jurisdictions.

5. The Keyworks Privilege Model

Application modules may have special privileges that they can use to obtain special framework services above and beyond other non-privileged applications. The Keyworks framework provides a method for such applications to request special privileges for each thread of execution. An example would be an application that is exempt from key recovery policy enforcement. In this case, the privileged application can place a request to the framework to be exempt from key recovery policy enforcement. The privilege is granted if the following two conditions are satisfied:

- The application credentials are successfully validated by framework
- The application credentials carry a vector of privileges that are equivalent or a superset of the requested privilege

When the applications request the Keyworks framework for a given privilege (with the CSSM_RequrestCssmExemption API), their credentials are verified and the maximum allowed exemptions are determined. As we will describe in Section 6, each application module is signed and given a privilege vector in its credentials. The framework verifies the application credentials and depending on that privilege vector, decides whether the requested privilege can be granted.

The types of privileges available to applications currently are mainly exemptions from various policy checks. Therefore in this paper exemptions and privileges are used interchangeably to describe the Keyworks product design. The current exemptions are defined as:

```
#define CSSM_EXEMPT_NONE                        0x00000000
#define CSSM_EXEMPT_MULTI_ENCRYPT_CHECK         0x00000001
#define CSSM_STRONG_CRYPTO_WITH_KR              0x00000002
#define CSSM_EXEMPT_LE_KR                       0x00000004
#define CSSM_EXEMPT_ENT_KR                      0x00000008
```

The semantics of these privilege types are defined as follows:

- CSSM_EXEMPT_NONE: No privilege can be granted as for exemption from any policy.

- CSSM_EXEMPT_MULTI_ENCRYPT_CHECK: Normally the framework prohibits the multiple encryption of the same data (i.e. encrypting again the result of a previous encryption operation.) This privilege grants exemption from multiple encryption prevention. The policy against multiple encryption is enforced due to the fact that it could potentially increase the effective key size, hence converting weak encryption to strong encryption. For example, chaining three encryption operations with key size of 40 bits and using three different keys practically converts the weak encryption to a much strong encryption with an effective key size much larger than 40. Therefore, multiple encryption can be considered a method of circumventing strong encryption controls and key recovery enforcement and should be prevented.

- CSSM_STRONG_CRYPTO_WITH_KR: This privilege enables the application to gain access to strong cryptographic operations provided key recovery blocks have been generated prior to the encryption. Lack of this privilege in the application module's credentials prevents usage of strong encryption altogether. The rational behind this exemption is to ensure that applications developed outside the US not only generate the key recovery blocks, but also handle them in an appropriate way. For example, a typical application can generate a KRB and use strong encryption, but simply discard the KRB instead of handling it according to the law enforcement requirements. In the export version of the toolkit, an application can not have access to strong encryption unless it is signed with this privilege. Through a review process, it is determined that the application handles the KRB appropriately, and only in that case it is singed with this privilege, hence given access to strong encryption.

- CSSM_EXEMPT_LE_KR: When this exemption is requested, the law enforcement rules that define what strong encryption policies are completely ignored.

- CSSM_EXEMPT_ENT_KR: When this exemption is requested, the enterprise key recovery policy module is no longer consulted to determine whether an operation requires key recovery fields for enterprise in the KRB.

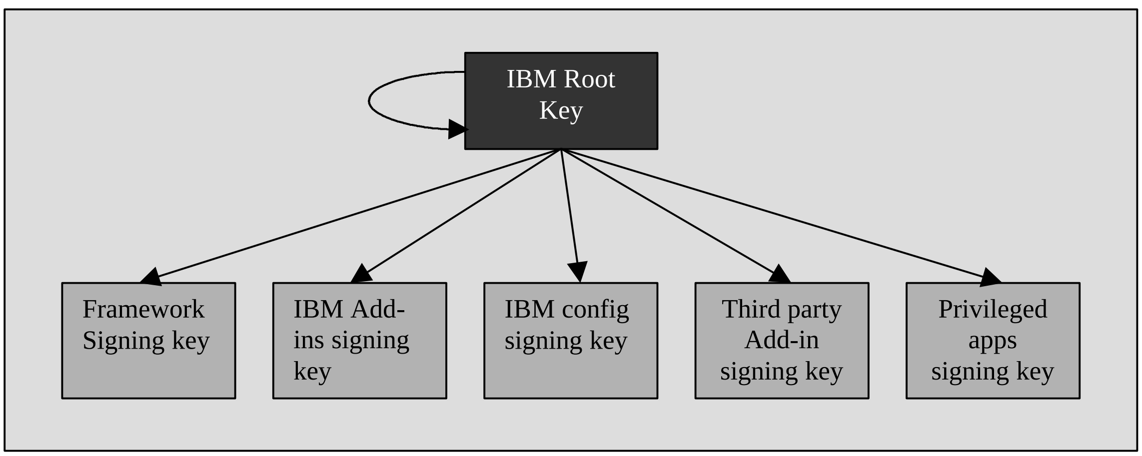
Note that the privilege mechanism in the framework is not necessarily confined to key recovery policy exemption. Other kinds of privileges for the applications can be defined and controlled through the privilege mechanism, for example where a specific service provider would require applications to have special privileges to access them. Each thread of execution should request its privileges from the framework at least once. The framework then associates the appropriate privilege vector obtained from the credentials with the thread identifier. The privileges are subsequently checked every time the thread requests the special operation. If the thread possesses the correct privilege, it will be exempt from the corresponding policy enforcement within the framework. The privileges of a thread do not propagate to its parent, siblings or children.

6. Integrity and Bilateral Authentication

One of the important architectural requirements for a key recovery solution is to provide a mechanism to prevent bypass or tampering of the policy enforcement module and to avoid interoperation with modules that have been tampered with before performing trusted operations. The integrity verification and bilateral authentication mechanism in Keyworks is designed to address these requirements. As an added benefit, this feature has enabled implementation of other features such as the privilege model as discussed in the previous section.

The idea behind the Keyworks integrity model is to form a chain of trust where an application trusts the framework, while the framework trusts a service provider and vice versa, hence different components can interoperate to perform secure operations while ensuring the integrity of one another. Furthermore, once different service providers and the framework mutually perform authentication and verification, the service providers can safely cooperate with each other as well. The design takes advantage of code signing techniques where the root of trust is embedded inside the code and can not be tampered with. Since it is very hard to define a universal root of trust for code signing, each software organization can define the set of public keys it trusts for code signing, which is what IBM has done. Each module can be signed using a known and trusted private key that is vouched for (certified) by the chosen IBM root. After code signing in Keyworks, the module is accompanied with the credentials that contain the DSS signature and the certificate chain (with IBM root as the anchor certificate) that can be used to verify this signature on the module. The module itself can verify the credentials to perform a self-check, or pass its credentials to another module for bilateral authentication. In this section, we describe how the framework performs self-check and then bilateral authentication with the service providers. First, we also briefly describe how the Keyworks modules are signed.

The integrity model in Keyworks heavily relies on the credentials after the object is signed. The object in this context can be a set of files such as KRSP configuration files, or a set of DLLs such as various service provider and policy modules. IBM facilitates the process of signing using a signing facility that allows developers sign their code and have their signing keys certified. The signing facility is a GUI based program that allows DSA Key generation, DSS code signing according to the CDSA standard specifications, and generation of X.509v3 DSA certificates that can be used for signature verification. The hierarchy of the keys used by the Keyworks for code signing is shown in the figure below. Arrow indicates signing relationship in the figure below.



The IBM root key pair is the trusted root of the hierarchy in Keyworks. The root private key signs three other public key certificates whose private key is used to sign Keyworks modules. These three keys are: *framework signing key* that signs the framework and policy module DLLs, *IBM Add-ins key* that signs the IBM service provider Add-in modules in the Keyworks product, *IBM configuration signing key* which is used for signing KRSP configuration files (see Appendix 2 for details). Third parties that develop their own service provider modules can also have their signing key certified by the IBM root key using the signing center.

When a module is signed, a set of credentials is generated for that module and should accompany the object whenever it is loaded and verified. These credentials are composed of three files that reside on disk: two *manifest files*, and a *signature file*. The manifest files contain the hash of the object and information about the signature format and algorithm. The signature file contains a signature of the manifest files, and the X.509v3 DSA certificate chain that can be used to verify this signature. The structure of the credential files and their relationship are depicted the Figure 3 below for cssm32.dll, which is the shared library for the framework on Windows platform. The relationship between the manifest and signature files is such that not only the hash of object is protected, but also the related information such as algorithm identifier and version are preserved. This way the risk of malicious attackers changing the object hash is minimized.

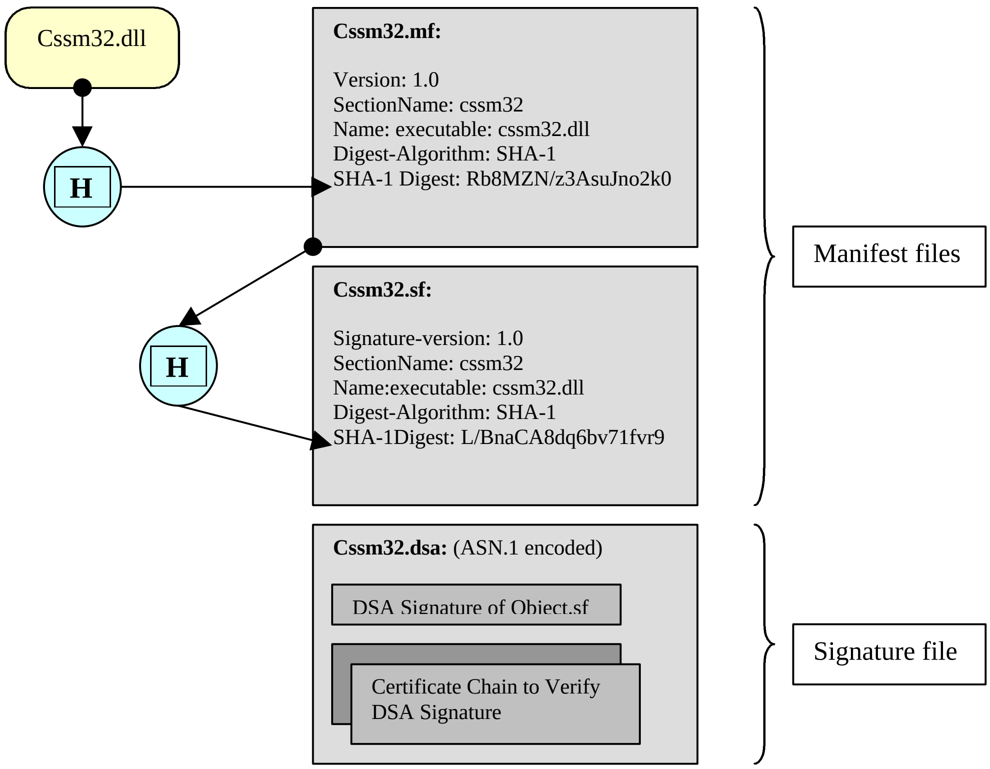


Figure 3. *Module Credentials*

When the framework DLL loads, it first performs a self-check to ensure that it has not been tampered with. To do this, the framework uses the Embedded Integrity Services Library (EISL) that is statically linked in. The EISL embodies all the functions needed to load and verify module credentials and is statically linked in to provide a tamper-proof integrity verification kernel trusted to perform signature verification.

The EISL Self-Check in the CSSM framework is accomplished by calling the EISL API function ISL_VerifyAndLoadModuleAndCredentials, which performs the following two steps:

– *Credential Verification and Identity Establishment*: The EISL code contains the IBM root public key that can be used to verify the anchor certificate in the certificate chain in the signature file cssm32.dsa. The assumption is that the leaf certificate in the chain contains the public key that is used to sign the manifest files. This ensures that the signature and the signing certificate chain in the credentials have not been swapped with a fake one. If the certificate chain verification fails, the module is not loaded. Next, the EISL uses the leaf certificate to verify the signature on cssm32.sf. If the signature verifies correctly, a hash of cssm32.mf is computed and compared to the one contained in cssm32.sf. If they match, credential verification is complete and it is established that IBM vouches for the signer of the object, hence the signature can be trusted by the framework.

– *Module Integrity Verification:* The EISL computes a hash of the cssm32.dll in memory and compares it to the value in the cssm32.mf file. If the two values match, the self-check passes and module load is successful.

The EISL uses the same mechanism to verify any module that is accompanied with its credentials. Some of the other modules that are verified in the framework using the EISL include key recovery policy files as we discussed in Section 3, service provider modules, and privileged applications.

The next step that occurs after the framework successfully is initialized is to attach a service provider such as a CSP. At attach time, the CSP and the framework perform bilateral authentication to avoid rogue applications taking advantage of the CSP services. The service provider also needs to be linked in with the EISL to be able to perform self-check. Bilateral authentication is composed of the following steps:

- Framework has performed a self-check at initialization as we described at the beginning of this section. Therefore it can be treated as a trusted entity.
- Framework reads the installation path of the CSP modules and credentials from system registry loads the module and verifies the credentials and integrity of the CSP just as it would perform a self-check. If the verification is successful the authentication procedure can proceed, otherwise the attach fails.
- Every Add-in module contains a function called “Add-inAuthenticate” that encapsulates the EISL functions that the Add-in needs to perform. The framework finds the address of this function, ensures that is lies within the module boundaries in memory to ensure that it is not a fake call. The framework calls the Add-inAuthenticate function.
- The CSP Add-inAuthenticate performs credential verification and integrity check on the framework module in memory. Note that the CSP does not perform a self-check since the framework has already performed that. If the CSP verified independently that the framework is trustworthy, it can also trust that fact that no one has tampered with the CSP module itself.
- If the framework passes the CSP’s integrity check, the framework provides its function table to the CSP. In other words, the framework avails itself to interoperate with the CSP since mutual trust between CSP and framework has been established.
- The CSP provides its own function table to the framework so CSP services can be exploited

Note that this approach works since the CSP does not export any of its entry points. The CSP publicizes its call table (entry points) to the framework only after successful bilateral authentication. As a result, rogue applications fail to obtain the entry points that provide service in the CSP and all calls can only go through the framework.

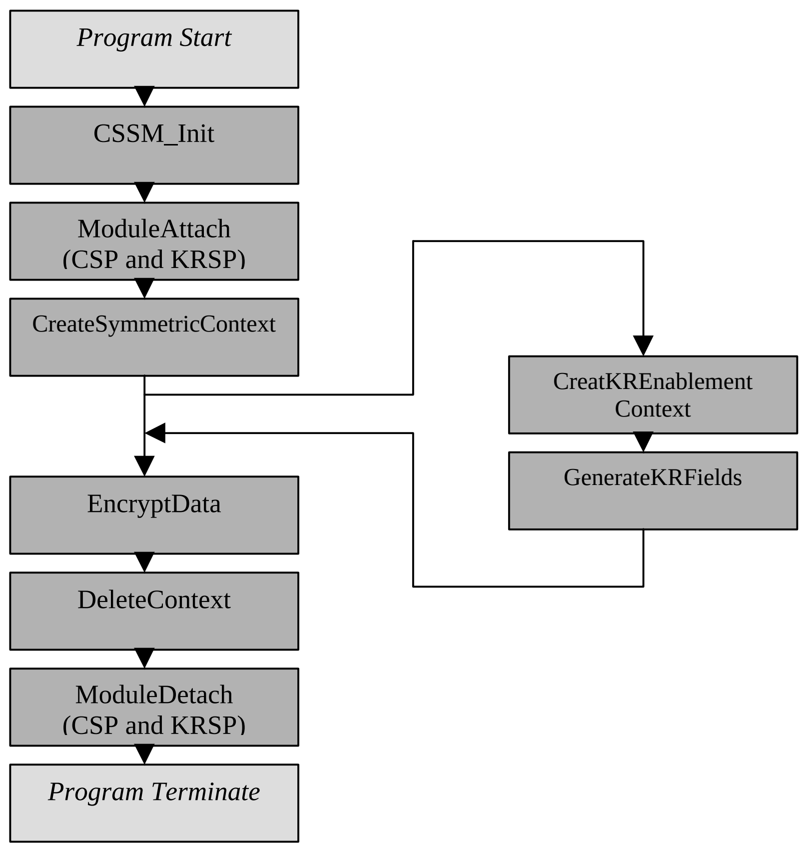
As mentioned in the previous section, all key recovery policy modules are verified before they are loaded into the KRPT. The policy modules are also accompanied by their credentials which are verified by the framework when the modules are loaded, however, there is no need to perform bilateral authentication between the framework and the policy modules. The policy modules can publicize their information, since they are designed to be read-only. Performing signature verification on the policy modules suffices to ensure that the contents have not been tampered with.

Similar to Keyworks modules, the applications that use Keyworks can also be signed and accompanied with credentials. The generated credentials are used to carry the assigned privileges to the application, and provide a means of trust establishment. The framework honors the privileges only if the root of trust signs the credentials, which in case of Keyworks is the IBM root public key.

Other files that are protected via the integrity mechanism are the configuration files for the IBM key recovery service provider (KRSP module). These configuration files contains important information that is needed for correct key recovery block generation such as KRA certificates. The information is not private; however, it should be protected from tampering. In this case, the responsibility of verifying the integrity of the configuration files lies with the KRSP. The mechanism stays the same. The KRSP uses the embedded root key in EISL to first verify the certificate chain in the credentials, then it checks the integrity of the configuration files using the installed credentials. Finally after loading the files it checks the trustworthiness of the contained data independently as is discussed in Appendix 2.

7. Operational View of the Framework

So far we have discussed different aspects of the framework architecture that enable key recovery while satisfying all the requirements. In order to put what we have discussed in context, in this section we present a simple encryption with key recovery example and illustrate how the services we have described so far are activated during the life cycle of our simple encryption example. The actual C program for this example can be found in Appendix 1. We define the lifecycle of a simple encryption application as shown in the figure below:



Each stage in the life cycle is denoted by a call to an API function that triggers a number of events inside the framework. Note that the call to CSSM_KR_CreateKREnablementContext is placed only if key recovery is required. The stages of the life cycle and the corresponding internal events that occur at that point are described below:

1. *CSSM_Init:* The loading of the framework DLLs and application call to CSSM_Init() function start the initialization process of the framework as listed:

   a) Internal CSSM initialization (context management, module management, thread safety, etc.)
   b) CSSM self check is performed as described in Section 6.

c) KR policy modules are loaded verified and added to the key recovery policy table as described in Section 4. The policy tables are cached in and can not be modified till the next time the framework is loaded.
d) All exemptions for the current thread are cleared. These exemptions are instated when the application calls the API function CSSM_RequestCSSMExemption while presenting its credentials. The thread retains these exemptions till its termination or if a new set of exemptions is requested using the same API function call.

2. *CSSM_ModuleAttach():* Before using the services of the CSP and the KRSP, the application needs to attach the module first. The module attach process is composed of the following steps:

   a) Set up internal module management structures
   b) Load and verify module credentials for the CSP and verify the signature on the CSP
   c) Call AddinAuthenticate in the CSP library to perform add-in self-check and bilateral authentication between the CSP and the framework as described in Section 6.
   d) Load and Initialize the CSP Add-in module and return a module handle to the application
   e) Load and verify module credentials for KRSP and verify the signature on KRSP. The KRSP however does not verify the framework as the CSP does.

3. *CSSM_CSP_CreateSymmetricContext():* For every cryptographic operation an appropriate context need to be created. In case of encryption, a symmetric context is created as follows:

   a) Set up context management structure for the new context in the framework
   b) Ask KRMM to apply the relevant KR policies and mark the context if KR is required for the requested operation
   c) Return a handle for the context to the application

4. *CSSM_KR_RecoveryEnablementContex():* Similar to an encryption operation, a context is needed to convey the required parameters to KRSP. The RecoveryEnablementContext contains user profiles for correct key recovery block generation. For more details please refer to Appendix 2. Note that this step is only executed if key recovery block generation is required. The application can call API functions such as CSSM_KR_GetPolicyInfo() on a given context to find out whether key recovery is needed.

5. *CSSM_KR_GenerateRecoveryFields():* A call to this API function starts the key recovery block generation in the KRSP. The KRSP uses the contents of the enablement context and the symmetric context to create the key recovery block (KRB) using the IBM SKR algorithm. For details of this process please refer to Appendix 2. If the KRB generation is successful, the annotation of the symmetric context is cleared by KRMM and the KRB data is returned to the application. If the KRB could not be generate successfully, the annotations are not cleared, and the application can not proceed with the encryption operation.

6. *CSSM_EncryptData():* This API function takes the handle to the context created in the previous step as an argument along with a few other arguments relevant to the operation. The most important event during this call is:

   a) Placing a call to KRMM to check if the requested operation is allowed enforces KR policy. The KRMM performs the following to make the decision:

i) The requesting thread privileges are considered to see if the calling thread possesses KR exemptions. If so, the operation can proceed. Otherwise, proceed to the next check.

ii) Check for KR annotations on the context. If the context is still annotated, the encryption request is not allowed.

b) Check for multiple encryption attempts. If the call is to perform encryption on data that is already encrypted, reject the call. The framework maintains a cache of all previously encrypted data, and if the input data match any of those entries, it is considered a multiple encryption attempt, which potentially can bypass the KR enforcement mechanisms and should be disallowed.

7. *CSSM_DeleteContext():* The context corresponding to the supplied handle is removed from the internal context management data structures

8. *CSSM_ModuleDetach():* Upon module detach all the corresponding internal module management structures are cleaned inside the framework, given that no other reference to the module exists.

8. Conclusion

In this paper we presented an overview of key recovery requirements that are motivated by export and import regulations of different jurisdiction with respect to strong encryption. We further discussed the motivation and design of features in the Keyworks product that cooperate to address the requirements and facilitate key recovery in a secure fashion. Using a simple example we provided a context where the role of each feature was illustrated from an operational point of view.

IBM Keyworks product provides a key recovery solution that enforces configurable manufacturing and usage policies for law enforcement (LE_MAN and LE_USE), enterprise (ENT) and individuals (INDIV). The policy configuration mechanism is flexible enough to allow several variations of policy establishment. The LE_MAN and LE_USE can have different policies, or upon jurisdictional agreement reuse each other’s policy. LE_USE also can override the LE_MAN access to the key recovery fields. Furthermore, access to strong encryption is effectively disabled till the key recovery block is generated properly, and no intervention from the designated KRA's is required to complete the KRB generation. In addition, the framework embodies a mechanism for integrity self-check and bilateral authentication with the service provider modules that prevents tampering and bypass of key recovery enforcement mechanisms and interoperation with non-trusted or tampered modules. The collection of all the above mentioned features cooperate to satisfy all the requirements mandated by jurisdictions where the Keyworks product is used, and the manufacturing jurisdiction in a satisfactory fashion.

Appendix 1: Sample Encryption Program

The following C program provides an example of how applications can perform an encryption operation. In this example, we attempt to perform an encryption operation that does not need key recovery. We also provide an example where key recovery if performed.

```
/* sample encryption program without key recovery */
CSSM_RETURN encryptdata_example(void)
{
        /* context information */
        CSSM_CC_HANDLE                  ContextHandle;

        CSSM_CRYPTO_DATA                PassPhrase;
        CSSM_DATA                       PassBuf;
        char                            PassData[] = "Your Secret pass phrase";

        /* key used for encryption */
        CSSM_KEY                        EncryptionKey;

        /* buffer used for encryption */
        uint32                          bytesEncrypted;

        CSSM_DATA                       ClearBuf;
        CSSM_DATA                       CipherBuf;
        CSSM_DATA                       RemBuf;
        CSSM_DATA                       InitVector
        unsigned char                   ClearData[256];
        unsigned char                   CipherData[256];
        unsigned char                   RemData[256];
        unsigned char                   DecryptData[256];
        /* dummy IV data */
        unsigned char                   ivData[8] = {0,0,0,0,0,0,0,0};

        /* key generation parameters */
        uint32                          KeyUsage        = CSSM_KEYUSE_ANY;
        uint32                          KeyAttr         = CSSM_KEYATTR_SENSITIVE;

        CSSM_RETURN                     ReturnValue = CSSM_OK;
        int                             keylen;
        CSSM_RETURN                     test_rval;
        CSSM_ERROR_PTR                  ErrorPtr;

        Char                             *DataString = "This is the data to be encrypted";

        CSSM_MODULE_HANDLE              CSPHandle;
        CSSM_GUID                       CSPGUID = IBMSWCSP_GUID;
        CSSM_VERSION                    CSPVersion;
        CSSM_API_MEMORY_FUNCS           MemoryFuncs;
        uint32 SubserviceID             = 0;
        uint32                          SubserviceFlags = CSSM_SERVICE_CSP;
        uint32                          Application = 0;
        const CSSM_NOTIFY_CALLBACK Notification = 0;
        const void *                    Reserved = NULL;

        /* set up the structures needed for CSSM_Init */
        CSPVersion.Major  = IBMSWCSP_MAJOR_VERSION;
        CSPVersion.Minor  = IBMSWCSP_MINOR_VERSION;
```

```
	/* pass the pointers to memory functions that can be used to allocate buffers for the application */
	MemoryFuncs.malloc_func		= app_malloc;
	MemoryFuncs.free_func		= app_free;

	MemoryFuncs.realloc_func		= app_realloc;
	MemoryFuncs.calloc_func		= app_calloc;

	CSPHandle = CSSM_ModuleAttach(		&CSPGUID,
						&CSPVersion,
						&MemoryFuncs,
						SubserviceID,
						SubserviceFlags,
						Application, Notification, Reserved);



	PassPhrase.Param		= &PassBuf;
	PassBuf.Length		= sizeof(PassData);
	PassBuf.Data		= PassData;
	PassPhrase.Callback	= NULL;

	keylen			= 56;


	ContextHandle		=CSSM_CSP_CreateKeyGenContext(	CSPHandle,
								CSSM_ALGID_DES,
								&PassPhrase, keylen,
								NULL,	NULL,	NULL,
								NULL, NULL);



	if (ContextHandle == 0)
	{
		return CSSM_FAIL;
	}


	memset ( &EncyrptionKey, 0, sizeof(EncryptionKey));

	ReturnValue = CSSM_GenerateKey(ContextHandle,
						KeyUsage,
						KeyAttr,
						NULL,
						&EncryptionKey);
	if (ReturnValue != CSSM_OK)
	{
		return CSSM_FAIL;
	}

	ReturnValue	= CSSM_DeleteContext(ContextHandle);
	if (ReturnValue != CSSM_OK)
	{
		return ReturnValue;
	}

	InitVector.Length = sizeof(ivData);
	InitVector.Data = ivData;

	ContextHandle	=	CSSM_CSP_CreateSymmetricContext(	CSPHandle,	CSSM_ALGID_DES,
			CSSM_ALGMODE_CBCPadIV8,
```

```
                        &EncryptionKey, &InitVector, CSSM_PADDING_NONE, 0);

        if (ContextHandle == 0)
        {
                return CSSM_FAIL;
        }

        /* setup the cleartext to be encrypted */
        strcpy(ClearData, DataString);

        ClearBuf.Length   = strlen(ClearData);
        ClearBuf.Data     = ClearData;

        /* set up the buffer for the ciphertext */
        memset(CipherData, 0x00, sizeof(CipherData));
        CipherBuf.Length = sizeof(CipherData);
        CipherBuf.Data    = CipherData;

        BytesEncrypted    = 0;

        memset(RemData, 0x00, sizeof(RemData));
        RemBuf.Length   = sizeof(RemData);
        RemBuf.Data             = RemData;

        ReturnValue       = CSSM_EncryptData(ContextHandle, &ClearBuf, 1, &CipherBuf, 1,
                                &bytesEncrypted, &RemBuf);

        if (ReturnValue != CSSM_OK)
        {
                return CSSM_FAIL;
        }


        ReturnValue = CSSM_DeleteContext(ContextHandle);
        if (ReturnValue != CSSM_OK)
        {
                return ReturnValue;
        }

        ReturnValue = CSSM_ModuleDetach(CSPHandle);

        return CSSM_OK;
}
```

<u>Encryption Example with Key Recovery Block generation</u>:

```
/* This example encrypts a given file while generating the key recovery block. For illustration
purposes only.  */
int main(int argc, char *argv[])
{
   // Handle to the cryptographic service provider
   CSSM_CSP_HANDLE          hCSP;
   // Handle to the key recovery service provider
   CSSM_KRSP_HANDLE         hKRSP;
   char                     *ClearFilename;

   ProcessArguments(argc, argv, &ClearFilename);

   Initialize();

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

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

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

   return 0;
}

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

```
    uint32          j;          // index
    uint32          k;          // index
    uint32          l;          // index

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

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

    if (pModuleList->NumberItems == 0)
    {
        printf("Error: no CSPs installed.\n");
        exit(1);
    }

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

    Found = CSSM_FALSE;

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

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

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

```
                for (l = 0; l < Total; l++)
                {
                    pCap = &(pInfo->CapabilityList[l]);
                    if (pCap->AlgorithmType == AlgorithmRequired)
                    {
                        Found = CSSM_TRUE;
                    }
                }
            break;

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

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

  }

  // Successfully attached to desired CSP
}
```

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

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

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

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

        ChoiceMade = CSSM_FALSE;

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

        while (!ChoiceMade)
        {
```

```
            printf("\n");

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

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

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

            fflush(stdout);

        } // while choice not made

    }

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

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

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

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

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

```
        exit(1);
    }
}

//---------------------------------------------------------------------------
//
// Function: GenerateKeyRecoveryFieldsAndEncrypt
//
// This function encrypts a file using strong encryption.  It performs all
// the necessary prerequisites such as generation of a key (could be replaced
// by string to key derivation) for the encryption, generation of the
// necessary key recovery fields, and actual encryption.  The encrypted
// file and the key recovery field file will be written out.
//
//---------------------------------------------------------------------------
void GenerateKeyRecoveryFieldsAndEncrypt(
    CSSM_CSP_HANDLE hCSP,
    CSSM_KRSP_HANDLE hKRSP,
    char *InputFilename)
{
    FILE            *ClearFile;         // clear file's handle
    CSSM_CC_HANDLE      hCryptoContext;     // context handle for encryption
    CSSM_KEY            Key;                // the symmetric key for encryption
    int             BytesRead;          // byte reading counter
    uint32          BytesEncrypted;     // byte encrypting counter
    unsigned char       ClearBuf[MAX_CLEAR_FILE_SIZE];  // buffer for cleartext
    CSSM_DATA           ClearData;          // buffer for cleartext
    CSSM_DATA           EncryptedData;          // buffer for ciphertext
    unsigned char       RemBuf[DES_PAD_LEN];// buffer for padding
    CSSM_DATA           RemData;            // buffer for padding
    CSSM_DATA           KRFData;            // buffer for key recovery fields
    CSSM_RETURN         RC;                 // return code

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

    GenerateKey(hCSP, &Key);

    GenerateSymmetricContext(hCSP, &Key, &hCryptoContext);

    GenerateKeyRecoveryFieldsForContext(hKRSP, hCryptoContext, &KRFData);

    WriteOutputFile(KRFData, InputFilename, KR_FIELDS_FILE_SUFFIX);

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

    if ((ClearFile = fopen(InputFilename, "rb")) == NULL)
```

```
    {
       printf("Error: could not open %s\n", InputFilename);
       perror("fopen");
       exit(1);
    }

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

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

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

    fclose(ClearFile);

    //
    // Encrypt the buffer
    //

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

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

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

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

    //
    // Write the encrypted file
```

```
    //

    WriteOutputFile(EncryptedData, InputFilename, ENCRYPTED_FILE_SUFFIX);

}

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

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

    *hCryptoContext =
        CSSM_CSP_CreateSymmetricContext(hCSP,
                                        CSSM_ALGID_DES,

             CSSM_ALGMODE_CBCPadIV8,
                                        Key,
                                        &DESIVData,
                                        CSSM_PADDING_NONE,
                                        0);

    if (hCryptoContext == 0)
    {
        printf("Error: could not perform symmetric encryption setup\n");
        pError = CSSM_GetError();
        printf("CSSM_CSP_CreateSymmetricContext error code = %d\n", pError->error);
        exit(1);
    }
}

//---------------------------------------------------------------------------
//
// Function: GenerateKeyRecoveryFieldsForContext
//
// This function generates the key recovery fields associated with a given
// symmetric context.  These key recovery fields can later be used to
// reocover the encryption key by authorized parties.
```

```
//
//---------------------------------------------------------------------------
static void GenerateKeyRecoveryFieldsForContext(
   CSSM_KRSP_HANDLE hKRSP,
   CSSM_CC_HANDLE hCryptoContext,
   CSSM_DATA *pKRFields)
{
   CSSM_CC_HANDLE  hKRContext; // context for key recovery field generation
   CSSM_RETURN     RC;         // return code
   uint32          KRFlags;    // key recovery algorithm flags
   CSSM_ERROR_PTR  pError;     // error info

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

   hKRContext = CSSM_KR_CreateRecoveryEnablementContext(hKRSP, NULL, NULL);

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

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

         KRFlags = KR_LE_MAN | KR_LE_USE | KR_ENT;


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

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

   //
   // Clean up
```

```
    //

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

Appendix 2: IBM Secureway Key Recovery Algorithm and Key Recovery Service Provider (KRSP)

This appendix presents an overview of the implementation and operation of the KRSP. The assumption is that the reader is already familiar with the IBM SKR algorithm and terminology as presented in the paper *"Two-Phase Cryptographic Key Recovery System"* by R.Gennaro, P. Karger, et al. First, two essential operational attributes of the KRSP are discussed: the KRSP configuration, and the key recovery block format and how it correlates to various key recovery operational scenarios. We conclude the discussion by presenting a simple KRB generation example where we show how the contents of the configuration files are used to generate the requested key recovery blocks.

KRSP Configuration

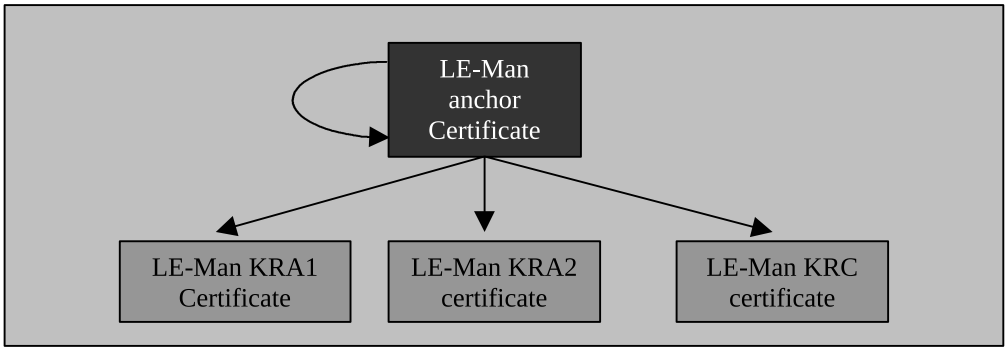
In order to operate correctly, the KRSP needs a set of self-protecting configuration files that contain the necessary information for creation of key recovery blocks. Depending on the key recovery policy, the key recovery block can have multiple parts catering key recovery to different entities. For example, a key recovery block can have a part that allows key recovery for LE_MAN and another that is set up for LE_USE key recovery. In order to set up each field, the KRSP requires identities and public key certificates of the key recovery agents (KRA), and the key recovery center (KRC) to execute the IBM SKR algorithm. Furthermore, the KRSP needs to know what fields are absolutely mandatory in the key recovery block. The collection of all this information constitutes the contents of the KRSP configuration files.

The basic set of files needed for KRSP configuration define the jurisdiction types that need to be present in the KRB and provide the required certificates in ASN.1 encoded format. These files are:

- *Jur-type.cfg:* This file is the master configuration file and contains the jurisdictions that must be present in the KRB every time a KRB generation API is called. In other words, regardless of the information received from the API, the jurisdictions specified in this file will be able to recover the key from the KRB. In the current implementation this file contains at least two jurisdictions: LE_MAN and LE_USE.

- *Enabler.cfg:* This file contains a self-signed X.509v3 certificate that is used for the boot-up of KRSP. During initialization, KRSP looks for the enabler certificate and validates the signature and validity date. If any of these steps fail, KRSP fails to initialize.

- *LE-man.cfg and LE-Use.cfg:* These two files contain an ASN.1 encoding of a structure that contains the certificate chains for the corresponding KRA and KRC entities. The law enforcement of each jurisdiction designate entities to act as key recovery center for them and the public key certificate of the approved entities are used to create the LE-*.cfg file. The certificate hierarchy contained in these files is shown in the figure below. Note that the number of KRA's is not limited to two, and the arrow in the figure shows a signing relationship.



Every jurisdiction entity can choose a root of trust authority to act as the anchor certificate. The chosen anchor certificate then can sign the KRA and KRC certificates used for KRB generation in IBM KRSP. All of these certificates are X.509v3 certificates containing 1024 bit RSA public keys of the trusted Anchor, KRA's and the KRC. LE_MAN and LE_USE can have totally different anchor certificates, or upon prior agreement use the other entity’s anchor certificate.

- *Ent.cfg:* Optionally, the ent.cfg can also be present in the set of configuration files. In this case, the ENT entry should also be present in the jur-type.cfg. The contents of ent.cfg are similar to that of law enforcement configuration files except that the enterprise anchor certificate is different from those of law enforcement and KRA's and the KRC are approved by the enterprise.

- *Ent-authinfo-hash.cfg*: If the enterprise desires key recovery, and the ent.cfg file is present there is a need for internal authentication for the enterprise. This file contains a SHA-1 hash of a secret known by the enterprise administrator (key recovery officer) only. This hash is incorporated in to the key recovery field of the KRB, and unless it matches the hash of the acting administrator’s password key recovery is not allowed. This process is to safeguard against malicious key recovery within the enterprise, since with this measure in place only the designated key recovery officer can start the key recovery process in the enterprise key recovery server.

Once the KRSP configuration files are generated, they are signed with the same key that is used to sign the KRSP module. The generated credentials have the same format as discussed in Section 6 and are placed in the meta-inf directory under the directory where the configuration files are installed.

When the application attaches the KRSP module by placing a call to CSSM_ModuleAttach() and supplying the IBM KRSP GUID as an argument, the KRSP starts its initialization. As part of this initialization, the configuration files are verified using their credentials, ASN.1 decoded and cached into memory till the KRSP is detached. Therefore the configuration files can not be changed at run-time. Furthermore, the KRSP also verifies the certificate chains in the configuration files before using them following the normal X.509v3 certificate verification procedures.

The law enforcement certificates in the current implementation are always retrieved from the configuration files LE-MAN.CFG and LE-USE.CFG. For enterprise, however, there is an option to get the certificates from the API data structures, as we will discuss later in the appendix. The ent.cfg files provides the default certificates, but are overridden if a different set of valid certificates are passed to the KRSP through the API. For individuals who desire key recovery, the only option is to obtain the certificates through the API call. There are no configuration files for individuals.

KRB Format

The key recovery block contains the protected key that can be recovered by different jurisdictions such as LE_MAN, LE_USE, Enterprise, and individual. The key recovery block format used in the KRSP is an implementation of the Common Key Recovery Block (CKRB) draft standard from the Open Group. This format as implemented by IBM is shown below:

CKRB Version # = 1.0 (2 bytes)	CKRB Length (4 bytes)
CKRB OID Length (2 bytes)	CKRB OID Value(8 bytes) IBM = 0x19970910
Opaque KRF Length (4 bytes)	
Opaque KRF Value- ASN.1 Encoded (Variable Length)	
Integrity Type Selected From CKRB (2 bytes) IBM = SHA-1 keyed hash	KRF Integrity Type Length (2 bytes)
Opaque KRF Integrity Value	

Notes:

1. All length values are represented in network byte order.

The IBM implementation is similar to the CKRB definition, except that due to lack of standard values for version number and OID fields IBM SKR implementation has defined its own values. The CKRB fields are defined as follows:

- *CKRB Version:* Indicates the version (major and minor) number of the KRB
- *CKRB Length:* contains the length of the KRB in bytes
- *CKRB OID Length:* contains the length of an object identifier that defines the type of key recovery method used.
- *CKRB OID:* contains the object identifier for the opaque key recovery field

- *Opaque KRF Length:* the length of the opaque KRF which is actually the SKR key recovery block
- *Opaque KRF:* a stream of bytes that actually contain the SKR key recovery block in an ASN.1 encoded format
- *KRF Integrity type:* the type of integrity protection used to protect the KRB, which in this case is a keyed hash of the Opaque KRF
- *KRF Integrity Length:* the length of the keyed hash value, which is 20 since we are using SHA-1 hash algorithm
- *KRF Integrity value:* keyed hash of the Opaque KRF and the session key. This value effectively ties the generated key recovery block to the session key. This is so the receiving end can verify that this key recovery block indeed corresponds to the curent session key.

The CKRB is a container for the actual key recovery block data. The opaque KRF can be handled according to the type indicated by the OID field. In the case of IBM KRSP, the KRF contains an ASN.1 encoding of the IBM SKR key recovery block, which itself is composed of two blocks: B1 and B2. The IBM SKR sub-block formats are shown below:

B1						
Header	Sender Info	Recvr Info	LE Use Set	LE Man Set	Ent Set	Indiv Set
Version # Time Stamp	Sender Name	None	KRC Info – KRA Name – Public key – Auth Info: PKEncrypted [UserName, Host Name, Host Address] KRA Info – Public Key – Auth Info: PKEncrypted [UserName, HostName, Host Addres] – Wrapped KGInfo	KRC Info – KRA Name – Public key – Auth Info: PKEncrypted [UserName, Host Name, Host Address] KRA Info – Public Key – Auth Info: PKEncrypted [UserName, HostName, Host Addres] – Wrapped KGInfo	KRC Info – Public key – Auth Info: Hash[Ent domain auth info] KRA Info – Public Key – Auth Info: Hash [Ent Domain auth info] – Wrapped KGInfo	KRC Info – Public key – Auth Info: Hash[Indiv auth info] KRA Info – Public Key – Auth Info: Hash [Indiv auth info] – Wrapped KGInfo

B2				
Header	LE Use Set Info	LEManSetInfo	EntSetInfo	IndSetInfo
Session key header	Nested Wrapped Session Key	Nested Wrapped Session Key	Nested Wrapped Session Key	Nested Wrapped Session Key

The B1 and B2 blocks are appropriately filled in by the KRSP and encoded to generate the KRF field in the CKRB. Both blocks provide containers for all supported four jurisdictions, and the KRSP fills in the relevant container according to the contents of jur-type.cfg file.

The B1 block is used to transport all authentication and identification information in the key recovery block. This information includes KRA names and public keys for each jurisdiction, authentication information indicating the source of the KRB, and similar information for KRC. The B2 block contains the nested wrapped session key, which is basically the session key encrypted with public keys of jurisdiction KRAs and KRCs, as described in IBM SKR paper referenced in the beginning of this appendix.

In order to provide key interoperability between distributed components of the key recovery system, i.e. KRSP, and the key recovery center, we have defined a portable key format. This format contains the cryptographic material of the key as well as all other attributes and parameters needed for correct decryption operation once the key is recovered. The nested wrapped session key in B2 fields is in the portable key format, therefore, after recovery it can easily be used to recover the cleartext.

KRSP Operational View

The KRSP operation similar to any other service provider module starts when the module is attached by the application calling CSSM_ModuleAttach(). This call triggers the initialization process inside the KRSP that includes loading and validation of the configuration files as well as verification of certificate chains in the configuration files. If the validation is successful, the certificates are cached in the KRSP for later use.

The KRB generation for IBM SKR KRSP requires creation of a key recovery enablement context by the application. Similar to cryptographic operations, KRB generation requires a set of parameters and attributes from the application. The KR enablement context acts as a container that groups the required parameters together for the benefit of the KRSP. The API function for creating the context is:

```
CSSM_CC_HANDLE CSSMAPI CSSM_KR_CreateRecoveryEnablementContext(
                                    CSSM_KRSP_HANDLE KRSPHandle,
                                    CSSM_KR_PROFILE_PTR LocalProfile,
                                    CSSM_KR_PROFILE_PTR RemoteProfile);
```

The local profile and remote profile arguments to this function contain fields that indicate the types of desired key recovery fields (LE_MAN, LE_USE, enterprise, individual), KRA certificate chain for each type, and user name and public key certificate. The LocalProfile provides the locally accepted information, whereas RemoteProfile can contain the preferred attributes for the receiving end. The certificate chains in the profile can override or supplement those in the KRSP configuration files.

When the key recovery enablement context creation is complete, the application calls CSSM_KR_GenerateRecoveryFields API function to have KRSP generate key recovery blocks. The KRSP takes the following steps to generate the KRB:

- *Input validation*: The application supplies the KRSP with a context handle pointing to the KR enablement context. The KRSP performs certificate verification on the certificate chains provided in the context before using them. Specifically, the application can provide profiles for enterprise and individual through the KR enablement context.
- *Generate SKR KRF*: In this step, KRSP calculates all the values that are needed to populate B1 and B2 blocks in the SKR KRB. The structures for B1 and B2 are filled in according to the contents of the master configuration file (jur-type.cfg) and the flags passed in as the argument of the API function. KRSP then proceeds to ASN.1 encode the blocks to generate the KRF.
- *Encode Open Group CKRB:* The next step is to create the CKRB. The result of the previous step – encoded B1 and B2— is put into the CKRB structure as the KRF and correct values for version, and type are filled in. The KRSP also needs to tie the KRB with the encapsulated session key, so a keyed hash of the KRF and the session key is calculated and added to CKRB, and the CKRB structure is AN.1 encoded. At this point, key recovery block generation is complete and the encoded CKRB is returned to the calling application.