Security Service API:
Cryptographic API Recommendation

NSA Cross Organization CAPI Team

June 12, 1995

EXECUTIVE SUMMARY

Until recently, the integration of cryptographic functionality into application software has required that developers tightly couple the application to the cryptographic module. This approach forces each new combination of application and cryptography to be treated as a distinct development, and does not provide the modularity and maintainability needed for commercial products[1]. A technique that can provide a much more flexible and powerful alternative is the use of a standardized Cryptographic Application Program Interface (CAPI).

The compelling case for a modular cryptographic interface has given rise to the development of numerous proposed CAPI standards. An NSA cross organizational team was formed to assess the ability of these proposed standards to meet anticipated needs. After an initial review, a CAPI strategy was developed and a detailed analysis of a subset of the proposals was performed. The strategy recommended by the team and the findings of the analysis are included in this report.

Rather than recommending the selection of a single CAPI, the team concluded that a combination of the three widely accepted proposals be adopted. These proposals include the GSS-API (Internet Engineering Task Force), the GCS-API (X/Open), and Cryptoki (RSA). Each of these CAPIs was designed to support significantly different levels of security awareness, with the GSS-API requiring little cryptographic awareness and Cryptoki requiring extensive knowledge of the underlying cryptography. The criteria used by the team to assess each of these CAPIs included algorithm independence, application independence, cryptomodule independence, degree of cryptographic awareness, modular design and auxiliary services, legacy cryptographic support, safe programming, and security perimeter design approach.

While the recommended three-CAPI suite addresses all types of applications, it is anticipated that the majority of applications will require minimal knowledge of the underlying cryptography. Therefore, the high-level GSS-API should be selected for use whenever practical.

1. INTRODUCTION

Until recently, the integration of cryptographic functionality into application software has required that developers tightly couple the application to the cryptographic module. This approach forces each new combination of application and cryptography to be treated as a distinct development, and does not provide the modularity and maintainability needed for commercial products[1]. A technique that can provide a much more flexible and powerful alternative is the use of a standardized Cryptographic Application Program Interface (CAPI).

As application developers become aware of the need for cryptographic protection, they are adding "hooks" to access the cryptographic functionality being developed by other programmers. Those "hooks" are known as the CAPI. As the sophistication of CAPIs increases, the benefit of a standard interface becomes apparent. Applications that utilize a standard CAPI can access multiple cryptographic implementations. This decreases implementation efforts. This also reduces licencing fees for algorithms when non-proprietary algorithms are available. There are numerous efforts currently under way to create CAPI standards. They range in scope from very generic cryptographic support like that found in GSS-API [2] to an interface more involved in the actual control of the cryptographic module (i.e., cryptographic token[1]) like that in Cryptoki [5]. A number of these CAPI efforts are receiving a great deal of support as applications are being written to use them. While we would ideally select a single CAPI standard for all applications, having multiple standards would fulfill the requirements for many different applications.

Increases in internal development costs, along with increased availability of commercial cryptography, have changed how our implementors integrate cryptography with applications. While the development of secure applications by industry solves the problem of our programmers modifying the application software, it replaces it with a new problem. Our programmers must now create software that maps the CAPIs used by the applications to the cryptographic modules. To accomplish the mapping from secure COTS applications to a standard subset of CAPIs two approaches are required. The first approach is to select and encourage using a subset of the CAPI standards efforts most suited to our needs, and to attempt to direct the other CAPI efforts to align with one of the standards in this subset. The second is to create a security infrastructure that supports this subset of CAPI standards.

In deciding which subset of CAPIs to use, many CAPIs were considered; some could not be included because of their company propriety nature and others because of their of lack of maturity. The remainder of this document describes the criteria used to select the subset of CAPIs and highlights the strengths and weaknesses of those selected.

---

1. Token is used within this recommendation to mean an entity that implements a cryptographic function. This definition is consistent with that presented in RSA's Cryptoki CAPI.

2. SCOPE

This document considers a set of CAPIs to be used with commercial, off-the-shelf (COTS) applications for use in system integration work. The CAPI isolates application developers from the variety of cryptographic libraries, hardware tokens, and software tokens, giving the user maximum flexibility in selecting cryptographic services for inclusion in an application. The CAPI also greatly simplifies the conversion of applications to use various cryptographic algorithms.

Applications operate in a variety of environments (i.e., hardware platform and operating system), some with low levels of trust, that are characterized as “untrusted environments,” and some with high levels of trust, that are characterized as “trusted environments.” Most commercial applications do not include adequate security assurance features, and are considered “untrusted applications.” Even if these applications are operated in a trusted environment, they are still “untrusted.” We believe software applications and environments will evolve to include more trust. Today’s CAPIs should not inhibit this evolution.

A concept of cryptographic awareness has been used by several CAPI developers to describe the level of security service that an application could be allowed to access. This concept can be extended to also include a description of the operating environment and the determination of the level of security service access allowed. The desired goal is for general-purpose applications (e.g., spreadsheets, document processors, E-mail, etc.) to be cryptographic unaware with a minimum number of high-level calls necessary. Ideally, these high-level calls would initiate secure contexts, protect data, and terminate security contexts. These calls would require no knowledge of specific cryptographic modules. At most, the application might indicate a desired “quality of protection.”

Other special-purpose applications, like Security Association Managers, may be cryptographic aware and allow an extensive suite of calls that permit more precise control of the cryptographic token. These calls would require extensive knowledge of the implementation of the cryptographic modules. In this case, the application may select a specific cryptographic algorithm or mode of operation.

In a few specific cases, the application must manipulate the cryptographic token directly to provide functions like inserting keys, selecting personalities and revoking capabilities. Since these functions provide access to the heart of the cryptographic token, they require the greatest amount of cryptographic knowledge. Care must be taken when using this interface since a mistake by the caller may compromise the security of the cryptographic token.

Exactly how the CAPI provides applications with access to cryptography will vary. The case where only one application requires access to a single cryptographic module will be handled differently than when multiple applications access multiple cryptographic modules simultaneously. Figure 1 illustrates the general CAPI architecture, where secure COTS applications that span the spectrum of cryptographic knowledge are present in the system. In order to support these applications the Cryptographic Services Manager must support all three CAPIs. Likewise, a broad range of cryptographic services are provided by hardware and software

cryptographic tokens. To provide these cryptographic services to the applications, again the three CAPIs must also be provided by the Cryptographic Services Manager. The Cryptographic Services Manager can then mediate access to the cryptographic tokens. Depending on the application environment, if only cryptographic-unaware applications were on the system, then the Cryptographic Services Manager would need only support the cryptographic-unaware CAPI at the client interface.

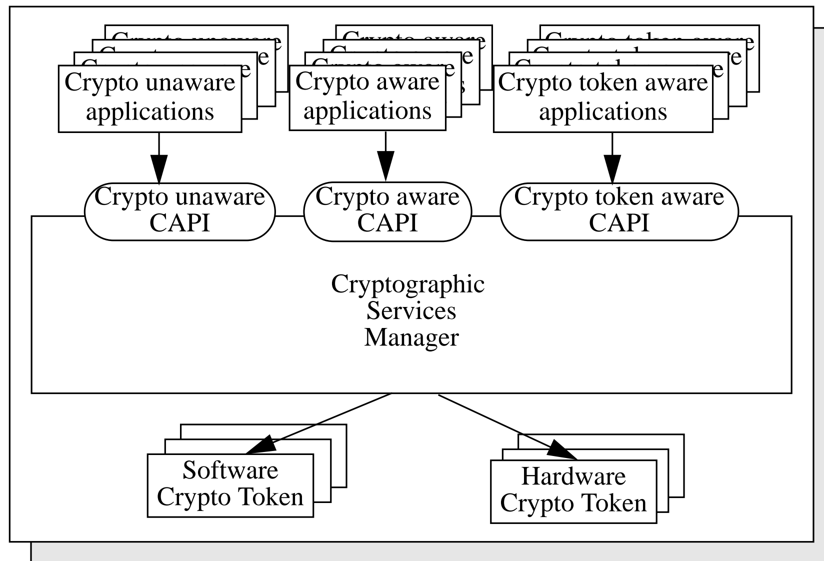

Figure 1: General CAPI Architecture

The background of the three CAPIs is discussed in Section 3. The criteria for choosing which CAPIs were to be evaluated is discussed in Section 4. The three chosen CAPIs are examined in Sections 5 through 7. These CAPIs were chosen because they fulfill our criteria and seem likely to survive in the standards process, because of widespread vendor and user interest.

Section 8 concludes with a three-suite CAPI recommendation. In general, less knowledge of cryptography is required for the higher-level CAPIs, while applications which are “cryptographic aware” may call lower-level CAPIs for finer granularity of control over cryptographic subsystems.

Sections 9 and 10 provide acknowledgments and references. The appendices include a list of acronyms, a glossary of terms and CAPI application scenarios.

3. BACKGROUND

The following section gives background information on each of the CAPIs reviewed. Within this section, and the section comparing a particular CAPI against our criteria, the terminology will mirror that used by the CAPI being discussed. This approach was chosen to allow the reader using the referenced CAPI to more easily apply our recommendations.

The Generic Security Services API (GSS-API) [2] and the extensions for independent data unit protection (IDUP-GSS-API) [3] support cryptographic-unaware applications. This CAPI is described in Section 5, and provides a high-level interface to authentication, integrity, non-repudiation and confidentiality services. The application merely indicates the required security services and (optionally) the quality of protection (QOP). GSS-API is session-oriented, and is used to protect session-style communications with other entities. IDUP-GSS-API does not assume real-time communications between sender and recipient, and protects each data unit (e.g., file or message) independently of all others. IDUP-GSS-API is therefore suitable for protecting data in a store-and-forward environment.

The Generic Crypto Services (GCS-API), described in Section 6, supports cryptographic-aware applications. This layer allows a caller to establish a cryptographic context which uses one or more cryptographic modules to protect data. The GCS-API supports protection of data on a per-buffer basis using one or more algorithms. GCS-API does not require (nor preclude) knowledge of specific algorithms. The source for this standard is X/Open [4]. The GCS-API is slated to be advanced through the POSIX standards process as well. Important inputs to this document include the X/Open Distributed Security Framework, the draft NIST Cryptographic Service Calls FIPS, and contributions from IBM and ANSI X9F1.

A final type of CAPI is an abstract token interface, which defines the arguments and results of various algorithms. It also specifies certain objects and data structures which the token makes available to the application. Ideally, implementations can be constructed so that only this layer needs to be replaced when using different cryptographic modules or tokens. This layer is specified in Cryptoki [5], from RSA Laboratories, and is discussed further in Section 7. Note this layer is the only one which interfaces to cryptographic tokens, and is thus the logical place for functions that allow user interaction (e.g. logon or PIN entry) and administrative control of the token. The lowest level CAPI (i.e., Cryptoki) is appropriate for use by developers of cryptographic devices and libraries.

Additional functionality that is supported by the CAPI and is not visible to the application includes access control (to specific functions and/or algorithms), security event generation, and auditing. Detailed analysis of this functionality is beyond of the scope of this document.

Industry and national standards bodies are working on CAPI standardization as well. In particular, ANSI X9 (Financial Services) is proposing a layered architecture similar to that described here, and NIST is planning to align its Cryptographic Service Calls FIPS with the X/Open GCS-API standard. Coordination with standards bodies is essential to encouraging these CAPIs be widely accepted by the commercial community. Additionally, as these CAPIs become standards, they will also be moved into the IEEE POSIX standards process, to simplify procurement.

4. CRITERIA

The following criteria were used to analyze the CAPIs receiving the largest amount of user and vendor support.

•**Algorithm Independence:** *Algorithm Independence* is defined as the property that the CAPI does not specify use of a particular algorithm to provide cryptographic service. The CAPI must provide access to a large number of choices of current and future cryptographic algorithms. This property gives an application access to any cryptographic algorithm supported by the underlying cryptomodule, enhancing interoperability. An example of algorithm independence is a reference to "*encryption algorithm*" rather than "*DES algorithm*."

•**Application Independence:** *Application Independence* is defined as the property that the CAPI is equally suitable for designing any application. The CAPI must provide cryptographic service to a wide variety of applications being written today, and to many new ones in the future. A CAPI that has the property of application independence will enable the programmer to write a wide variety of applications. This will give the CAPI widespread use and longevity. A CAPI that is application independent will *not*, for example, be specifically designed for programming E-mail applications. As another example, a well-designed CAPI should be able to support coding of a store-and-forward application, like E-mail, as well as a connection-oriented application, like file transfer.

•**Cryptomodule Independence:** *Cryptomodule Independence* is the property that the CAPI, in providing its cryptographic service, can use any current or future cryptomodule as easily as any other. The application should not need to know the specifics of the underlying cryptographic implementation. For example, the application need not know whether or not the cryptography is provided in hardware or software. In addition, a single application may use multiple cryptomodule implementations.

•**Degree of Cryptographic Awareness:** *Cryptographic Awareness* refers to the amount of cryptographic knowledge the programmer has. For the majority of applications, a minimal degree of cryptographic knowledge is needed by the developer. For some applications, for example those in key management, the programmer is required to have a higher degree of cryptographic knowledge. CAPI specifications can fall anywhere in the spectrum from requiring little knowledge to requiring a great degree of expertise in cryptography. A complete CAPI suite should support both cryptographic-aware and cryptographic-unaware applications.

•**Modular Design and Auxiliary Services:** *Modular Design* is the division of the CAPI into units which work together to provide the complete security service, each of which has a focused purpose. *Auxiliary Services* are support services used by the goal cryptographic service. These can include, for example: key management, authentication, certificate management, query capability, and (user-to-CAPI) session set-up/tear-down capability. The CAPI architecture must be functionally complete. In simpler architectures, there may be only two modules: cryptographic service and key management. It is expected that auxiliary services that are not fully developed and modularized today will become separate modules. In many current standards, the lack of APIs providing access to these support security services has caused the inclusion of that functionality within another API - usually the cryptographic calls themselves. While these services meet our needs, continuing development on them will most likely result in numerous other auxiliary service APIs.

As a separate auxiliary service, the CAPI must also include and/or support a function to authenticate its underlying cryptosubsystem. It is envisioned, for example, that cryptographic token authentication functionality could filter up through the CAPI suite from a lower-level CAPI (Cryptoki).

•**Legacy Support:** *Legacy support* is defined as providing the extensibility, so that current cryptography can be supported. It is vital that all CAPIs we recommend support and do not preclude our current cryptographic products.

•**Safe Programming:** *Safe programming* is the term used for describing steps taken to guard against inadvertent security mishaps by the programmer. Three aspects that contribute to safe programming include consistent naming conventions, information hiding, and ease of use. By having consistent naming conventions, the possibility of the programmer misunderstanding the purpose and use of procedures and parameters decreases. Information hiding is provided by the use of opaque handles and pointers, to prevent the inadvertent exposure of sensitive information. Ease of use results in the CAPI doing many of the detailed tasks of parameterizing and sequencing of cryptographic operations, and is likely to decrease the probability of programmer error. This is especially important for CAPI specifications aimed at the cryptographic-unaware programmer.

•**Security Perimeter:** A *security perimeter* is defined as the boundary that prevents sensitive security-related information from leaking out of the trusted computing base into untrusted applications. In trusted systems, architectures with a security perimeter will contain the cryptography within the perimeter while the application is outside. The CAPI provides the access between the untrusted components and the trusted components. The CAPI must not be used as a portal to violate the security perimeter; it must restrict access to sensitive cryptographic data (e.g., unprotected keys). It must not propagate such information beyond the CAPI interface.

5. GSS-API

5.1 Introduction

The Common Authentication Technology (CAT) working group of the Internet Engineering Task Force (IETF) is creating a set of APIs and mechanisms to provide security services to application programmers. The initial offering from this group was a suite of RFCs (request for comments), numbers 1508-1511, which were published in late 1993. These documents outlined the initial generic security API, the C bindings for the API, the Kerberos authentication protocol and a CAT overview. Since that time, GSS-API received widespread attention as a number of development activities attempted to utilize the interface to access security services in the system. As is so often the case, deficiencies in the initial RFC have been identified, and work is currently under way to create a second draft of the GSS-API along with a companion draft to support Independent Data Unit Protection (IDUP). The intent is for the cryptographic-unaware application to use the GSS-API and IDUP-GSS-API.

One useful concept provided by the GSS-API is that of security contexts. A security context is the relationship established between peers, using credentials established locally in conjunction with each peer or received by peers via delegations[2].

Since the work within the IETF transpired without our involvement, there was little effort to ensure that our requirements can be met in the API. This section presents the features of GSS-API that met our criteria and those features that do not. Based upon these points, a recommendation on our involvement, use, and support of the API is made.

5.2 Strengths

The GSS-API specification meets a number of the criteria listed above.

Algorithm Independence

GSS-API is cryptographic algorithm independent. This allows applications to be completely unaware of the algorithms providing the protection. In cases where an application or user is knowledgable of the cryptographic capabilities located on the system, the interface supports the ability to specify a quality of protection or a desired set of mechanisms when requesting a security context. During the creation of that context, the Cryptographic Service Manager below the CAPI will determine whether to create the requested security context, deny the context, or grant the context with other mechanisms being used. The security policy decision process is beyond the scope of this document.

Cryptomodule Independence

GSS-API is a high-level CAPI and hides the fact that the cryptographic mechanism is implemented in hardware or software. The only reference the application has to the cryptographic mechanism is an opaque handle, which represents the context created by the cryptographic subsystem. Each security context may utilize more than one cryptographic module to protect the information. Also, since security contexts are represented by pointers and not the identity of the application, each application may have numerous concurrent security contexts each potentially using different mechanisms.

Degree of Cryptographic Awareness

The GSS-API has a small set of routines. The application using the GSS-API can be cryptographic unaware. Although a few of the GSS-API calls have a number of parameters, most can be set to specify default actions or services, thus reducing the amount of information the application must provide for the calls.

Modular Design and Auxiliary Services

The GSS-API is split into four groups of calls where each group has a small set of well-defined calls. The first provides credential management, the second provides security context management, the third provides the per message calls, and the last group is the miscellaneous support calls. In each case, the ordering of the parameters and the naming of functions and parameters follow the same rules.

Safe Programming

By using opaque pointers to identify sensitive information like credentials, contexts, and other complex data structures, applications can reference the information without having access to the sensitive information itself. Call sequence is ensured by the fact that the GSS-API requires credential pointers to do context operations and context pointers to do the per-message calls.

Security Perimeter

In systems which separate the client applications from the cryptographic service provider with a security perimeter, GSS-API will reduce the amount of sensitive security information which is exposed to the application. Most data structures internal to the cryptographic module are identified by the application as opaque pointers. In this way, the application is prevented from having access to the data itself. Also, since GSS-API is at a fairly high level, keys and other sensitive cryptographic data are never exported beyond the CAPI.

5.3 Weaknesses

The following GSS-API weaknesses were found.

Application Independence

The GSS-API works in a session-oriented paradigm, and RFC 1508 does not support store and forward applications or applications with multiple receivers. The initial specification could be forced to work in these scenarios by modifying the underlying mechanisms and changing the target name to represent an alias or group, but this approach is not very clean. Draft 2 of the GSS-API and the IDUP draft address these issues but are not as far along on the standards track. (This is discussed further in Section 8.)

Modular Design and Auxiliary Services

Authentication of the cryptomodule by the application is not addressed by the GSS-API specification. It is vital to have high confidence in the identity of its cryptosubsystem; therefore, the CAPI must provide an interface to allow an application to authenticate its cryptosubsystem.

The security architecture used in the IETF CAT working group provides access to the basic cryptographic services: authentication, data integrity, and confidentiality; it did not, however, include a number of the security services seen in other CAPIs. GSS-API does not provide an interface for services like user authenticated logon, key management, access control, and the storage and access to security database information. Applications which need to access these services must utilize another API designed for the specific service.

Legacy Support

Since the GSS-API interface does not include auxiliary security services like user authentication, it is more difficult to ensure legacy support in a system which uses GSS-API than one which uses a CAPI which provides that service. To ensure compatibility, other security services must be considered in conjunction with GSS-API. Even within the scope of cryptography, the GSS-API interface does not export some of our legacy capabilities to the application. For example, the application cannot authenticate the cryptographic module below the interface or access the key management capabilities of our current cryptography. It is envisioned that access to services like key management and the cryptomodule authentication would be performed through a Key Management API or a lower-level CAPI like GCS-API or Cryptoki.

5.4 General

The initial GSS-API presented in RFC 1508 is not complete and will not support the wide range of generic applications which are being considered when analyzing the CAPI candidates. Fortunately, the CAT working group has taken steps to expand the GSS-API interface to support applications that do not fit into the session-oriented paradigm. If the current RFC 1508 is superseded by the new draft GSS-API [6] and the IDUP extensions, then the IETF will produce a high-level API for security services which we should adopt for use in the majority of their applications which require cryptographic services but are not trusted with cryptographic knowledge.

6. X/OPEN GCS-API

6.1 Introduction

The Security Working Group (SWG) of the X/Open consortium is creating a set of APIs and mechanisms to provide security services to applications; however, the GCS-API is designed for cryptographic-aware applications. These applications understand at least the basics of the operation of cryptographic algorithms and key management. As stated earlier, important inputs to this document include the X/Open Distributed Security Framework, the draft NIST Cryptographic Service Calls FIPS, and contributions from IBM and ANSI X9F1. GCS-API is still going through the standards development phase and is the least mature of the three CAPIs recommended in this document.

6.2 Strengths

The GCS-API meets a number of the listed criteria.

Algorithm Independence

Using GCS-API, applications may indicate specific algorithms or the desired quality of protection (QOP) when creating a cryptographic context. An IBM contribution to this effort (not yet adopted) allows specification of a default QOP as well. The CAPI calls themselves do not assume any particular algorithms are in use.

Application Independence

GCS-API is aimed at cryptographic-aware applications, such as the mechanisms used to implement security specific application paradigms (session-oriented communications, store-and-forward, etc.). Thus, its application independence is due to its relatively low level.

Cryptomodule Independence

GCS-API may be implemented on top of any cryptographic technology, cryptomodule, or algorithm. As discussed in Section 2, it is desirable for GCS-API to make use of a lower layer cryptographic token interface, as discussed in Section 7.

Safe Programming

The GCS-API interface uses cryptographic contexts to access pointers to the appropriate key and algorithm structures (information hiding). Requiring context pointers to do cryptographic calls provides some assurance that calls will be invoked in the proper order. For those calls which must process multiple buffers of data while maintaining cryptographic state between calls, a chaining flag is available (e.g., intermediate chaining values, each call accepts a chaining flag: "first," "middle," "last," "only"...). Values and semantics for the flag vary from call to call. This means that other parameters to the call have different semantics (and indeed may not even be used in some cases) depending on the value of this flag. One obvious example is the "Output-Value" parameter of the Generate-Hash function. This is only used (valid) if the chaining flag is "last" or "only."

Modular Design and Auxiliary Services

The GCS-API provides access to cryptographic algorithms to support the usual set of services: authentication, data integrity and confidentiality. It also contains a robust set of protected and unprotected key management calls.

Our current cryptography can be used as a cryptomodule under GCS-API, only if the GCS-API conformance policy is lenient to the extent that not all calls must be implemented (again, see Section 8). Specifically, no unprotected keys can be output from our cryptography and no initialization vectors can be specified. Otherwise, mapping GCS-API calls to our cryptography calls in both directions can be accomplished.

6.3 Weaknesses

The following GCS-API weaknesses were found.

Degree of Cryptographic Awareness

GCS-API is not recommended for general applications. The use of GCS-API should be the exception, only to be used by cryptographic-aware applications.

Modular Design and Auxiliary Services

Authentication of the cryptomodule by the application is not addressed by the X/Open specification. It is vital to have high confidence in the identity of its cryptosubsystem; therefore, the CAPI must provide an interface to allow an application to authenticate its cryptosubsystem.

Safe Programming

In general, there is a lack of consistency in naming, parameter conventions, etc. This is probably due to the relatively preliminary state of the draft standard.

Security Perimeter

GCS-API will export unprotected keys to applications with access to them (the draft refers to these as “cryptographic-enforcing” callers.) Our goal is that no unprotected key ever be passed outside the security perimeter, and the CAPI will generally be a portal through the security perimeter. This is not a condemnation of the X/Open design but instead limits the applications which should use the interface. If the application consuming the unprotected key is within the trust boundary and the system access control can be trusted, then the GCS-API design may be equivalent to one in which all unprotected key handling appears to be contained within the security perimeter.

6.4 General

A CAPI at this level, cryptographic aware, is designed to be used by developers of the GSS-API and IDUP-GSS-API and for applications which use cryptography in ways not accommodated by these higher layer CAPIs. While GCS-API is still being developed, the foundation is strong enough to conclude that, if changes are made, the specification has good potential to suit our needs.

7. RSA CRYPTOKI

7.1 Introduction

Cryptoki (“Crypto-Key”) is a new member of RSA's Public Key Cryptography Standards (PKCS) family, that provides guidance to the commercial cryptography community, and is distributed without charge by RSA. Cryptoki provides a standard low-level CAPI, primarily for access to personal cryptographic tokens. RSA realized that their existing commercial libraries were not flexible or general enough to support the needs of applications working with such devices, and therefore developed Cryptoki. Additional goals in the Cryptoki design include portability, generality, support for resource sharing, and algorithm independence.

7.2 Strengths

The RSA Cryptoki specification meets a number of the criteria.

Algorithm Independence

Cryptoki is algorithm independent. While the draft standard does specify particular algorithms, key types, and algorithm specific object attributes for RSA and public algorithms, the use of object identifiers to specify algorithms in the CAPI allows any organization the capability to register algorithms with any registration authority and add them to Cryptoki. Our existing algorithm suite will utilize a unique object identifier to specify our algorithm suite. This will allow applications to access attributes which are unique to our algorithms. Applications need not specify which algorithms are to be used for various cryptographic functions. If necessary, the application can query the cryptomodule to discover which algorithms are supported.

Application Independence

Application independence is provided since Cryptoki is at such a low level and because it defines a highly abstract and general view of the cryptography provider. A cryptography provider, as a token, stores a collection of objects and can manipulate them to perform cryptographic operations. Cryptoki is thus suitable for cryptographic-aware applications.

Cryptomodule Independence

Cryptoki meets the design goal of module independence by defining an abstract token model, that is implemented by the combination of the physical token and the Cryptoki library. Cryptoki was written with personal cryptographic tokens (e.g., smart cards, PCMCIA cards) in mind. Extension mechanisms are defined to allow the addition of new capabilities (i.e., functions, object types, etc.) by individual implementors. This provides a way, for example, to access our specific functions (although other tokens might not support these extensions).

Modular Design and Auxiliary Services

Cryptoki supports basic cryptographic operations such as hash, signature and encryption, that in turn provide the required security services of integrity, authentication, and confidentiality. Signatures also provide some support for nonrepudiation services, although

clearly a more complete architecture is needed for full support. Other operations include ASN.1 encoding and decoding of keys and algorithm identifiers. Many of the basic cryptographic operations (e.g., encryption and key agreement) can be used as primitives in support of key management operations. One feature that provides some level of access control is the notion of public and private objects. Public objects are accessible after opening the token, private objects are accessible only after logging into (or initializing) the token. Perhaps, this is adequate for the Cryptoki-envisioned single-user token environment. In a multi-user environment, additional controls would be required to protect a single token from multiple applications making simultaneous accesses.

One of the stated design goals for Cryptoki is resource sharing. Cryptoki does utilize the notions of contexts and pointers, which are suitable for multitasking, multithreaded systems. The design of the Cryptoki CAPI is operating system (OS) independent (beyond the need for a C compiler). Lack of any OS dependencies means the implementor would have to provide access management in the case where many applications share a single token. To achieve good security, an application using Cryptoki should require trusted OS facilities for getting the user's PIN, which is a capability supported by Cryptoki to provide user authentication.

Legacy Support

A legacy compliant implementation of Cryptoki could be built.[1] Nothing in the draft standard precludes exposing any of our cryptographic functionality as Cryptoki functions and objects.

Safe Programming

Cryptoki is designed in an object-oriented fashion. The object-oriented design provides a large degree of data hiding, which is one of the major mechanisms to support safe programming. The final version will consist of a language independent specification and C bindings, but equivalent C++ bindings could also be defined. In some respects, C++ bindings would be simpler than the C bindings. In addition, the Cryptoki draft uses consistent naming rules for routines, constants, and data types. However, these rules should be explicitly stated in the standard.

Security Perimeter

Cryptoki does explicitly hide sensitive information (e.g. private keys). Practically everything that Cryptoki manipulates is an opaque object with a set of attributes. One such attribute is the SECRET attribute, that when TRUE, indicates the object may not be exported outside of the token.

---

1. Cryptoki and our current libraries are not compatible, and software must be written to translate between the two interfaces.

7.3 Weaknesses

The following RSA Cryptoki weaknesses were found.

Degree of Cryptographic Awareness

Cryptoki is layered directly on top of actual cryptographic tokens or algorithm libraries. As such, it requires a substantial degree of cryptographic awareness. The generality of Cryptoki makes it well suited as a foundation for higher-level libraries such as the GCS-API.

Modular Design and Auxiliary Services

Cryptoki provides few additional services beyond cryptography, key management, and user authentication. In addition, Cryptoki provides extensive facilities for querying the capabilities of the cryptomodule, but very little for querying its state. (The state of a token might be supported as an attribute unique to that kind of token.)

Authentication of the cryptomodule by the application is not addressed by the Cryptoki specification. It is vital to have high confidence in the identity of its cryptosubsystem; therefore, the CAPI must provide an interface to allow an application to authenticate its cryptosubsystem. Cryptoki provides extension mechanisms, so such a function could be added as an extension; ideally, however, this should be specified in the actual standard.

Safe Programming

Cryptoki was not designed for novice C programmers. This CAPI uses the C language in a sophisticated, object-oriented way, and programmers using this CAPI will need substantial C expertise and familiarity with object-oriented programming concepts. This usage of the C language maps very naturally to a C++ language binding.

7.4 General

Cryptoki is a CAPI for cryptographic-aware applications. Cryptoki presents an abstract token interface with most of the functionality required at this degree of cryptographic awareness. Additional functionality can be added using the normal Cryptoki extension mechanisms. By using Cryptoki, only a minimal amount of code needs to be changed when using different cryptographic tokens.

8. CONCLUSION

This document has examined three CAPIs, each of which support a variety of functions at varying levels of cryptographic awareness. Table 1 summarizes the criteria from Section 4 and indicates how the three recommended CAPIs fared with respect to the criteria.

Criteria	GSS/IDUP	GCS	Cryptoki
Algorithm Independence	yes	yes	yes
Application Independence	yes[1]	yes[2]	yes[2]
Cryptomodule Independence	yes	yes	yes
Degree of Cryptographic Awareness	yes	no[3]	no[3]
Modular Design and Auxiliary Services	----------	----------	----------
Key Management	implicit[4]	yes	yes
User Authentication	no	no	yes
Certificate Management	some	no	no
Query Capability	no	no	yes
Set-up/Tear-down Capability	yes	yes	yes
Cryptomodule Authentication	no	no	no
Legacy Support	yes	yes	yes
Safe Programming [5]	4	2	2
Security Perimeter	yes	yes[6]	yes

Table 1: Summary of CAPI Capabilities

[1] GSS and IDUP each handle different paradigms.
[2] GCS-API and Cryptoki are sufficiently low-level to be application independent.
[3] GCS-API and Cryptoki are intended for the cryptographic-aware programmer.
[4] Key management is provided by the underlying GSS mechanisms.
[5] Safe programming is weighted from 1 through 5, with 5 being the most safe.
[6] Keys are protected physically (using hardware) or logically (encrypted under a facility master key). Similarly for intermediate function results (if not kept beneath the API), and "exported" contexts.

All three CAPIs will support our needs in the areas of legacy support and cryptomodule independence. The main difference between the CAPIs is in the areas of the amount of cryptographic knowledge required by both the application programmer and user, and the amount of potentially sensitive information that can be recovered through the CAPI. GSS-API provides the safest interface, but the most limited capability to manipulate the cryptography. Cryptoki and GCS-API provide applications with more capabilities to manipulate the cryptography that increases the ability of the application to misuse the interface. Since the majority of applications will be cryptographic unaware, the bottom-line recommendation is for applications to use GSS-API. Only if an applica-

tion absolutely needs to be cryptographic aware by performing security support or cryptographic token functions, should GCS-API or Cryptoki be considered for use by the application. Table 2 lists envisioned applications correlating to the recommended CAPI. Note, CAPIs are not strictly for applications, but are available for communications protocols as well.

Application	Recommended CAPI
Word Processor	GSS-API
Mail Application	IDUP-GSS-API
SMTP; X.400	GSS-API
Directory Service; X.500	GSS-API
SNMP; SNMPv2	GSS-API
IPSP; NLSP; TLSP; GULS;MSP	GSS-API
HTTP	GSS-API
Key Management Application	GCS-API
Key Management Protocol	GCS-API
Authentication Application	GCS-API
Security Association Protocol	GCS-API
Certification Manager	GCS-API or Cryptoki
Certificate Manager	Cryptoki
Cryptographic Token Application	Cryptoki

Table 2: Recommended CAPI per Application

8.1 Recommendation

To restate, the bottom-line recommendation is for applications to use GSS-API. With respect to GCS-API and Cryptoki, these should be used only by cryptographic-aware applications. Besides being directly called by cryptographic-aware applications, the two lower level CAPIs, GCS-API and Cryptoki, could also be used to build the underlying GSS-API mechanisms. This is the proposed three-tiered CAPI suite. GCS-API builds the security support functions for GSS-API. Likewise, Cryptoki builds the token functions for GCS-API. While this presents the most complete set of functions, this document does not mandate having to incorporate all three CAPIs. The use of the three-tiered CAPI suite depends on the security policy and the capabilities of the intended environment.

8.2 Strategy

All CAPIs are still in the development stage, that allows us to provide guidance and requirements to the appropriate standard committee or author. We are already performing this task. Ongoing efforts are being initiated for implementations of these CAPIs to be

acquired or generated in order to validate this recommendation. This includes building an environment with just GSS-API (the most common scenario) as well as the three-tiered CAPI suite (the complete environment-specific scenario).

8.3 Transition

This document strongly recommends that all projects attempt to use GSS-API for all their application security and cryptographic calls. If the project is developing a security support application or the application needs more cryptographic control than what is accessible through GSS-API, then GCS-API should be used. For development of an application for control of cryptographic tokens, Cryptoki should be used. All measures should be made to use one or a combination of these CAPIs. Development of "yet-another-CAPI" should be avoided at all costs.

To simplify the inclusion of future cryptography with CAPI-compliant applications, projects should standardize on a low-level interface to the cryptography. This document does not recommend what that interface should be, but researchers are investigating the use of Cryptoki for that interface and are currently developing a Cryptoki interface for our cryptography to replace the interface currently used. A goal of the ongoing CAPI project is to create a stronger recommendation in this area in the near future.

8.4 Risks

As with all recommendations, this one does contain risks. The major risk is that a computer systems vendor with a large market share creates a new CAPI and it becomes the de-facto standard. If this happens, we have a number of options depending on the relationship between the new CAPI and the existing CAPI suite supported by us. In the unlikely case that the new CAPI is very similar to one of the existing CAPIs in our suite, the new CAPI could replace that CAPI in the recommendation. A more likely situation is that the CAPI would not map nicely to one of the existing CAPIs. In this case, the most viable solution is to add the new CAPI to the recommendation. However, every new CAPI added to the recommendation increases the amount of support software that must be written. With this in mind, it is important that the number of CAPIs we support be as few as possible and that vendors be encouraged to adopt an existing CAPI standard rather than create their own.

The second risk is that an application that should be cryptographic unaware uses a lower-level CAPI directly and incorrectly. Since much of the software we will be using in the future will be COTS, we have little say over what an application developer does in their software. A number of approaches can be taken to limit the damage from this risk, ranging from general CAPI compliance testing to using a cryptographic infrastructure below the CAPI to reduce the amount of sensitive information that can be lost as a result of improper use of the CAPI.

The third risk is that the GCS-API is still maturing and that during the standards process GCS-API may be modified in ways that do not meet out requirements. However, we are already actively involved with the X/Open SWG to minimize this risk and to ensure that GCS-API will be an acceptable CAPI.

9. ACKNOWLEDGMENTS

This document is the result of a collaborative team effort. Acknowledgment is due to the many people who participated. Special acknowledgments are due to Shu-jen Chang, who authored the draft NIST FIPS, Cryptographic Service Calls, and Rich Ankney of Fischer International, who served as a consultant for the document.

10. REFERENCES

[1] Emmett Paige, Jr., “Selection of Migration Systems,” Assistant Secretary of Defense Memorandum, November, 1993.

[2] Linn, J., “Generic Security Service Application Program Interface,” RFC 1508, November, 1993.

[3] Adams, C., “Independent Data Unit Protection Generic Security Service Application Program Interface (IDUP-GSS-API),” Internet draft, March, 1995.

[4] X/Open, “X/Open Preliminary Specification: Generic Cryptographic Service API,” draft 3, March, 1995.

[5] Kaliski, B., “Cryptoki: A Cryptographic Token Interface, Version 1.0, draft 4,” April, 1995.

[6] Linn, J., “Generic Security Service Application Program Interface, Version 2,” Internet draft, March, 1995.

[7] National Security Telecommunications and Information Systems Security Committee, “National Information Systems Security Glossary,” NSTISSI No. 4009, 5 June 1992

[8] National Computer Security Center, “An Introduction to Certification and Accreditation,” NCSC-TG-029, January 1994

[9] National Computer Security Center, “A Guide to Understanding Security Modelling in Trusted Systems, NCSC-TG-010,” October 1992

APPENDIX A: ACRONYMS

ANSI	American National Standards Institute
ASN.1	Abstract Syntax Notation version One
API	Application Program Interface
CAPI	Cryptographic Application Program Interface
CAT	Common Authentication Technology
COTS	Commercial Off The Shelf
DES	Digital Encryption Standard
FIPS	Federal Information Processing Standard
GCS-API	Generic Crypto Service Application Program Interface
GSS-API	Generic Security Service Application Program Interface
GULS	Generic Upper Layers Security (protocol)
HTTP	Hyper Text Transfer Protocol
IDUP-GSS-API	Independent Data Unit Protection GSS-API
IEEE	Institute of Electrical and Electronic Engineers
IPSP	Internet Protocol (IP) Security Protocol
MSP	Message Security Protocol
NIST	National Institute of Standards and Technology
NLSP	Network Layer Security Protocol
NSA	National Security Agency
OS	Operating System
PASC	Portable Applications Standard Committee
PKCS	Public Key Cryptography Standard
PIN	Personal Identification Number
POSIX	Portable Operating System Interface
QOP	Quality of Protection
RFC	Request For Comments
SNMP	Simple Network Management Protocol
TCB	Trusted Computing Base
TLSP	Transport Layer Security Protocol
uTCB	unTrusted Computing Base

APPENDIX B: GLOSSARY

access control: Process of limiting access to the resources of an automated information system (AIS) only to authorized users, programs, processes, or other systems. [7].

access control policy: The set of rules that define the conditions under which an access may take place [4].

API - Application Programming Interface: The interface between the application software and the application platform, across which all services are provided. The application programming interface is primarily in support of application portability, but system and application interoperability are also supported by a communication API [4].

asymmetric keys: Different keys are used to encrypt and decrypt data (i.e., public and private keys) [4].

authentication: Security measure designed to establish the validity of a transmission, message, or originator, or a means of verifying an individual's eligibility to receive specific categories of information [7].

availability: The property of being accessible and usable upon demand by an authorized entity [8].

cipher: Cryptographic system in which units of plain text are substituted according to a predetermined key [7].

ciphertext: Enciphered information [7].

clear text: Intelligible data that is not encrypted [4].

confidentiality: Assurance that information is not disclosed to unauthorized entities or processes [7].

credentials: Information, passed from one entity to another, that is used to establish the sending entity's access rights [7].

context: See security context.

cryptographic algorithm: Well-defined procedure or sequence of rules or steps used to produce cipher text from plain text and vice versa [7].

cryptographic context: See security context.

cryptography: Principles, means, and methods for rendering plain information unintelligible and for restoring encrypted information to intelligible form [7].

data integrity: Condition that exists when data is unchanged from its source and has not been accidentally or maliciously modified, altered, or destroyed [7].

data origin authentication: Corroboration that the source of data is as claimed [7].

data structure: The format in which the cryptographic material and/or attributes are stored.

decipher: Convert enciphered text to the equivalent plain text by means of a cipher system [7].

decode: Convert encoded text to its equivalent plain text by means of a code [7].

decrypt: Generic term encompassing decode and decipher [7].

encipher: Convert plain text to equivalent cipher text by means of a cipher [7].

encode: Convert plain text to equivalent cipher text by means of a code [7].

encrypt: Generic term encompassing encipher and encode [7].

key: Information (usually a sequence of random or pseudorandom binary digits) used initially to set up and periodically change the operations performed in crypto-equipment for the purpose of encrypting or decrypting electronic signals, for determining electronic counter-countermeasures patterns, or for producing other key [7].

key management: Process by which key is generated, stored, protected, transferred, loaded, used, and destroyed [7].

nonrepudiation: Method by which the sender of data is provided with proof of delivery and the recipient is assured of the sender's identity, so that neither can later deny having processed the data [7].

peer-entity authentication: The corroboration that a peer entity in an association is the one claimed [4].

private key: A key used in an asymmetric cryptographic algorithm. Possession of this key is restricted, usually to only one entity [4].

public key: The key used in an asymmetric cryptographic algorithm, that is publicly available [4].

secret key: The shared key between two entities that is used in a symmetric cryptographic algorithm [4].

security aware: The caller of an API that is aware of the security functionality and parameters which may be required and/or provided by an API [4].

security context: The relationship established between peers, using credentials established locally in conjunction with each peer or received by peers via delegations [1].

security policy: The set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information [9].

symmetric keys: see secret key.

token: The underlying hardware or software cryptographic device [5].

trusted computing base: Totality of protection mechanisms within a computer system, including hardware, firmware, and software, the combination of which is responsible for enforcing a security policy [7].

APPENDIX C: SCENARIOS

This appendix is provided to illustrate how the recommended CAPIs are envisioned to be a part of system implementations. For these scenarios either hardware or software cryptographic tokens could be used.

C.1 General

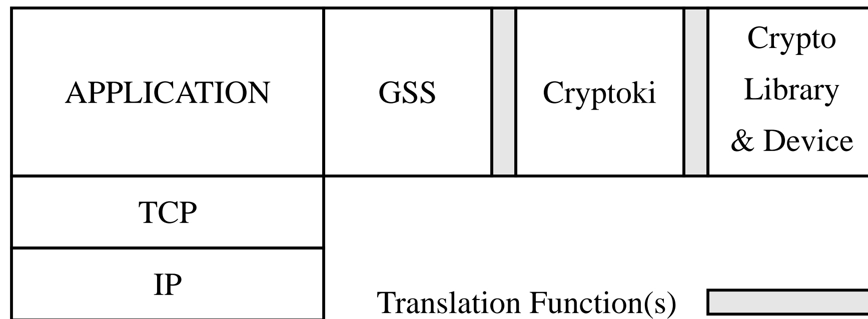

Figure 2: General CAPI Scenario

This scenario (**Figure 2**) illustrates how any cryptographic-unaware application on either a Trusted Computing Base (TCB) or unTrusted Computing Base (uTCB) requiring security services can make GSS-API calls to receive those services. Likewise any cryptographic token that builds to Cryptoki can be accessed by any application via a GSS-API to Cryptoki layering. In order to make this a reality, a one-time effort to develop the thin layers of translation code will have to occur.

C.2 Complete

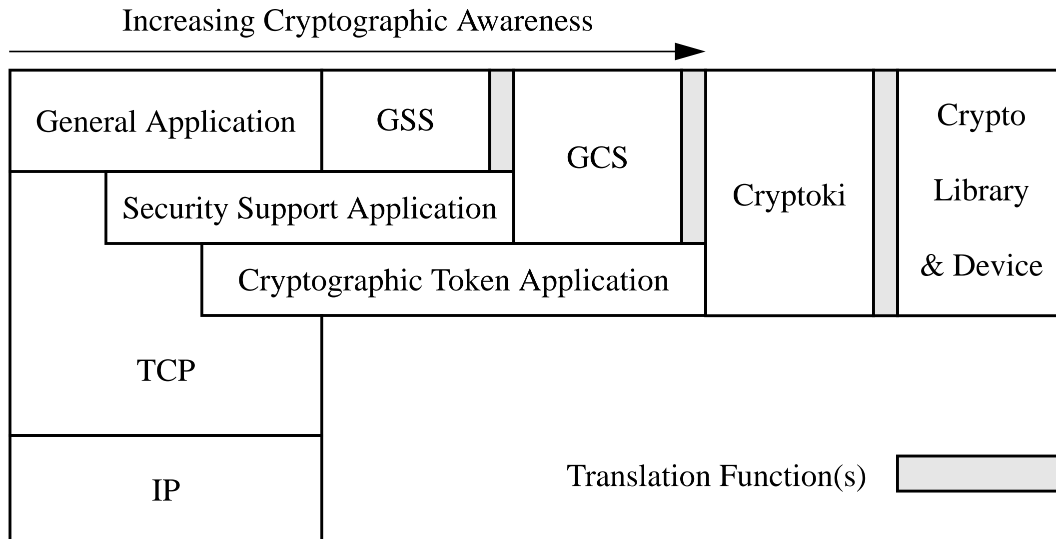

Figure 3: Complete CAPI Scenario

This scenario (**Figure 3**) provides the same capabilities as the general scenario with the addition of the GCS-API. The GCS-API provides the capability for security support applications and to make cryptographic-aware calls on a TCB. Security management is envisioned to be introduced at this layer. Again in order to make this a reality, a one-time effort to develop the thin layers of translation code will have to occur.

C.3 Future Risk Scenario

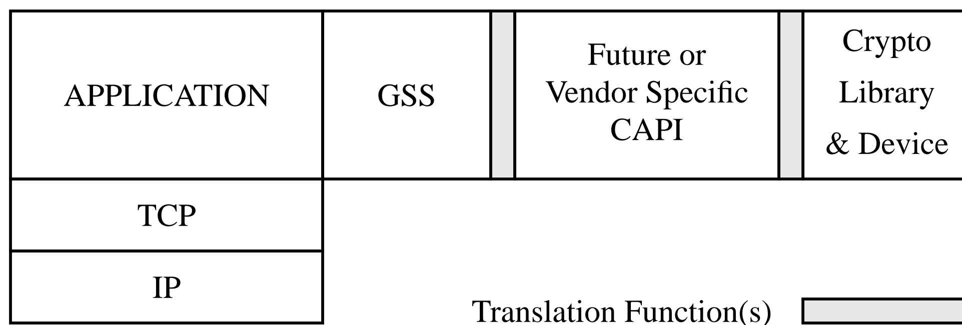

Figure 4: Future CAPI Risk Scenario

This scenario (**Figure 4**) still has the cryptographic-unaware application on either a TCB or uTCB making GSS-API calls, but allows for future or vendor specific CAPIs to be incorporated underneath. Even though a new CAPI was added, the application remains unaffected.

C.4 DMS: A Government Specific Example

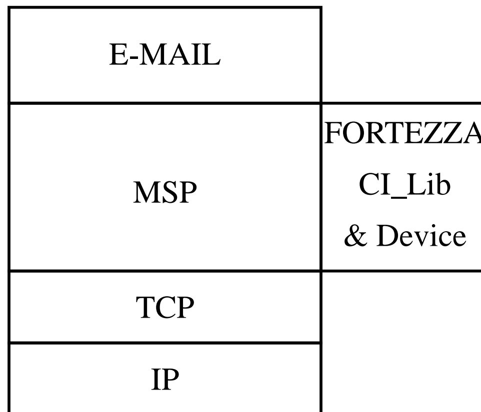

Figure 5: DMS Without CAPIs

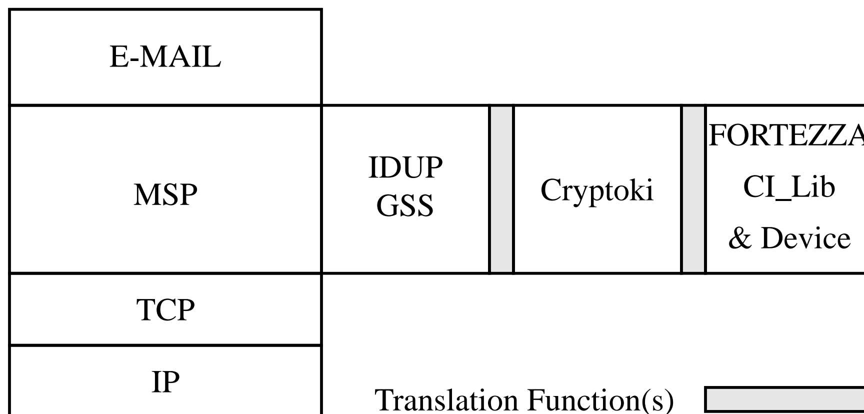

Figure 6: DMS Using CAPIs

The first scenario (**Figure 5**) shows how DMS in either a TCB or uTCB currently makes use of the FORTEZZA card by directly making CI_Library calls from the Message Security Protocol (MSP). The second scenario (**Figure 6**) shows what has to be done in order to incorporate the recommended CAPI solution. The CI_Library calls within MSP would have to be replaced with GSS-API calls. Meanwhile, translation code will have to be developed between Cryptoki and the CI_Library calls. Finally, a thin layer of translation code between GSS-API and Cryptoki will have to be included. The benefit of this solution is that it requires a one-time alteration to MSP, and then requires no further alterations to MSP for future FORTEZZA releases.