# *X/Open Preliminary Specification*

## Generic Cryptographic Service API (GCS-API)

## Base

*X/Open Company Ltd.*

X/Open Preliminary Specification

Generic Cryptographic Service API (GCS-API) Base

ISBN: 1-85912-195-0

X/Open Document Number: P442

Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.org

# Contents

## List of Figures

## List of Tables

# Preface

**X/Open**

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable open system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and so protects investment in existing software while enabling additions and enhancements. It also allows users to move between systems with a minimum of retraining.

X/Open defines this CAE in a set of specifications which include an evolving portfolio of application programming interfaces (APIs) which significantly enhance portability of application programs at the source code level, along with definitions of and references to protocols and protocol profiles which significantly enhance the interoperability of applications and systems.

The X/Open CAE is implemented in real products and recognised by a distinctive trade mark — the X/Open brand — that is licensed by X/Open and may be used on products which have demonstrated their conformance.

**X/Open Technical Publications**

X/Open publishes a wide range of technical literature, the main part of which is focussed on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys, and business titles.

There are two types of X/Open specification:

- *CAE Specifications*

  CAE (Common Applications Environment) specifications are the stable specifications that form the basis for X/Open-branded products. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement an X/Open CAE specification can enjoy the benefits of a single, widely supported standard. In addition, they can demonstrate compliance with the majority of X/Open CAE specifications once these specifications are referenced in an X/Open component or profile definition and included in the X/Open branding programme.

  CAE specifications are published as soon as they are developed, not published to coincide with the launch of a particular X/Open brand. By making its specifications available in this way, X/Open makes it possible for conformant products to be developed as soon as is practicable, so enhancing the value of the X/Open brand as a procurement aid to users.

- *Preliminary Specifications*

  These specifications, which often address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations, are released in a controlled manner for the purpose of validation through implementation of products. A Preliminary specification is not a draft specification. In fact, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE specification.

  Preliminary specifications are analogous to the *trial-use* standards issued by formal standards organisations, and product development teams are encouraged to develop products on the basis of them. However, because of the nature of the technology that a Preliminary specification is addressing, it may be untried in multiple independent implementations, and may therefore change before being published as a CAE specification. There is always the intent to progress to a corresponding CAE specification, but the ability to do so depends on consensus among X/Open members. In all cases, any resulting CAE specification is made as upwards-compatible as possible. However, complete upwards-compatibility from the Preliminary to the CAE specification cannot be guaranteed.

In addition, X/Open publishes:

- *Guides*

  These provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant. X/Open Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming X/Open conformance.

- *Technical Studies*

  X/Open Technical Studies present results of analyses performed by X/Open on subjects of interest in areas relevant to X/Open's Technical Programme. They are intended to communicate the findings to the outside world and, where appropriate, stimulate discussion and actions by other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism for X/Open to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of an X/Open technical activity. Although at the time of its publication, there may be an intention to progress the activity towards publication of a Specification, Guide or Technical Study, X/Open is a consensus organisation, and makes no commitment regarding future development and further publication. Similarly, a Snapshot does not represent any commitment by X/Open members to develop any specific products.

## Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision, in this case as the subject technology develops and to align with emerging associated international standards. X/Open makes a distinction between revised specifications which are fully backward compatible and those which are not:

- a new *Version* indicates that this publication includes all the same (unchanged) definitive information from the previous publication of that title, but also includes extensions or additional information. As such, it *replaces* the previous publication.

- a new *Issue* does include changes to the definitive information contained in the previous publication of that title (and may also include extensions or additional information). As such, X/Open maintains *both* the previous and new issue as current publications.

**Corrigenda**

Most X/Open publications deal with technology at the leading edge of open systems development. Feedback from implementation experience gained from using these publications occasionally uncovers errors or inconsistencies. Significant errors or recommended solutions to reported problems are communicated by means of Corrigenda.

The reader of this document is advised to check periodically if any Corrigenda apply to this publication. This may be done in any one of the following ways:

- anonymous ftp to ftp.xopen.org
- ftpmail (see below)
- reference to the Corrigenda list in the latest X/Open Publications Price List.

To request Corrigenda information using ftpmail, send a message to ftpmail@xopen.org with the following four lines in the body of the message:

```
open
cd pub/Corrigenda
get index
quit
```

This will return the index of publications for which Corrigenda exist. Use the same email address to request a copy of the full corrigendum information following the email instructions.

**This Document**

This document is a Preliminary Specification (see above) and is structured into the following sections:

- **Basic GCS-API**
  The Basic GCS-API comprises a set of functionality that is expected to meet the cryprographic service requirements of most general application developers. The Basic GCS-API section presents a simple overview of the types of cryptographic functions, a simplified model of the GCS-API architecture, and the minimum set of generic cryptographic functions that can support the requirements of general applications.
- **Advanced GCS-API**
  The Advanced GCS-API comprises an additional set of functionality that would only be used by applications that are developed to manage cryptographic policy and provide long term management of keys and the cryptographic service itself. The Advanced GCS-API section presents a more detailed description of the concepts and Architecture of the GCS-API and the additional functions.
- **Informative Appendices**
  A number of informative appendices are included providing discussion on implementation considerations, example walkthroughs of the use of the GCS-API in key exchange protocols, and other sundry matters.

### Basic GCS-API

- Chapter 1 is an introduction to the Basic GCS-API including an overview of cryptographic services, the GCS-API Programming Model and the concept of a *Cryptographic Context*.
- Chapter 2 presents an overview of the Basic GCS-API functions, explaining their use and providing some code examples.
- Chapter 3 defines the GCS-API data types, parameter passing conventions and defined constants necessary for the use of the Basic GCS-API.
- Chapter 4 presents the C-language functions that form the Basic GCS-API.

### Advanced GCS-API

- Chapter 5 is an introduction to the Advanced GCS-API providing a more detailed description of the scope and applicability of the GCS-API including discussion of the legal and security considerations that arise in the deployment of cryptographic services.
- Chapter 6 describes the key life cycle.
- Chapter 7 defines the logical data structures that underly the GCS-API.
- Chapter 8 presents an overview of the Advanced GCS-API functions.
- Chapter 9 defines the additional GCS-API data types, parameter passing conventions and defined constants necessary for the use of the Advanced GCS-API.
- Chapter 10 presents the C-language functions for general cryptographic services and protected key management services that form part of the Advanced GCS-API.
- Chapter 11 presents the C-language functions for clear key management services.
- Chapter 12 describes the conformance requirements.

### Informative Appendices

- Appendix A presents factors to be considered by implementations of this specification.
- Appendix B presents a set of example template CCs that could be used as the basis for supporting a majority of common cryptographic uses.
- Appendix C presents walkthroughs of some typical uses of cryptographic services to demonstrate the applicability of this specification.
- Appendix D lists additional functional areas that have been rules out of scope of this current specification but which may considered for inclusion in a future specification.
- Appendix E presents an example of key test pattern generation and verification.
- Appendix F presents a discussion of key parity.
- A glossary and index are provided.

### Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for filenames, and C-language keywords, type names, data structures and their members.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  - — C-language variable names, for example, substitutable argument prototypes
  - — C-language functions; these are shown as follows: *name*( ).
- Normal font is used for the names of constants and literals.
- The notation **<file.h>** indicates a header file.
- The notation [EABCD] is used to identify a C-language return code EABCD.
- Syntax, code examples and user input in interactive examples are shown in `fixed width` font.
- Variables within syntax statements are shown in *`italic fixed width font`*.
- Language-independent functions and arguments use ***bold italic*** font, for example, ***function***( ) and ***argument***.

# *Trade Marks*

Kerberos™ is a trade mark of the Massachusetts Institute of Technology.

OSF™ is a trade mark of The Open Software Foundation, Inc.

X/Open® is a registered trade mark, and the ''X'' device is a trade mark, of X/Open Company Limited.

# Acknowledgements

X/Open gratefully acknowledges the work of the X/Open Cryptographic Working Group in the development of this specification.

**Document Development**

This specification is the result of the input and discussion of many ideas and concepts and the comparison of these with existing implementations. Specific input and development effort has been provided by:

BULL, S.A.
Hewlett Packard
International Business Machines Corporation (IBM)
International Computers Limited (ICL)
USA National Institute of Standards and Technology (NIST)
USA National Security Agency (NSA)
Olivetti Systems and Networks s.r.l
OpenVision
Siemens Nixdorf
Trusted Information Systems, Inc

**Other Contributions**

The following organisations have contributed to this specification by reviewing drafts.

Fischer International
RSA Data Security, Inc

# Referenced Documents

The following documents are referenced in this specification:


RFC 1510
: Internet Proposed Standard, The Kerberos Network Authentication System, John Kohl, B.Clifford Neuman, issue 5.2, 21 April 1993.

CESG Memo
: CESG Memorandum No.1 Issue 1.2 Oct 1992, Glossary of Security Terminology.

Federal Criteria
: Federal Criteria Version 1.0 Dec 1992, Federal Criteria for Information Technology Security.

ISO / IEC 7498-2
: ISO / IEC 7498-2: 1989, Information Processing Systems — Open Systems Interconnection — Basic Reference Model — Part 2: Security Architecture.

ISO / IEC 10181
: ISO / IEC 10181, Information Technology — Open Systems Interconnection — Security Frameworks in Open Systems —

  10181-1: Part 1: Security Frameworks Overview
  10181-2: Part 2: Authentication Framework
  10181-3: Part 3: Access Control
  10181-4: Part 4: Non-repudiation Framework
  10181-5: Part 5: Integrity Framework
  10181-6: Part 6: Confidentiality Framework
  10181-7: Part 7: Security Audit Framework

ITSEC
: Information Technology Security Evaluation Criteria, Provisional Harmonised Criteria, June 1991, Version 1.2, published by the Commission of the European Communities.

OIW OSI Security
: Stable Implementation Agreements for Open Systems Interconnection Protocols: Part 12 — OS Security, December 1994.

POSIX.0
: IEEE Std 1003.0 / D15, June 1992, Draft Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 0.

PKCS #1
: RSA Data Security, Inc. Public-Key Cryptography Standards (PKCS) PKCS #1: RSA Encryption Standard, November 1993.

PKCS #3
: RSA Data Security, Inc. Public-Key Cryptography Standards (PKCS) PKCS #3: Diffie-Hellman Key-Agreement Standard.

PKCS #8
: RSA Data Security, Inc. Public-Key Cryptography Standards (PKCS) PKCS #8: Private-Key Information Syntax Standard, November 1993.

X.509
: ISO / IEC 9594-8: 1990, Information Technology — Open Systems Interconnection — The Directory — Part 8: Authentication Framework, together with:

Technical Corrigendum 1: 1991 to ISO / IEC 9594-8: 1990.

The following X / Open documents are referenced in this specification:

Base GSS-API
X / Open CAE Specification, December 1995, Generic Security Service API (GSS-API) Base (ISBN: 1-85912-131-4, C441).

XDSF
X / Open Guide, December 1994, Distributed Security Framework (ISBN: 1-85912-071-7, G410).

The following publications provide a more detailed description of cryptography and its uses:

SCHNEIER
Bruce Schneier, *Applied Cryptography*, John Wiley & Sons, 1996.

*Chapter 1*

# Basic GCS-API - Introduction

The increasing use of network services such as the Internet has enhanced awareness of the need for security in distributed computer systems, particularly in the light of the publicity surrounding successful breaches of security, for example, the *sniffing* of user identities and passwords passed in the clear over the Internet.

Security services providing for authentication of identities, data-origin authentication, non-repudiation, data separation, confidentiality and integrity protection rely on underlying cryptographic services. However, the wide-spread and common use of cryptography within applications is hindered by two things:

- the lack of agreed application programming interfaces
- legislative constraints that may apply to the supply, use, export or import of the technology

It has long been recognised that a standard application programming interface specification is needed for cryptographic services and this document addresses that need.

## 1.1 Structure of document

As described in the Preface, this document is structured into two major sections, a Basic section and an Advanced section.

The first part of the specification, the **Basic** section, presents a simple overview of the types of cryptographic functions, a simplified model of the GCS-API architecture, and the minimum set of generic cryptographic functionality that can support the requirements of general applications wishing to use cryptographic service. It is expected that the majority of the cryptographic service needs of most application developers can be met by the Basic GCS-API functionality.

The second part of the specification, the **Advanced** section, presents a more detailed description of the concepts, detailed data structures and additional sets of functions that would only be used by applications that are developed to manage cryptographic policy and provide long term management of keys and the cryptographic service itself.

## 1.2 Scope of Basic GCS-API

The scope of the basic section of this specification is to provide cryptographic services in support of both algorithm unaware and algorithm aware applications. As such, the interface specification is provided for use by programmers who develop applications that rely on cryptographic services and key management services.

The objectives to be met by the interfaces defined in this specification may be categorised as functional and non-functional. In addition, legal constraints on the use of some cryptographic services need to be accommodated, see Chapter 5.

### 1.2.1 Functional Objectives of Basic GCS-API

A common set of functions are required to support all types of callers. These are termed *General Application Cryptographic Services* and comprise the following:

1. data encipherment and decipherment
2. integrity checkvalue generation and verification
3. production of irreversible hash of data
4. generation of random numbers

Key management applications require the following additional functions:

1. generation, derivation and deletion of keys
2. export and import of keys

### 1.2.2 Non-Functional Objectives

The non-functional requirements to be supported by this specification are the requirements that make this specification *Generic* and include:

1. the API shall be cryptographic algorithm independent
2. the API shall be application independent
3. the API shall be cryptographic subsystem independent. (That is, appropriate to both hardware and software implementations)
4. the API shall not impose a particular placement of access control to cryptographic services within an operating system kernel
5. the API shall not constrain future extensibility.

## 1.3 Overview of Cryptographic Services

This subsection provides a brief introductory description of cryptographic services for those readers who are unfamiliar with the subject. For a more detailed treatise on the subject readers are referred to Schneier (*see Referenced Documents).*

Cryptographic services provide a set of functions for encoding and decoding information so that the information may be stored or exchanged securely. Cryptographic functions provide a basis for implementing the following security services:

- Confidentiality of information, preventing unauthorised disclosure
- Integrity of information, detecting unauthorised modification
- Origin authentication, providing verification of the origin of information.

Examples of the basic models of the application of cryptographic services are functions for the *encipherment* and *decipherment* of data, and the generation of *Hash Values* or *Digital Signatures* on sets of data. In addition functions to support key management and distribution are important.

### 1.3.1 Encipher and Decipher Functions

The basic concept underlying cryptography is the enciphering of data. Encipher functions encode a set of data, termed *cleartext* or *plaintext*, into a protected format termed *ciphertext* using a reversible mechanism. The ciphertext may be stored or exchanged with a reduced risk of unauthorised disclosure of the data. A corresponding decipher function can be used to decode *ciphertext* back into its corresponding **cleartext** form. Thus:

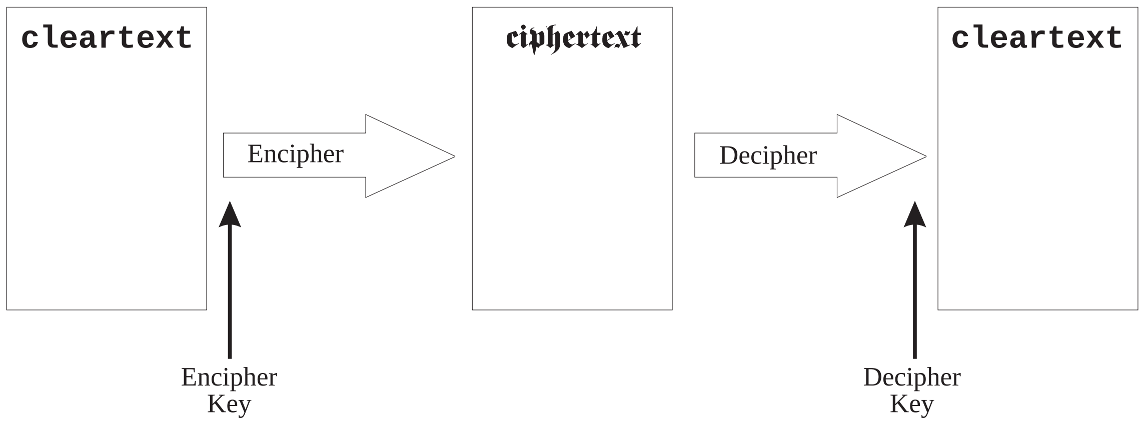


**Figure 1-1** Encipher and Decipher Functions

The encoding is controlled by the algorithm used and a secret value termed a key. The protection afforded to the ciphertext depends upon the strength (but not the secrecy) of the algorithm and the protection of the key used to control the algorithm. Encipherment functions preserve all the original data represented by the cleartext. This type of function is the basis of the provision of information confidentiality services.

### 1.3.2 Symmetric-Key and Asymmetric-Key Encipherment

There are two classes of encipherment algorithm:

**Symmetric-Key Algorithms - (Secret-Key Algorithms)**
are algorithms in which the encipher key and the decipher key are identical. For the exchange of enciphered data a single key value must be shared between the originator and the recipient and protected by both parties. For this reason these types of algorithm are also termed *Secret-Key* algorithms.

**Asymmetric-Key Algorithms - (Public-Key Algorithms)**
are algorithms in which the encipher key and decipher key are different. The encipher and decipher keys are generated as a pair by a single operation. Data enciphered by using one key of the pair may be deciphered using the other key of the pair. For the exchange of enciphered data each party to the exchange makes one of their own pair of keys public, the *public-key*, and keeps the other key private, the *private-key*. The originator of an exchange enciphers the data using the *public-key* of the recipient. The recipient is then able to decipher the received data using his own *private-key*.

### 1.3.3 Hash (Unprotected Checksum) Functions

Hash functions encode a set of data that may be of variable length using a one-way function to create a unique fixed length hash value or message digest of the set of data. The hash value is unprotected in the sense that it does not depend upon any secret value component and any individual with the same input data and same algorithm can generate the hash value.

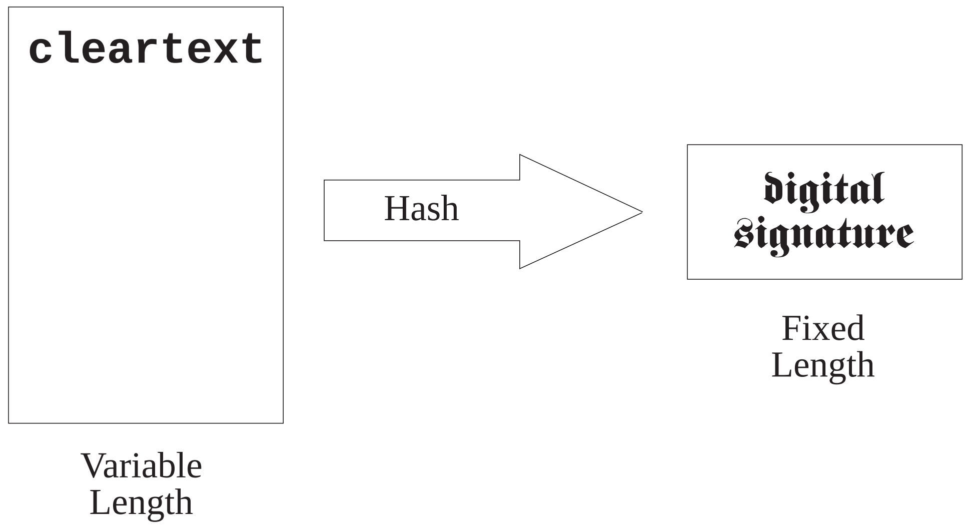


**Figure 1-2** Generate Hash value

A hash function does not preserve the original data represented by the cleartext and therefore the original cleartext cannot be recovered from a hash value. The value of these types of function are that the hash value is unique to a particular input cleartext and can therefore be used to check that the corresponding cleartext has not been modified.

A hash function is the basis of the provision of information integrity services. The hash value generated by the originator of the information is stored or exchanged with the *cleartext*. The recipient is able to regenerate the hash value from the received cleartext and verify that cleartext is unmodified by comparing the newly generated hash value with that received with the information.

### 1.3.4 Digital Signature (Protected Checksum) Functions

An asymmetric encipher function and a hash function may be used in combination to provide a digital signature service. The Digital Signature is protected in the sense that its value depends upon the originator's private key and it can therefore only be generated by an individual possessing that key.

First a hash value is produced by the hash function. This is then enciphered using the asymmetric encipher function using the originator's private key.

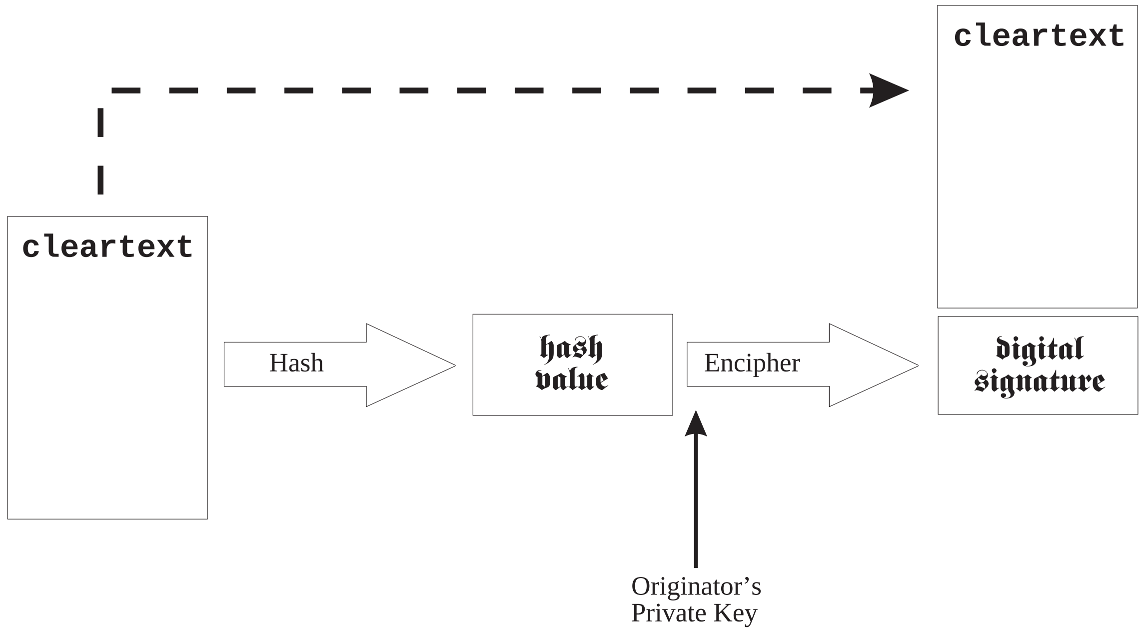


**Figure 1-3** Generate Digital Signature

The recipient may verify the digital signature by comparing the values obtained by recomputing the hash value of the received cleartext and comparing this with the value obtained by deciphering the digital signature using the originator's public key.

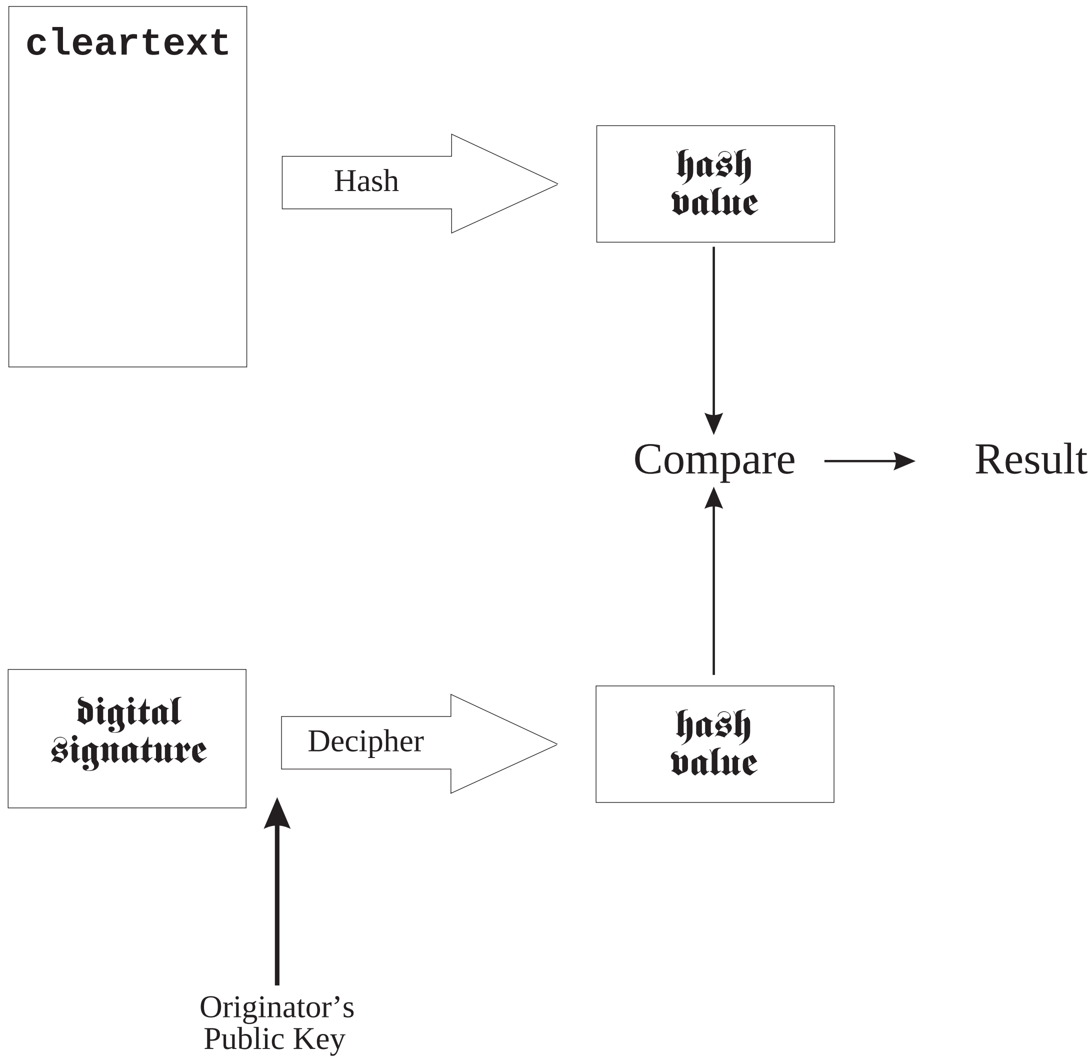


**Figure 1-4** Verify Digital Signature

### 1.3.5 Key Management Functions

In order to exchange cryptographically protected information then the parties exchanging the information require to have access to the appropriate keys. This means that cryptographic keys, or information permitting their derivation, also have to be exchanged.

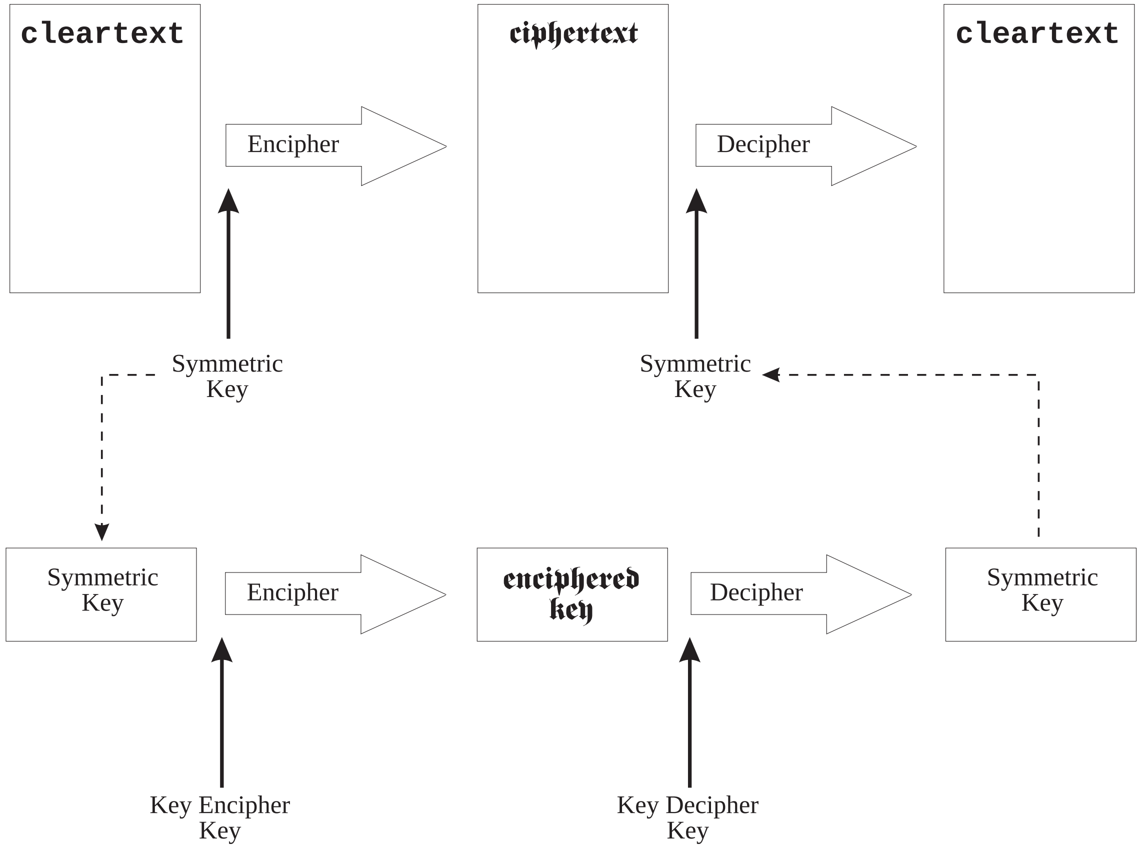


**Figure 1-5** Key Encrypting Key

The strength of the protection of data using cryptographic services depends critically upon the protection of the key values used to control the algorithms. Functions to securely create and support the secure distribution of cryptographic keys are therefore an essential part of any cryptographic service.

Keys may be generated or derived. A key generation function will generate a key based on random information. A key derivation function will derive a key based upon some caller defined input string, such as a pass phrase.

To distribute keys securely they are normally protected by enciphering under a Key Exchange Key, or Key Encrypting Key. Note that the individual parties exchanging keys need to have previously distributed by some other method the Key Exchange key.

## 1.4 The GCS-API Programming Model

The Generic Cryptographic Service Application Program Interface (GCS-API) is a set of interfaces to a Cryptographic Support Facility (CSF) that may support a number of different cryptographic algorithms dependent upon the implementation. It also provides support for key management on behalf of individual applications and shared key management between applications. This is illustrated in Figure 1-6.

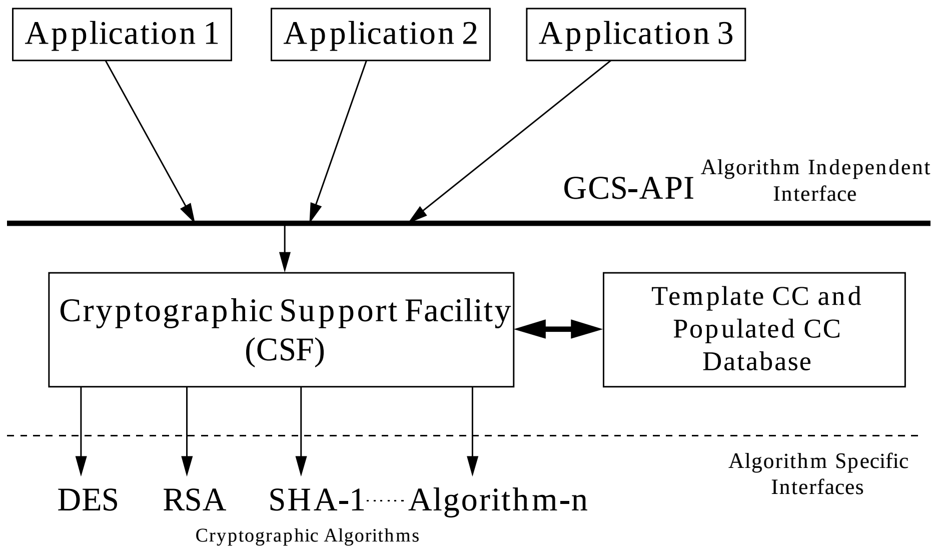


**Figure 1-6** Basic CSF Model

The interface presented by the GCS-API supports the development of portable applications by being:

**Algorithm Independent**
The GCS-API may hide the details and complexities and specific algorithms from callers. For example, a caller may invoke an encipher function without needing to be aware of which algorithm is being used nor of the specific parameters required by that algorithm. However, the GCS-API also supports algorithm specific callers that require to use a specific set of algorithms.

**Implementation Independent**
The GCS-API hides the details of the implementation from callers. For example, whether the implementation is in software, hardware, or a combination of both. An application can therefore be unaware of the necessity to open a physical device to access a hardware implementation.

## 1.5 Cryptographic Context (CC)

In invoking a cryptographic operation it is insufficient for a caller to simply supply the input data and a key. Other information has to be assembled such as which algorithm is to be used, how it is to be used and algorithm and key specific parameters.

GCS-API algorithm independence is achieved by using the concept of a *Cryptographic Context* (CC). A CC is an protected object that is opaque to callers of the GCS-API and which encapsulates all the information pertaining to the context of the cryptographic operation to be performed. A CC includes the algorithm identity, algorithm specific parameters, key specific parameters, and optionally a key. The contents of a CC are detailed in Chapter 7 in the Advanced GCS-API section. Callers of the Basic GCS-API do not need to be aware of the contents of a CC.

A CSF maintains a database of CCs that may be referenced by name by callers. There are two types of CC, those that are populated with a key and those that are not.

**Template CCs**
: Template CCs are those CCs that do not contain a key and cannot be used directly in cryptographic operations. The purpose of these types of CCs is to provide templates applicable to the algorithms supported by the particular CSF implementation and which configure the use of the cryptographic services in accordance with the local site security policy.

**Populated CCs**
: Populated CCs are those CCs that do contain a key and may be used directly in cryptographic operations. An implementation for use in a multi-user environment will enforce an access control policy on the use of populated CCs.

The general method of use of a CC is for a key management application to:

- retrieve a template CC appropriate to the functions it wishes to perform,
- to populate that CC by calling on the CSF to generate a key, and then
- either use that CC itself in subsequent calls to cryptographic operations on the CSF, or
- store the CC with an appropriate name for subsequent use by other callers.

A general application will retrieve a previously stored populated CC from the CSF for use in its operations.

The advance section of the GCS-API includes functions for creating template CCs. See Chapter 8.

### 1.5.1 Naming of Template CCs

The ability to name CCs may be used to support both algorithm aware and algorithm independent applications. A CC name may be used to identify the specific contents and purpose of the CC, for example RSA_SIGN_SHA-1, for use by algorithm aware callers. Additionally a CC name may be used to identity local default algorithms, for example LOCAL_SIGN, for use by algorithm unaware applications.

A possible set of defaults is:

| **CC Name** | **Meaning** |
|---|---|
| LOCAL_SYM_ENCIPHER_DECIPHER | Default symmetric encipher/decipher CC |
| LOCAL_ASYM_ENCIPHER | Default asymmetric encipher CC |
| LOCAL_ASYM_DECIPHER | Default asymmetric decipher CC |
| LOCAL_SIGN | Default signature generate CC |
| LOCAL_VERIFY | Default signature verify CC |
| LOCAL_HASH | Default hash CC |
| LOCAL_EXPORT | Default export key CC |
| LOCAL_IMPORT | Default import key CC |

**Table 1-1** Default CC Names

Appendix B presents a set of example Template CC definitions for common algorithms.

*Chapter 2*

# *Basic GCS-API Services*

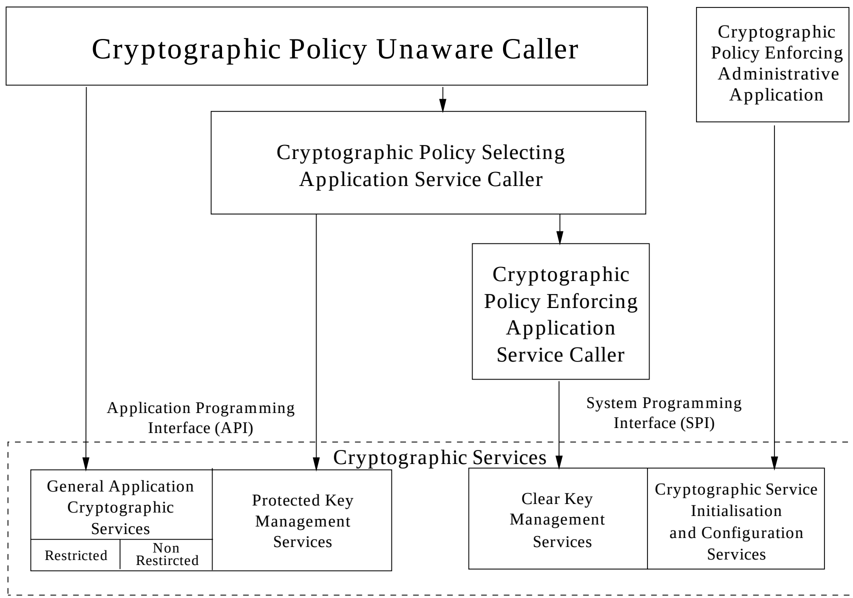


**Figure 2-1** CSF Services

The CSF services comprise both operational and management services and are illustrated in Figure 2-1.

They include the following categories:

- General Cryptographic Services (Part of the API)
- Protected Key Management Services (Part of the API)
- Clear Key Management Services (Part of the SPI)
- Cryptographic Service Initialisation and Configuration Services (Not within the current scope of this specification.)

This chapter describes the basic services supported by the GCS-API the advanced services supported by the GCS-API are described in Chapter 5. The basic services comprise the General Cryptographic Services together with a subset of the Protected key Management Services. Each subsection lists the functions supported and the GCS Authorities, if any, required by a caller in order to successfully invoke the function. The detailed manual page for each of these functions is included in Chapter 4.

GCS_Authorities relate to the type of authority a caller of the CSF has for the enforcement of cryptographic security policy. The GCS_Authorities have been defined to support the principles of the separation of duties and of least privilege.

The GCS_Authorities of a caller of the GCS-API are established in an implementation defined manner when a caller initialises a session with a CSF. A caller is not required to manipulate GCS_Authorities during the use of the GCS_API but should be aware that a call may fail because of inadequate authorisation.

## 2.1 Session Management

| Function | GCS Authorities |
|---|---|
| gcs_initialise_session | - |
| gcs_terminate_session | - |

**Table 2-1** CSF Session Management Functions

CSF Session Management functions are used to establish and release connections with the CSF. These functions provide for the authentication of the caller and the establishment of a security context for the session created between a caller and the CSF.

The security context is represented by an protected opaque object to which a handle is returned to a caller initiating a session. This session_context is included as a parameter to every call to the GCS-API to provide a method of continuous authentication and to support stateless implementations of the CSF. The security context includes any necessary identity authentication and authorisation attributes, including GCS-API Authorities.

The function *gcs_initialise_session* is used to initiate a session, *gcs_terminate_session* is used to terminate a session and release the security context.

## 2.2 Cryptographic Context Retrieval Functions

| Function | GCS Authorities |
|---|---|
| gcs_delete_cc | - |
| gcs_list_cc | - |
| gcs_retrieve_cc | - |

**Table 2-2** Cryptographic Context Retrieval Functions

A cryptographic key has to be protected from disclosure and has to be used in the context of the algorithm and associated parameters that govern its use. To simplify the manipulation of this information by general applications the GCS-API groups a key and other related data into a protected structure termed a Cryptographic Context (CC).

Cryptographic Contexts may be stored under the control of the CSF as one of two types:

- Template CCs that include all the necessary context information necessary to perform a particular type of operation with the exception of a key. These types of CC are created by administrators of the CSF to act as templates for use by other callers of the CSF. The creation of these templates permits the set of cryptographic policies for the use of the CSF to be predefined.
- Populated CCs which include all the necessary context information to perform a particular type of operation including a key. These types of CCs are created by key management applications.

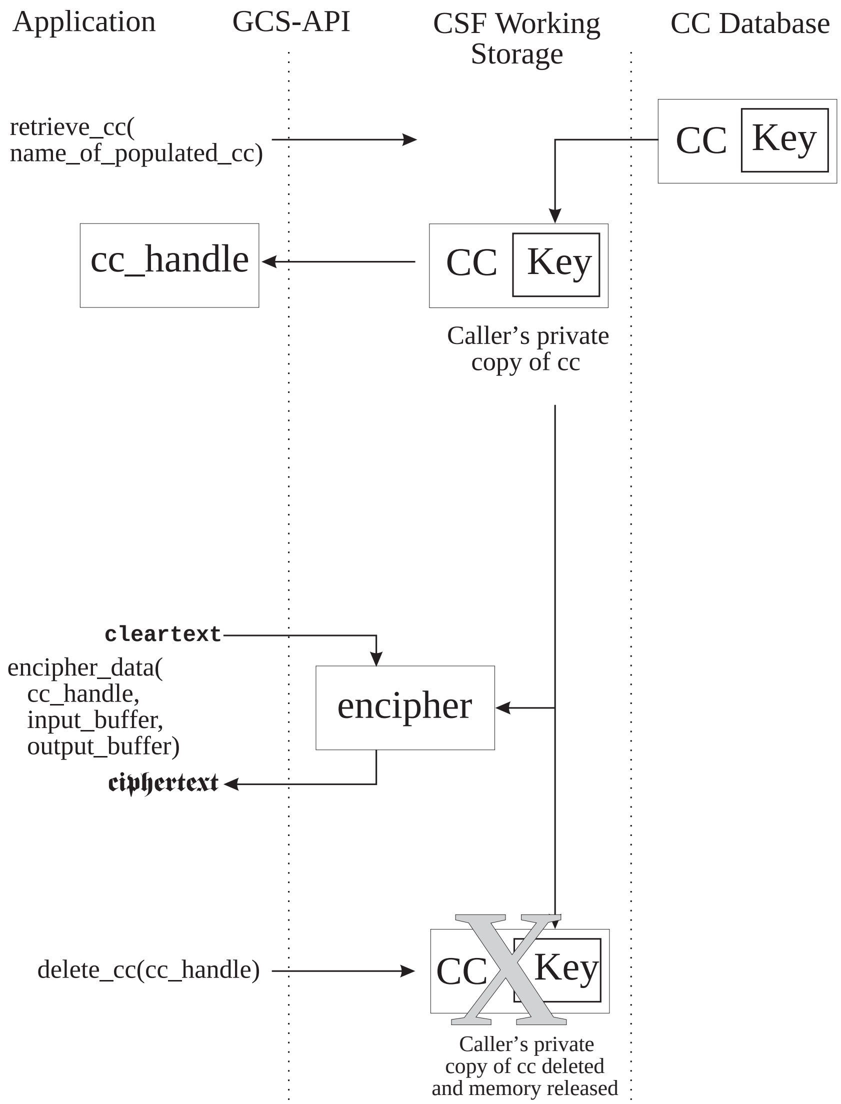


**Figure 2-2** Retrieval and Use of a Populated CC

*gcs_retrieve_cc* enables a caller to retrieve a handle to a CC so that it may be used. *gcs_delete_cc* is used to delete the handle to a CC and release any resources associated with its use by that caller. *gcs_list_cc* provides for a caller to query the CSF for the names of stored CCs that it may attempt to retrieve.

Here is an example printing all the cc names allowed for this application.

```
main()
{
   gcs_buffer_desc  my_cc_name;
   OM_unit32        index;
   OM_unit32        returnCode=GCS_S_CONTINUE_NEEDED;

   .........
   for (index=0; returnCode == GCS_S_CONTINUE_NEEDED; index++)
```

```
    {
       returnCode = gcs_list_cc(&minor_status,
                                &session_context,
                                index,
                                GCS_NULL,  /* no need for domain */
                                &my_cc_name,
                                GCS_NULL);

      if (my_cc_name.value != GCS_NULL)
       printf(" the cc name is = %s \n",my_cc_name.value);

    }

    gcs_release_buffer(&minor_status,&session_context,&my_cc_name);
    ........
}
```

A general application may retrieve a previously populated CC that has been stored under the control of the CSF for shared use by a number of applications, for example a user's private key. This is illustrated in Figure 2-2

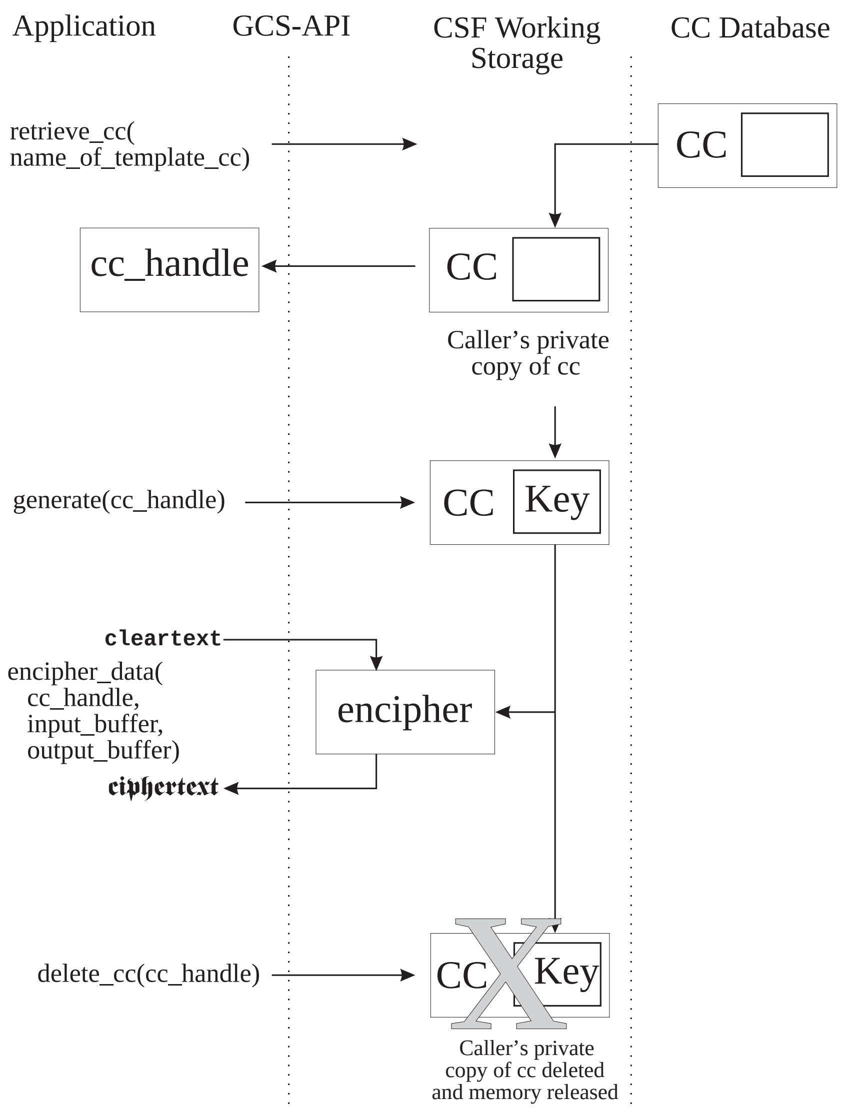


**Figure 2-3** Retrieval and use of a Template CC

A key management application may retrieve a handle to a template CC for subsequent population with a key. This is illustrated in Figure 2-3.

## 2.3 Key Creation

| Function | GCS Authorities |
|---|---|
| gcs_derive_key | GCS_C_SELECTION |
| gcs_generate_key | GCS_C_SELECTION |

**Table 2-3** Key Creation Functions

Before a template CC may be used for cryptographic operations it requires populating with a key. This is achieved using *gcs_derive_cc* to derive a key from an input parameter, for example a user supplied string, or *gcs_generate_key* to internally generate a key value or key value pair. This is illustrated in Figure 2-3.

## 2.4 Hash and Signature Functions

| Function | GCS Authorities |
|---|---|
| gcs_generate_checkvalue | - |
| gcs_verify_checkvalue | - |
| gcs_generate_hash | - |
| gcs_generate_random_number | - |

**Table 2-4** Hash and Signature Functions

The cryptographic hash and signature functions listed above provide the basis for integrity and digital signature operations and will be supported by all CSF implementations. *gcs_generate_checkvalue* and *gcs_verify_checkvalue* generate cryptographically protected hash values (e.g., digital signatures). *gcs_generate_hash* generates a hash of the supplied input data. *gcs_generate_random_number* is used to generate a cryptographically strong random number.

A code example for the retrieval of a populated CC and its use to generate check value is given below. This example does include all the necessary code to create a compilable program but only emphasises the GCS-API calls necessary.

```
BOB retrieves his key and use it to sign some data.


#include <libgcs.h>


main()
{
   OM_uint32               minor_status ;
   OM_uint32               ret ;
   gcs_session_context_t   session_context ;
   gcs_cc_t                bob_s_crypto_context ;

#define BUFFER_SIZE 256

   gcs_buffer_desc         cc_name ;
   gcs_buffer_desc         init_param ;
   gcs_buffer_desc         buffer ;
```

```
gcs_buffer_desc         check_value ;

char user_s_CC_name[MAX_CC_NAME_LENGTH]     = "BOB_S_CC " ;
char buffer_data[BUFFER_SIZE];

int i ;

/*** Initialisation of a session between bob and the Cryptographic
   * Security Module.
   * This is omitted for clarity.


   /***
   * Retrieve the cryptographic context from the database.
   */

cc_name.length = MAX_CC_NAME_LENGTH ;
(char *) cc_name.value = user_s_CC_name ;

if ( (ret = gcs_retrieve_cc(&minor_status, &session_context, NULL,
                        &cc_name, NULL, FALSE, &bob_s_crypto_context))
      != GCS_S_COMPLETE) {
   fprintf(stderr, "Error %d in gcs_retrieve_cc0, ret) ;
   exit (-1) ;
}

/***
   * Fill buffer with data to be signed.
   */



/***
   * Compute checkvalue of the buffer
   */

if ( (ret = generate_check_value(&minor_status,&session_context,&buffer,
                            NULL,GCS_C_ONLY,&bob_s_crypto_context,
                            NULL,&check_value))
      != GCS_S_COMPLETE) {
   fprintf(stderr, "Error %d in generate_check_value0, ret) ;
   exit (-1) ;
}


/***
   * Store or transmit the computed check value.
   */

/***
   * Release buffers and delete cryptographic context
   */
```

```
    if ( (ret = gcs_delete_cc(&minor_status,&session_context,
                              &bob_s_crypto_context))
        != GCS_S_COMPLETE) {
       fprintf(stderr, "Error %d in gcs_delete_cc0, ret) ;
       exit (-1) ;
    }


    if ( (ret = gcs_release_buffer(&minor_status,&check_value))
        != GCS_S_COMPLETE) {
       fprintf(stderr, "Error %d in gcs_release_buffer0, ret) ;
       exit (-1) ;
    }


     return (0) ;

}
```

## 2.5 Data Encipherment Functions

| Function | GCS Authorities |
|---|---|
| gcs_encipher_data | GCS_C_ENCIPHER_DECIPHER |
| gcs_decipher_data | GCS_C_ENCIPHER_DECIPHER |
| gcs_protect_data | GCS_C_ENCIPHER_DECIPHER |
| gcs_decipher_verify | GCS_C_ENCIPHER_DECIPHER |

**Table 2-5** Data Encipherment Functions

The data encipherment operations listed above provide the basis for confidentiality operations. Legislative constraints on the use or supply of cryptographic services for data encipherment means that these functions may not be supported by all CSF implementations or may have operational constraints imposed on them and callers may require specific authorisation to use them, as represented by the GCS_C_ENCIPHER_DECIPHER GCS Authorisation.

*gcs_encipher_data* and *gcs_decipher_data* provide for the simple enciphering and deciphering of a set of data.

*gcs_protect_data* provides for the simultaneous enciphering and generation of a hash value or digital signature over the same data for the purposes of providing both confidentiality and integrity, and possibly data origin authentication. *gcs_decipher_and_verify* provides for the simultaneous deciphering and verification of a hash value or digital signature associated with the received ciphertext.

## 2.6 Cryptographic Context Storage Functions

| Function | GCS Authorities |
|---|---|
| gcs_store_cc | GCS_C_SELECTION or GCS_C_KEY_USAGE |
| gcs_remove_cc | GCS_C_SELECTION or GCS_C_KEY_USAGE |

**Table 2-6** Cryptographic Context Storage Functions

These functions provide for the storage of CCs under the control of the CSF and their subsequent removal. See Figure 2-4.

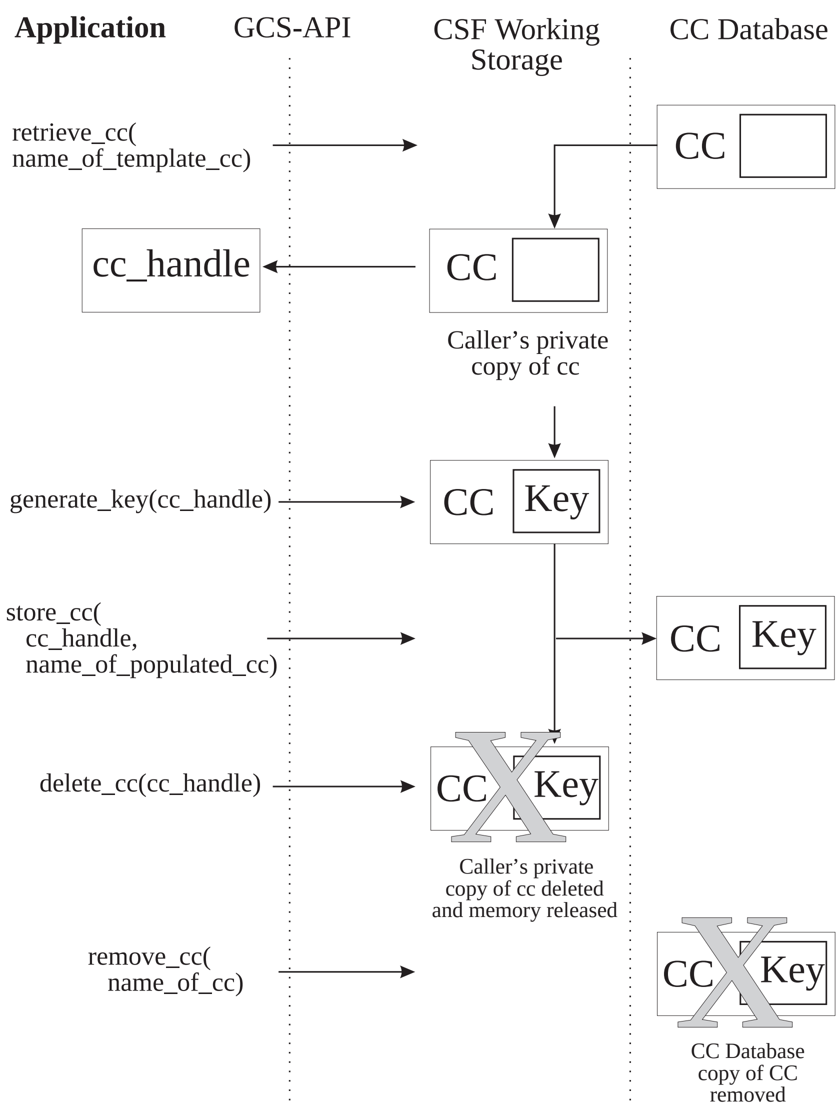


**Figure 2-4** CC Storage Management Functions

*gcs_store_cc* provides for the storage of a CC and the assignment of a caller defined name to the stored CC. The act of storage provides for the global referencing of that CC by any caller of the CSF subject to any authorisation policy enforced by the CSF. *gcs_remove_cc* removes a CC from

CSF controlled storage and therefore it is then no longer available for use.

The code example below is of the retrieval of a Template CC, its population with a key, and storage as a populated cc for subsequent use.

```
/***
   *  Retrieve a MD5+RSA cryptographic context from the database,
   *  populate it with a key and store it as bob’s crypto context
   */


#include <libgcs.h>


main()
{
   OM_uint32               minor_status ;
   OM_uint32               ret ;
   gcs_session_context_t   session_context ;
   gcs_cc_t                template_cc ;

   gcs_buffer_desc         cc_name ;
   gcs_buffer_desc         init_param ;
   char      admin_name[MAX_USER_NAME_LENGTH]  = "ADMIN " ;
   char      admin_pswd[MAX_PSWD_LENGTH]             = "MGT_PSWD " ;
   char template_cc_name[MAX_CC_NAME_LENGTH]   = "RSA-SIGN-MD5 " ;
   char bob_s_CC_name[MAX_CC_NAME_LENGTH]      = "BOB_S_CC " ;


   /*** Initialisation of a session between administrator and the Cryptographic
      * Security Module.
      *  This has been omitted for clarity.
      */

   /***
      * Retrieve a template cryptographic context from the database,
      * containing RSA and MD5 algorithms.
      */

   cc_name.length = MAX_CC_NAME_LENGTH ;
   (char *) cc_name.value = template_cc_name ;

   if ( (ret = gcs_retrieve_cc(&minor_status, &session_context, NULL,
                           &cc_name, NULL, FALSE, &template_cc))
        != GCS_S_COMPLETE) {
      fprintf(stderr, "Error %d in gcs_retrieve_cc0, ret) ;
      exit (-1) ;
   }


   /***
```

```
        * Generate a key and populate the cryptographic context with it,
        * and then store the new cryptographic context in the database under
        * the name 'BOB_S_CC'.
        */

     if ( (ret = gcs_generate_key(&minor_status, &session_context, &template_cc))
          != GCS_S_COMPLETE) {
        fprintf(stderr, "Error %d in gcs_generate_key0, ret) ;
        exit (-1) ;
     }

     (char *) cc_name.value = bob_s_CC_name ;
     if ( (ret = gcs_store_cc(&minor_status, &session_context, NULL,
                      &bob_s_CC_name, &template_cc, NULL, NULL, NULL))
          != GCS_S_COMPLETE) {
        fprintf(stderr, "Error %d in gcs_store_cc0, ret) ;
        exit (-1) ;
     }

     /***
        * Release buffers and delete cryptographic context
        */

     if ( (ret = gcs_delete_cc(&minor_status,&session_context,
                       &template_cc))
         != GCS_S_COMPLETE) {
        fprintf(stderr, "Error %d in gcs_delete_cc0, ret) ;
        exit (-1) ;
    }



     return (0) ;
  }
```

## 2.7 Key Exchange Functions

| Function | GCS Authorities |
|---|---|
| gcs_export_key | GCS_C_KEY_USAGE |
| gcs_import_key | GCS_C_KEY_USAGE |
| gcs_key_agreement | GCS_C_KEY_USAGE |

**Table 2-7** Key Exchange Functions

The key exchange functions provide for the encapsulation of a key into an object protected by a key exchange key (KEK) for the purposes of exchanging the key with another CSF or of binding the key with an object that has been protected by the key for the purposes of messaging or data storage.

*gcs_export_key* provides for the export of a key from a supplied CC. *gcs_import_key* provides for the import of a key protected under a KEK and its insertion into a supplied CC. This is illustrated in Figure 2-5.

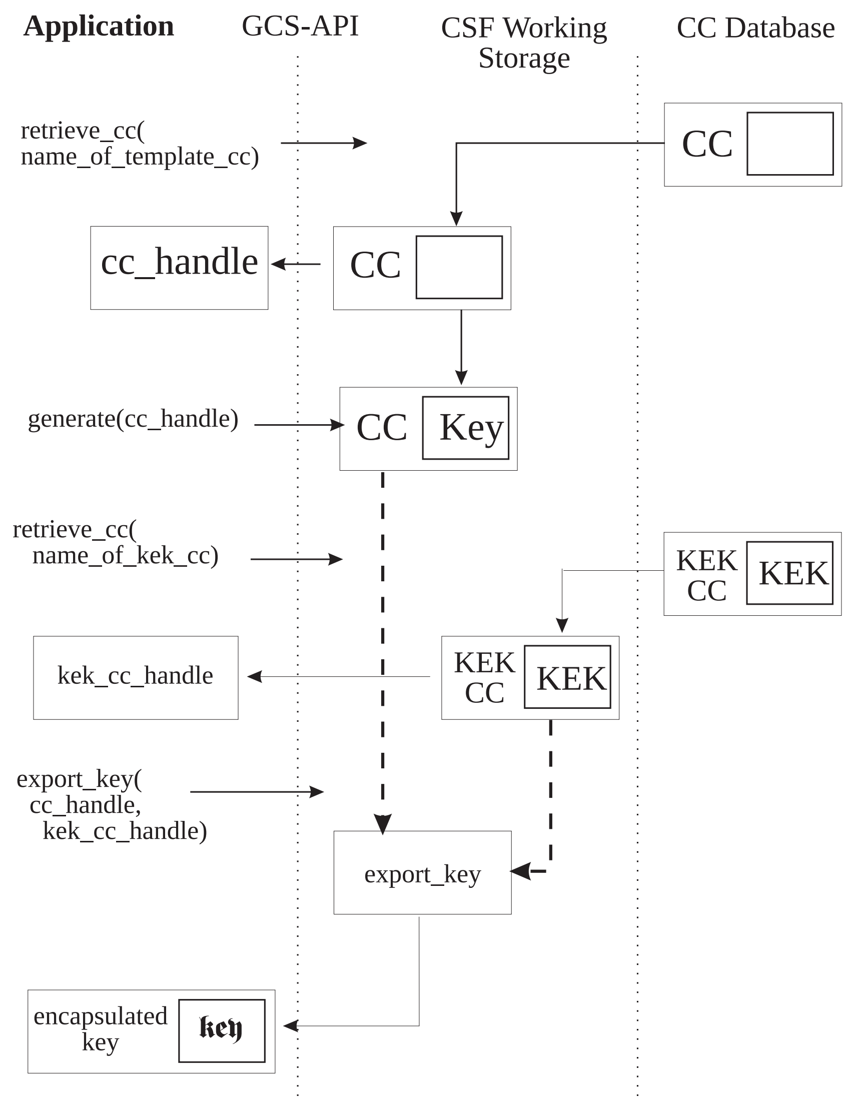


**Figure 2-5** Key Export

*gcs_key_agreement* provides support for more complex key exchange protocols as implemented by the CSF.

The GCS_C_KEY_USAGE authorisation is required by a caller of these functions as it is normally necessary to set key usage and key lifetime parameters within the CC.

## 2.8 GCS-API Utility Functions

| Function | GCS Authorities |
|---|---|
| gcs_get_csf_params | - |
| gcs_release_buffer | - |
| gcs_release_bit_string | - |

**Table 2-8** GCS-API Utility Functions

*gcs_get_csf_parameters* provides for the querying of implementation specific parameters such as maximum buffer size and the type of implementation (hardware, software, etc.).

*gcs_release_buffer* provides for the release of any buffers assigned by the GCS-API on a callers behalf.

*gcs_release_bit_string* provides for the release of any storage space allocated by *gcs_get_cc*, *gcs_generate_random*, *gcs_export_key* and *gcs_get_csf_params*.

*Chapter 3*

# *Basic Parameter Passing Conventions*

This chapter describes the data types used by the C-language versions of the basic GCS-API functions. It also explains calling conventions for these functions.

## 3.1 Structured Data Types

Wherever these GCS-API C-bindings describe structured data, only fields that must be provided by all GCS-API implementations are documented. Individual implementations may provide additional fields, either for internal use within GCS-API routines, or for use by non-portable applications.

## 3.2 Integer Types

GCS-API defines the following integer data type:

```
OM_uint32     32-bit unsigned integer
```

Where guaranteed minimum bit-count is important, this portable data type is used by the GCS-API routine definitions. Individual GCS-API implementations include appropriate **typedef** definitions to map this type onto a built-in data type.

## 3.3 String Data and Similar Data

### 3.3.1 Byte Strings

Many of the GCS-API routines take arguments and return values that describe contiguous multi-byte data. All such data are passed between the GCS-API and the caller using the **gcs_buffer_t** data type. This data type is a pointer to a buffer descriptor consisting of a **length** field, which contains the total number of bytes in the data, and a **value** field, which contains a pointer to the actual data:

```
typedef struct gcs_buffer_desc_struct{
   size_t   length;
   void    *value;
} gcs_buffer_desc, *gcs_buffer_t;
```

Storage for data passed to the application by a GCS-API routine using the **gcs_buffer_t** conventions is allocated by the GCS-API routine. The application may free this storage by invoking the *gcs_release_buffer*( ) routine. Allocation of the **gcs_buffer_desc** object is always the responsibility of the application; unused **gcs_buffer_desc** objects may be initialised to the value GCS_C_EMPTY_BUFFER.

### 3.3.2 Character Strings

Certain multi-octet data items may be regarded as simple Latin-1 character strings as defined in the ISO / IEC 8859-1 standard. An example of this is the *input-string* argument to *gcs_verify_key_pattern*( ). Character strings are passed between the application and the GCS-API using the **gcs_buffer_t** data type, defined earlier.

### 3.3.3 Bit Strings

Certain multi-octet data items may be regarded as simple bit strings. An example of this is the *export_data* argument to *gcs_export_key*.( ) Some GCS-API routines also return bit strings. The **gcs_bit_string_t** data type is a pointer to a buffer descriptor consisting of a **length** field, which contains the total number of bits, and a **bits** field which contains a pointer to the actual data, with the most significant bit first (in the lowest address bit).

```
typedef struct gcs_bit_string_desc_struct{
   OM_uint32   length;
   char    *bits;
} gcs_bit_string_desc, *gcs_bit_string_t;
```

Bit strings are passed between the application and the GCS-API using the **gcs_bit_string_t** data type.

Certain GCS-API functions return an array of bit strings. This is defined as follows:

```
typedef struct gcs_bit_string_set_desc_struct {
       OM_uint32           count;
       gcs_bit_string_t    bit-strings;
} gcs_bit_string_set_desc, *gcs_bit_string_set_t
```

### 3.3.4 Opaque Data Types

Certain multi-octet data items are considered opaque data types at the GCS-API, because their internal structure only has significance to the CSF. Examples of such opaque data types are the

- *session_context* argument to all GCS-API functions.
  This is opaque to the GCS-API and is passed between the GCS-API and the application using the **gcs_session_context_t** datatype
- *CC* argument to several GCS-API functions.
  This is opaque to the caller and is passed between the GCS-API and the application using the **gcs_cc_t** datatype. The design of the interface does not preclude a hardware implementation. The implementation defines whether the CC is held entirely within the CSF or outside the CSF. The contents must be protected against modification, any key values contained therein will generally also be confidentiality protected.

## 3.4 Contexts

The **gcs_cc_t** data type contains a caller-opaque cryptographic context defined by the implementation. The cryptographic context holds the algorithm context and key context information.

## 3.5 Session Context Parameters

The **gcs_session_context_t** data type contains a caller-opaque set of session context parameters which may be required by the implementation. These are set by a call to *gcs_initialise_session*. One example of their use is to include identification and authorisation information relating to the caller of the CSF.

## 3.6 Status Values

One or more status codes are returned by each GCS-API routine. Two distinct sorts of status code are returned. These are termed GCS status codes and minor status codes. An implementation of GCS functions shall return GCS_S_COMPLETE and other status values appropriate for the implementation of the function. The characteristics of a particular implementation may make some status returns inappropriate for that implementation. For example, status codes reflecting a hardware failure are inappropriate for a purely software implementation.

### 3.6.1 GCS Status Codes

GCS-API routines return GCS status codes as their **OM_uint32** function value. These codes indicate major status errors that are independent of the underlying mechanism used to provide the security service.

A GCS status code can indicate a single fatal generic API error from the routine and a single calling error. In addition, supplementary status information may be indicated by setting bits in a **Supplementary Info** field in a GCS status code.

These errors are encoded into the 32-bit GCS status code as follows:

```
     MSB                                                          LSB
     |------------------------------------------------------------|
     | Calling Error | Routine Error  |    Supplementary Info     |
     |------------------------------------------------------------|
 Bit 31           24 23            16 15                         0
```

Hence if a GCS-API routine returns a GCS status code whose upper 16 bits contain a non-zero value, the call failed. If the **Calling Error** field is non-zero, the invoking application's call of the routine was erroneous. Calling errors are defined in Table 3-1. If the **Routine Error** field is non-zero, the routine failed for one of the routine-specific reasons listed in Table 3-2 on page 29. Whether or not the upper 16 bits indicate a failure or a success, the routine may indicate additional information by setting bits in the **Supplementary Info** field of the status code. This specification does not currently define any supplementary information but it is included to accommodate a possible future expansion in scope that might require such information.

| Name | Value in Field | Meaning |
|---|---|---|
| [GCS_S_CALL_INACCESSIBLE_READ] | 1 | A required input argument cannot be read. |
| [GCS_S_CALL_INACCESSIBLE_WRITE] | 2 | A required output argument cannot be written. |
| [GCS_S_CALL_BAD_STRUCTURE] | 3 | An argument is malformed. |

**Table 3-1** Calling Errors

**Table 3-2** Routine Errors

| Name | Value in Field | Meaning |
|---|---|---|
| [GCS_S_COMPLETE] | 0 | Successful completion. |
| [GCS_S_COMPLETE_QCF] | 1 | Successful completion; supplied CC has quasi-compromised flag set. |
| [GCS_S_CONTINUE_NEEDED] | 2 | The routine must be called again to complete its function. See individual function descriptions in Chapter 4 and Chapter 10 for a detailed description. |
| [GCS_S_FAILURE] | 3 | Miscellaneous failure (see text in function descriptions). |
| [GCS_S_AUTHORISATION_FAILURE] | 4 | Authorisation failure. |
| [GCS_S_BAD_FLAG] | 5 | The flag supplied is not valid. |
| [GCS_S_BAD_SIZE] | 6 | The input buffer size exceeds the maximum that can be handled by implementation |
| [GCS_S_BUFFER_OVERFLOW] | 8 | The output buffer could have overflowed. |
| [GCS_S_BAD_CC] | 9 | The crypto context supplied is invalid |
| [GCS_S_BAD_SUBJECT_CC] | 10 | Subject CC supplied is invalid. |
| [GCS_S_BAD_AC] | 11 | Invalid algorithm context supplied. |
| [GCS_S_BAD_KC] | 12 | Invalid key context supplied. |
| [GCS_S_BAD_KGK_CC] | 13 | Key generating key CC supplied is invalid. |
| [GCS_S_BAD_KEK_CC] | 14 | Key encrypting key CC supplied is invalid. |
| [GCS_S_BAD_ARCHIVE_CC] | 15 | The KEK supplied in the CC is invalid. |
| [GCS_S_BAD_DEVICE] | 16 | The specified device is unknown. |
| [GCS_S_BAD_PART] | 17 | Invalid key part specified. |
| [GCS_S_BAD_KEY_USAGE] | 18 | The key usage specified is not valid. |
| [GCS_S_INCORRECT_KEY_STATE] | 19 | Operation not permitted for key state supplied. |
| [GCS_S_BAD_TPG] | 20 | Invalid test pattern generator specified. |
| [GCS_S_BAD_EXPORT_DATA] | 21 | Export data unit specified is not valid. |
| [GCS_S_BAD_PROTOCOL] | 22 | Invalid protocol supplied. |
| [GCS_S_BAD_PARAMETER] | 23 | Invalid parameter name. |
| [GCS_S_BAD_PARAM_VALUE] | 24 | Invalid parameter value. |
| [GCS_S_BAD_REASON] | 25 | Reason for revocation not valid. |
| [GCS_S_BAD_EXPORT_MECH] | 26 | Specified export mechanism is not valid or is not specified as permitted export mechanism in supplied CC. |
| [GCS_S_RNG_NOT_INITIALISED] | 27 | The random number generator has not been initialised. |
| [GCS_S_BAD_SUBJECT_CONTAINER] | 28 | The subject container supplied is not valid |
| [GCS_S_INVALID_REFERENCE] | 29 | The CC reference supplied does not refer to a valid crypto context. |

| Name | Value in Field | Meaning |
|---|---|---|
| [GCS_S_BAD_ARCHIVE_STRING] | 30 | The bit string supplied could not be used to restore a CC. |
| [GCS_S_BAD_IV] | 31 | Invalid initialisation vector supplied |
| [GCS_S_BAD_SESSION_CONTEXT] | 32 | session context supplied is not valid |
| [GCS_S_CONFIDENTIALITY_FLAG] | 33 | The confidentiality flag is not set to YES |
| [GCS_S_BAD_DOMAIN_ID] | 34 | the CC domain id supplied is not valid |
| [GCS_S_BAD_CC_NAME] | 35 | the CC name supplied is not valid |
| [GCS_S_DEVICE_BUSY] | 36 | The specified device is busy. |
| [GCS_S_NO_CHECK] | 37 | The checkvalue is not verified. |
| [GCS_S_NO_VERIFY] | 38 | The key cannot be verified. |
| [GCS_S_BAD_CC_LIST] | 39 | List of cryptographic contexts supplied is not valid. |
| [GCS_S_CC_LOCKED] | 40 | The cryptographic context requested is locked. |
| [GCS_S_INVALID_STATE_TRANSITION] | 41 | Key state transition requested is not permitted |
| [GCS_S_IV_REQUIRED] | 42 | An initialisation vector is required but has not been supplied |

The function specifications also use the name [GCS_S_COMPLETE], which is a zero value, to indicate an absence of any API errors or supplementary information bits.

Table 3-2 on page 29 includes the error codes applicable to both the Basic GCS-API and the Advanced GCS-API.

All [GCS_S_*] symbols equate to complete **OM_uint32** status codes, rather than to bit-field values. For example, the actual value of the symbol [GCS_S_BAD_SIZE] (value 3 in the **Routine Error** field) is 3 << 16.

The macros:

GCS_CALLING_ERROR( )
GCS_ROUTINE_ERROR( )
GCS_SUPPLEMENTARY_INFO( )

are provided, each of which takes a GCS status code and removes all but the relevant field. For example, the value obtained by applying GCS_ROUTINE_ERROR( ) to a status code removes the **Calling Errors** and **Supplementary Info** fields, leaving only the **Routine Errors** field. The values delivered by these macros may be directly compared with a [GCS_S_*] symbol of the appropriate type. The macro GCS_ERROR( ) is also provided, which when applied to a GCS status code returns a non-zero value if the status code indicates a calling or routine error, and a zero value otherwise.

A GCS-API implementation may choose to signal calling errors in a platform-specific manner instead of, or in addition to the routine value; routine errors and supplementary information should be returned by means of routine status values only.

### 3.6.2 Minor Status Codes

GCS-API C-language functions return a *minor_status* argument, which is used to indicate specialised errors from the underlying security mechanism. This argument may contain a single mechanism-specific error, indicated by an **OM_uint32** value.

The *minor_status* argument is always set by a GCS-API function, even if it returns a calling error or one of the generic API errors indicated above as fatal, although other output arguments may remain unset in such cases. However, output arguments that are expected to return pointers to storage allocated by a function must always be set by the function, even in the event of an error, although in such cases the GCS-API function may elect to set the returned argument value to NULL to indicate that no storage was actually allocated. Any length field associated with such pointers (as in a **gcs_buffer_desc** structure) should also be set to zero in such cases.

The GCS status code [GCS_S_FAILURE] is used to indicate that the underlying mechanism detected an error for which no specific GCS status code is defined. The minor status code provides more details about the error.

## 3.7 Optional Arguments

Various arguments are described as optional. This means that they follow a convention whereby a default value may be requested. The following conventions are used for omitted arguments. These conventions apply only to those arguments that are explicitly documented as optional.

### 3.7.1 gcs_buffer_t Types

Specify GCS_C_NO_BUFFER as a value. For an input argument this signifies that default behaviour is requested, while for an input,output argument it indicates that the information that would be returned by the argument is not required by the application.

### 3.7.2 Integer Types

Individual argument documentation lists values to be used to indicate default actions. These are passed by value.

### 3.7.3 Pointer Types

Specify NULL as the value.

## 3.8 Constants

The tables below set out the constants defined by the specification, and the value to which they are set.

| Name | Value | Meaning |
|---|---|---|
| [GCS_C_TRUE] | 1 | True |
| [GCS_C_FALSE] | 0 | False |
| [GCS_C_NULL] | NULL | Null |
| [GCS_C_EMPTY_BUFFER] | NULL | Empty buffer |
| [GCS_C_NO_BUFFER] | NULL | No buffer is supplied or returned |
| [GCS_C_NO_BIT_STRING] | NULL | The bit string supplied or returned is null |

**Table 3-3** Optional Parameter Constants

### 3.8.1 Algorithm Independent CC Names

The default set of algorithm independent CC names is:

| CC Name | Meaning |
|---|---|
| LOCAL_SYM_ENCIPHER_DECIPHER | Default symmetric encipher/decipher CC |
| LOCAL_ASYM_ENCIPHER | Default asymmetric encipher CC |
| LOCAL_ASYM_DECIPHER | Default asymmetric decipher CC |
| LOCAL_SIGN | Default signature generate CC |
| LOCAL_VERIFY | Default signature verify CC |
| LOCAL_HASH | Default hash CC |
| LOCAL_EXPORT | Default export key CC |
| LOCAL_IMPORT | Default import key CC |

**Table 3-4** Algorithm Independent CC Names

### 3.8.2 Chain Flag

The chain flag can take on one of several values as illustrated below.

| Chain Flag | Value | Meaning |
|---|---|---|
| GCS_FIRST | 1 | if set, indicates the first of several input buffers |
| GCS_MIDDLE | 2 | if set, indicates the second, or subsequent input buffer, but not the last |
| GCS_LAST | 3 | If set, indicates the last of several input buffers |
| GCS_ONLY | 4 | If set, indicates only one buffer is input |

**Table 3-5** Chain Flag Values

### 3.8.3 Storage Unit Classes

The following constants are defined for use as the storage unit class component in a *CC_reference* in a call to *gcs_store_cc*( )

| Storage Unit Class | Value | Meaning |
|---|---|---|
| GCS_C_DISK | 1 | Disk storage unit class |
| GCS_C_MEMORY | 2 | Memory storage unit class |
| GCS_C_CDROM | 3 | CD-ROM storage unit class |
| GCS_C_SMARTCARD | 4 | Smart Card storage unit class |

**Table 3-6** Storage Unit Class

### 3.8.4 CSF Parameters

The following constants define the names of the parameters that may be retrieved using *gcs_get_csf_params*( ).

| Parameter Name | Value | Meaning |
|---|---|---|
| GCS_C_MAX_BUFFER_SIZE | 0 | Maximum buffer size supported |
| GCS_C_IMPLEMENTATION_TYPE | 1 | Type of implementation |

**Table 3-7** CSF Parameters

### 3.8.5 CSF Implementation Type

The following constants are defined for the implementation types that may be returned by *gcs_get_csf_params*( ).

| Implementation Type | Value | Meaning |
|---|---|---|
| GCS_C_UNKNOWN | 0 | The implementation cannot return type |
| GCS_C_HARDWARE | 1 | Hardware implementation |
| GCS_C_SOFTWARE | 2 | Software implementation |
| GCS_C_BOTH | 3 | Mixed hardware and software implementation |

**Table 3-8** CSF Implementation Types

Chapter 4

# *Basic CSF Application Program Interface (API)*

This chapter presents the functions that comprise the basic GCS-API.

In the majority of these definitions a cryptographic context is included as an input parameter providing information on the algorithm(s) and key(s) to be used in the function. A cryptographic context is also included as an output parameter because the CC may be modified by the call, eg., usage counts and key states may be modified any time the CC is used to provide a key used within a function. The check value of the CC and the validity period of a key within the CC are checked on each use of the CC.

**NAME**

gcs_decipher_data — returns the input cipher text data as clear text

**SYNOPSIS**

```
OM_uint32 gcs_decipher_data(
    OM_uint32              *minor_status,
    gcs_session_context_t  *session_context,
    gcs_buffer_t           input_data,
    gcs_buffer_t           IV,
    OM_uint32               chain_flag,
    gcs_cc_t               *cc,
    gcs_buffer_t           intermediate_result,
    gcs_buffer_t           output_data
);
```

**DESCRIPTION**

This function transforms the input data from ciphertext, to cleartext using the given reversible cryptographic algorithm, key and related parameters specified in *cc*.

Data greater in length than the maximum buffer size supported by an implementation may be transformed by successive calls to *gcs_decipher_data*, passing *intermediate_result* from one call as input to the next call. The maximum buffer size may be determined by calling *gcs_get_csf_params*.

The lengths of the clear text and cipher text may or may not be the same.

The caller must possess the GCS_C_ENCIPHER_DECIPHER authority. If successful, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_decipher_data*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*input_data* (in)
The input cipher text data to be deciphered.

*IV* (optional, in)
The optional initialisation vector dependent upon the algorithm specified by *cc*.

*chain_flag* (in)
This argument can be set to GCS_FIRST, GCS_MIDDLE, GCS_LAST or GCS_ONLY.

*cc* (opaque,in/out)
The cryptographic context from which the algorithm, key and related parameters are taken to decipher the input data. It is returned with the key state updated as appropriate.

*intermediate_result* (in/out)
The intermediate results from the decipher calculation are returned with successive calls to *gcs_decipher_data*.

*output_data* (out)
The clear text corresponding to the cipher text input data is returned in the output buffer. If

the pointer and length within the *gcs_buffer_t* structure are GCS_NULL then the implementation allocates a buffer for the output of the ciphertext. If the pointer and length within the *gcs_buffer_t* structure are not GCS_NULL then the implementation will attempt to use the specified buffer when writing the ciphertext.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_CONTINUE_NEEDED]
*gcs_decipher_data* requires to be called again supplying the value returned in *intermediate_result* as an input parameter.

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_COMPLETE_QCF]
Successful completion but cc has quasi compromised flag set in key context.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BUFFER_OVERFLOW]
The input buffer length exceeds the maximum buffer size supported by the implementation or the output buffer has overflowed.

[GCS_S_BAD_SUBJECT_CC]
The cryptographic context supplied is not valid.

[GCS_S_INCORRECT_KEY_STATE]
The key state in the CC supplied does not permit the requested action, ie., key state must be active or quiescent.

[GCS_S_IV_REQUIRED]
An initialisation vector is required but has not been supplied.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required authority or some other authorisation failure has occurred.

[GCS_S_BAD_FLAG]
The chaining flag specified is not valid.

**ERRORS**

No other errors are defined.

**NAME**

gcs_decipher_verify — decipher input data and verify check value

**SYNOPSIS**

```
OM_uint32 gcs_decipher_verify(
    OM_uint32              *minor_status,
    gcs_session_context_t  *session_context,
    gcs_buffer_t           input_data,
    gcs_buffer_t           IV,
    gcs_buffer_t           check_value,
    OM_uint32               chain_flag,
    gcs_cc_t               *confidentiality_cc,
    gcs_cc_t               *integrity_cc,
    gcs_buffer_t           intermediate_result,
    gcs_buffer_t           output_data
);
```

**DESCRIPTION**

This function transforms the cipher text into cleartext, using the reversible cryptographic algorithm, key and related parameters as specified in *confidentiality_cc* and the optional *IV*. It simultaneously verifies the check value against that derived from the cleartext derived from *input_data* and may authenticate the origin of a set of data, ie., prove the knowledge of the key used to generate the check value.

Data greater in length than the maximum buffer size supported by an implementation may be transformed by successive calls to *gcs_decipher_verify*, passing *intermediate_result* from one call as input to the next call. The maximum buffer size may be determined by calling *gcs_csf_params*.

The lengths of the clear text and cipher text may or may not be the same.

The caller must possess the GCS_C_ENCIPHER_DECIPHER authority. If successful, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_decipher_verify*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*input_data* (in)
The input cipher text data to be deciphered.

*IV* (optional,in)
The optional initialisation vector dependent upon the algorithm specified in *cc*. The IV block of random data is there to make each message unique. It can also be used as a confounder.

*check_value* (in)
The check value to be verified.

*chain_flag* (in)
This argument can be set to GCS_FIRST, GCS_MIDDLE, GCS_LAST or GCS_ONLY.

*cc* (opaque,in/out)
The cryptographic context supplied, from which the algorithm, key and related parameters are taken to decipher the data input. The cryptographic context is returned, with key state updated as appropriate.

*intermediate_result* (in/out)
The intermediate results from the decipher calculation are returned with successive calls to *gcs_decipher_verify*.

*output_data* (out)
The deciphered data output from the function. If the pointer and length within the *gcs_buffer_t* structure are GCS_NULL then the implementation allocates a buffer for the output of the ciphertext. If the pointer and length within the *gcs_buffer_t* structure are not GCS_NULL then the implementation will attempt to use the specified buffer when writing the clear text.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_CONTINUE_NEEDED]
*gcs_decipher_verify* requires to be called again supplying the value returned in *intermediate_result* as an input parameter.

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_COMPLETE_QCF]
Successful completion but CC has quasi compromised flag set in key context.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BUFFER_OVERFLOW]
The input buffer length exceeds the maximum buffer size supported by the implementation.

[GCS_S_INCORRECT_KEY_STATE]
The key state in the CC supplied does not permit the requested action. ie., the key state must be active.

[GCS_S_BAD_SUBJECT_CC]
The subject cryptographic context supplied is not valid.

[GCS_S_IV_REQUIRED]
An initialisation vector is required and has not been supplied.

[GCS_S_NO_CHECK]
The check value input does not compare with that computed using the input data and the specified CC.

[GCS_S_BAD_SIZE]
The input data exceeds MAXSIZE in length.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

## NAME

gcs_delete_cc — delete a cryptographic context

## SYNOPSIS

```
OM_uint32 gcs_delete_cc(
    OM_uint32              *minor_status,
    gcs_session_context_t  *session_context,
    gcs_cc_t               *subject_cc
);
```

## DESCRIPTION

This function deletes the caller's copy of the cryptographic context referred to by *subject_cc* frees the memory allocated to it and sets the *subject_cc* pointer to NULL.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_delete_cc*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this context are required to support uses such as continuous I&A and authorisation.

*subject_cc* (opaque,in, out)
: The cryptographic context to be deleted.

## RETURN VALUE

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
: Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
: The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
: The cryptographic context supplied is not a valid context.

[GCS_S_FAILURE]
: An implementation specific error or failure has occurred.

## ERRORS

No other errors are defined.

## NAME

gcs_derive_key — derive a protected secret key or a public and private key pair

## SYNOPSIS

```
OM_uint32 gcs_derive_key(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    gcs_bit_string_t             input_string,
    gcs_cc_t                     *kgk_cc,
    gcs_cc_t                     *subject_cc
);
```

## DESCRIPTION

This function derives a secret key or a public and private key pair from *input_string*.

The algorithm, key size, key usage and other parameters associated with the cryptographic context are specified in *subject_cc*.

The derived key will be protected and the cryptographic context header flag is set appropriately (i.e., context_confidentiality is set to YES.)

The key is output within the key context part of *subject_cc*. The caller must possess the GCS_C_SELECTION GCS authority or the call will fail.

If successful, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_derive_key*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*input_string* (in)
: The input string used as the basis for deriving a secret key or a public and private key pair and interpreted per spawn method indicated in *kgk_cc*.

*kgk_cc* (optional, in/out)
: When supplied this references the cryptographic context used to derive a key using the derivation mechanism specified in the algorithm context of *kgk_cc*.

*subject_cc* (opaque,in/out)
: The subject_cc cryptographic context supplied is populated to include the secret key or public and private key pair derived by *gcs_derive_key* and returned.

## RETURN VALUE

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
: Successful completion.

[GCS_S_COMPLETE_QCF]
: Successful completion but CC has quasi-compromised flag set in key context.

[GCS_S_BAD_SESSION_CONTEXT]
: The session context supplied is not valid.

[GCS_S_BAD_KGK_CC]
The key generating key cryptographic context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
The subject cryptographic context supplied is not valid.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required GCS authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_encipher_data — transform the input data to ciphertext

**SYNOPSIS**

```
OM_uint32 gcs_encipher_data(
    OM_uint32              *minor_status,
    gcs_session_context_t  *session_context,
    gcs_buffer_t           input_data,
    gcs_buffer_t           IV,
    OM_uint32               chain_flag,
    gcs_cc_t               *cc,
    gcs_buffer_t           intermediate_result,
    gcs_buffer_t           output_data
);
```

**DESCRIPTION**

This function transforms the clear text input data into cipher text, using the reversible cryptographic algorithm, key and related parameters as specified in *cc*.

Data greater in length than the maximum buffer size supported by an implementation may be transformed by successive calls to *gcs_encipher_data*, passing *intermediate_result* from one call as input to the next call. The maximum buffer size may be determined by calling *gcs_get_csf_params*.

The lengths of the clear text and cipher text may or may not be the same.

The caller must possess the GCS_C_ENCIPHER_DECIPHER authority. If successful, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_encipher_data*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*input_data* (in)
: The input clear text data to be enciphered.

*IV* (optional,in)
: The optional initialisation vector dependent upon the algorithm specified in *cc*. The IV block of random data is there to make each message unique. It can also be used as a confounder.

*chain_flag* (in)
: This argument can be set to GCS_FIRST, GCS_MIDDLE, GCS_LAST or GCS_ONLY.

*cc* (opaque,in/out)
: The cryptographic context supplied, from which the algorithm, key and related parameters are taken to encipher the data input. The cryptographic context is returned, with key state updated as appropriate.

*intermediate_result* (in/out)
: The intermediate results from the encipher calculation are returned with successive calls to *gcs_encipher_data*.

*output_data* (out)
The enciphered data output from the function. If the pointer and length within the *gcs_buffer_t* structure are GCS_NULL then the implementation allocates a buffer for the output of the ciphertext. If the pointer and length within the *gcs_buffer_t* structure are not GCS_NULL then the implementation will attempt to use the specified buffer when writing the ciphertext.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_CONTINUE_NEEDED]
*gcs_encipher_data* requires to be called again supplying the value returned in *intermediate_result* as an input parameter.

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_COMPLETE_QCF]
Successful completion but CC has quasi compromised flag set in key context.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BUFFER_OVERFLOW]
The input buffer length exceeds the maximum buffer size supported by the implementation.

[GCS_S_INCORRECT_KEY_STATE]
The key state in the CC supplied does not permit the requested action. ie., the key state must be active.

[GCS_S_BAD_SUBJECT_CC]
The subject cryptographic context supplied is not valid.

[GCS_S_IV_REQUIRED]
An initialisation vector is required and has not been supplied.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_export_key — transform a key into a protected form for export

**SYNOPSIS**

```
OM_uint32 gcs_export_key(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    gcs_cc_t                     *subject_cc,
    gcs_cc_t                     *kek_cc,
    gcs_bit_string_t             export_data
);
```

**DESCRIPTION**

The *gcs_export_key* function transforms a key and associated information, contained within or referenced by *subject_cc*, into an exchangeable protected form using a key enciphering key, contained within or referenced by *kek_cc*. This service returns a mechanism specific token (*export_data*) including the transformed key.

If *subject_cc* contains a private and public key pair, the *gcs_export_key* function only returns the public key.

This service is provided to support key distribution services. The caller must possess the GCS_C_KEY_USAGE GCS authority or the function will fail.

If successful, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_export_key*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*subject_cc* (opaque,in/out)
: The cryptographic context containing the key to be exported. The key context of *subject_cc* may be updated by the call to this function.

*kek_cc* (opaque,in/out)
: The key enciphering key used to encipher the key and associated information contained in *subject_cc*.

*export_data* (in/out)
: The partial protocol data unit, a mechanism-specific structure which reflects the *protocol_type* containing protocol specific information. On return, it includes the enciphered and protected key for export.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
: Successful completion.

[GCS_S_COMPLETE_QCF]
: Successful completion but *subject_cc* or *kek_cc* has quasi compromised flag set in key context.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid, ie., is revoked or has been deactivated.

[GCS_S_BAD_SUBJECT_CC]
The *subject_cc* supplied is not valid.

[GCS_S_BAD_KEK_CC]
The *kek_cc* supplied is not valid.

[GCS_S_BAD_EXPORT_MECH]
The export_mechanism specified in *subject_cc* is inconsistent with the contents of kek_cc.

[GCS_S_BAD_EXPORT_DATA]
The export data supplied is not valid.

[GCS_S_INCORRECT_KEY_STATE]
The key state of the kek_cc does not permit the requested action.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required GCS authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_generate_check_value — return the check value of the input data

**SYNOPSIS**

```
OM_uint32 gcs_generate_check_value(
    OM_uint32                   *minor_status,
    gcs_session_context_t       *session_context,
    gcs_buffer_t                input_data,
    gcs_buffer_t                IV,
    OM_uint32                    chain_flag,
    gcs_cc_t                    *cc,
    gcs_buffer_t                intermediate_result,
    gcs_buffer_t                check_value
);
```

**DESCRIPTION**

This function returns the check value of the input data contained in *input_buffer* computed using the cryptographic algorithms, key and related parameters as specified by *cc* and the optional initialisation vector, *IV*. The function is used to compute a checkvalue from a data item for the purposes of integrity, or data origin authentication.

The maximum size of data that an implementation of the GCS-API can handle may be determined by a call to *gcs_get_csf_params*. Check values for data greater than the maximum size that can be handled by an implementation may be generated by successive invocations of *gcs_generate_check_value*. The contents of *intermediate_result* contain an intermediate result, if the chaining flag is set to GCS_FIRST or GCS_MIDDLE. In this case, the intermediate result is re-input as a parameter to the next call to *gcs_generate_check_value*. The *chain_flag* indicates if an invocation is the first, a middle, the last, or only invocation. The function works even if the input data is zero.

If successful, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_generate_check_value*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*input_data* (in)
: The data for which the check value is to be generated. The input data may need to be split into sections that do not exceed the maximum input data size that can be handled by an implementation and successive calls made to *gcs_generate_check_value.*

*IV* (optional, in)
: The optional initialisation vector used to generate the checkvalue.

*chain_flag* (in)
: This argument can be set to one of four values, indicating how the input data have been split for hashing. Note that data can only be chained if the cryptographic algorithm in the *CC* supplied permits it. The *chain_flag* can be set to GCS_FIRST, GCS_MIDDLE, GCS_LAST or GCS_ONLY.

*cc* (opaque,in /out)
The cryptographic context used to generate the check value on the input data. The cryptographic context is returned with key states updated as appropriate.

*intermediate_result* (in /out)
If *chain_flag* is set to GCS_FIRST, or GCS_MIDDLE, the intermediate results from the checkvalue calculation are returned in this parameter. This needs to be input to the next call to *gcs_generate_check_value*( ).

*check_value* (out)
If *chaining_flag* is set to either GCS_LAST or GCS_ONLY, then a call to *gcs_generate_check_value*( ) returns the check value in *check_value*.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_COMPLETE_QCF]
Successful completion but the quasi compromised flag is set in the key context of *cc*.

[GCS_S_CONTINUE_NEEDED]
Another call to the function is required.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
The subject CC supplied is not valid.

[GCS_S_BAD_SIZE]
The input buffer size exceeds maximum size that can be handled by the implementation.

[GCS_S_BAD_FLAG]
The chaining flag specified is not valid.

[GCS_S_INCORRECT_KEY_STATE]
The key state in the CC supplied does not permit the requested action.

**ERRORS**

No other errors are defined.

**NAME**

gcs_generate_hash — irreversibly hash input data

**SYNOPSIS**

```
OM_uint32 gcs_generate_hash(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    gcs_buffer_t                  input_data,
    gcs_cc_t                     *cc,
    gcs_buffer_t                  intermediate_result,
    OM_uint32                     chain_flag,
    gcs_buffer_t                  output_data
);
```

**DESCRIPTION**

This function takes the *input_buffer* and hashes it according to the non-keyed cryptographic context defined by *cc*. The maximum size of data that an implementation of the GCS-API can handle may be determined by a call to *gcs_get_csf_params*. Hash values for data greater than the maximum size that can be handled by an implementation may be generated by successive invocations of *gcs_generate_hash*. The contents of *intermediate_result* contain an intermediate result, if the chaining flag is set to GCS_FIRST or GCS_MIDDLE. In this case, the intermediate result is re-input as a parameter to the next call to *gcs_generate_hash*. The *chaining_flag* indicates if an invocation is the first, a middle, the last, or only invocation. The function should still succeed even if the input data length is zero.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_generate_hash*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*input_data* (in)
: The input data containing the data to be hashed. This must not exceed the maximum size that can be handled by the implementation.

*cc* (opaque, in/out)
: The cryptographic context which includes the non-keyed algorithm context for the hash.

*intermediate_result* (in/out)
: When the *chain_flag* is set to GCS_MIDDLE or GCS_LAST, the caller returns the last *intermediate_result* returned from the function as the *intermediate_result* for the next call to the function.

*chain_flag* (in)
: This argument can be set to one of four values, GCS_FIRST, GCS_MIDDLE, GCS_LAST, and GCS_ONLY, indicating how the input data have been split for hashing.

*output_buffer* (out)
: The results of the hashing are returned in the output buffer when the *chaining_flag* is set to GCS_LAST or GCS_ONLY.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_CONTINUE_NEEDED]
Another call to the function is required.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
The cryptographic context supplied is not valid, ie., does not contain a suitable hash algorithm.

[GCS_S_BAD_FLAG]
The value supplied in the chaining flag is not valid.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred. No cryptographic mechanisms are specified.

[GCS_S_BAD_SIZE]
The size of the input buffer exceeds the size that can be handled by the implementation.

[GCS_S_BUFFER_OVERFLOW]
The output buffer has overflowed.

**ERRORS**

No other errors are defined.

## NAME

gcs_generate_key — generate a secret key, or a public and private key pair

## SYNOPSIS

```
OM_uint32 gcs_generate_key(
    OM_uint32                   *minor_status,
    gcs_session_context_t       *session_context,
    gcs_cc_t                    *cc
);
```

## DESCRIPTION

This function generates a secret key or public and private key pair and populates the *cc*. The algorithm, key size, key usage and other parameters associated with the cryptographic context are specified in the *cc* supplied.

The generated key is protected. The function will fail if the context_confidentiality flag is not set to YES. The caller must possess the GCS_C_SELECTION GCS authority.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_generate_key*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*cc* (opaque,in/out)
The cryptographic context supplied should include the *algorithm_context* and the *key_data* without the key bits. The populated cryptographic context is returned, including the secret key or the public and private key pair generated by *gcs_generate_key*.

## RETURN VALUE

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
The cryptographic context supplied is not valid.

[GCS_S_RNG_NOT_INITIALISED]
The CSF pseudo-random number generator has not been initialised.

[GCS_S_CONFIDENTIALITY_FLAG]
The confidentiality flag is not set to YES, ie., the CC is intended for clear key.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required GCS authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

## NAME

gcs_generate_random_number — return a cryptographically strong random number

## SYNOPSIS

```
OM_uint32 gcs_generate_random_number(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    OM_uint32                     size,
    gcs_bit_string_t              random_number
);
```

## DESCRIPTION

This function generates a cryptographically strong random number *size* bits in length and returns it in *random_number*. If successful, the function returns [GCS_S_COMPLETE].

A cryptographically strong number is one that does not have a period, is random, and might repeat. The arguments for *gcs_generate_random_number*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*size* (in)
The length in bits of the random number generated.

*random_number* (out)
The generated random number bit string

## RETURN VALUE

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_RNG_NOT_INITIALISED]
The CSF pseudo-random number generator has not been initialised.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

## ERRORS

No other errors are defined.

## NAME

gcs_get_csf_params — get csf parameters

## SYNOPSIS

```
OM_uint32 gcs_get_csf_params(
    OM_uint32              *minor_status,
    gcs_session_context_t  *session_context,
    gcs_cc_t               *subject_cc,
    OM_uint32              *parameter_name,
    OM_uint32              *parameter_integer_value,
    gcs_bit_string_t        parameter_bit_string
 );
```

## DESCRIPTION

This function returns the CSF parameters for the algorithm specified in *subject_cc*. Two parameters are defined by the specification, the MAX_BUFFER_SIZE and the IMPLEMENTATION_TYPE. Other parameters may be defined by the implementation. MAX_BUFFER_SIZE allows a caller to partition large files into blocks of manageable size for subsequent cryptographic functions.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_get_csf_params*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this context are required to support uses such as continuous I&A and authorisation.

*subject_cc* (opaque,in /out)
The cryptographic context containing the algorithm queried.

*parameter_name* (in)
The name of the parameter. The GCS-API currently defines GCS_C_GET_MAX_BUFFER_SIZE and GCS_C_IMPLEMENTATION_TYPE .

*parameter_integer_value* (out) CSF parameter integer values.

*parameter_bit_string* (out)
CSF bit string parameters.

## RETURN VALUE

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
The *subject_cc* supplied is not valid.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_import_key — transform a key into an operational form for import

**SYNOPSIS**

```
OM_uint32 gcs_import_key(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    gcs_cc_t                     *kek_cc,
    gcs_bit_string_t              export_data,
    gcs_cc_t                     *subject_cc
);
```

**DESCRIPTION**

The *gcs_import_key* function transforms a key and associated information contained within the *export_data* into an operational format key contained within or referenced by *subject_cc*.

The *export_data* is of an exchangeable protected format as produced by the *gcs_export_key* service. *kek_cc* references the key encrypting key under which the imported key is protected and *kek_cc* specifies the key distribution protocol being used.

This service is provided to support key distribution services. The caller must possess the GCS_C_KEY_USAGE GCS authority.

If successful, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_import_key*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*kek_cc* (opaque,in)
: The cc containing the key enciphering key under which the imported key is protected.

*export_data* (in)
: The input protocol data unit in protected exchangeable format as created by *gcs_export_key*.

*subject_CC* (opaque,in/out)
: The cryptographic context supplied, if required for the specified *export_mech*, which is to be populated with the imported key and any associated information. The key in its operational format is returned in *subject_cc*. The *subject_cc* provides the defaults for key control parameters such as key usage, initial key state, key validity periods, etc.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
: Successful completion.

[GCS_S_COMPLETE_QCF]
: Successful completion but *subject_cc* or *kek_cc* has quasi compromised flag set in key context.

[GCS_S_BAD_SESSION_CONTEXT]
: The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
The cryptographic context supplied is not valid.

[GCS_S_BAD_KEK_CC]
The kek_cc supplied is not valid.

[GCS_S_BAD_EXPORT_MECH]
The export_mechanism specified in subject_cc is inconsistent with the contents of kek_cc.

[GCS_S_BAD_PDU]
The partial protocol data unit supplied is not valid.

[GCS_S_INCORRECT_KEY_STATE]
The key state of kek_cc does not permit the requested action.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required GCS authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_initialise_session — initialise a session with the CSF

**SYNOPSIS**

```
OM_uint32 gcs_initialise_session(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    gcs_buffer_t                  initialise_parameters
);
```

**DESCRIPTION**

This function initialises a session between the caller and the CSF. It may be used to authenticate a caller and establish the context for the session between the caller and the CSF, including authorisations for the use of CSF functions and defaults that are individual to the caller, or the principal the caller represents.

To complete initialisation then a sequence of calls to *gcs_initialise_session* may be required. In this case the function returns [GCS_S_CONTINUE].

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_initialise_session*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque, in/out)
An implementation specific parameter that defines the context of the current session between the caller and the CSF. It is used as an input parameter to all other CSF functions to support continuous I&A and authorisation services. If *gcs_initialise_session*( ) returns GCS_S_CONTINUE then the partially completed *session_context* is reinput to the next call to *gcs_initialise_session*( ).

*initialise_parameters* (opaque,in)
The set of implementation defined parameters required to initialise a session with the CSF.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_CONTINUE]
A further call to *gcs_initialise_session*( ) is required.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
An authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_key_agreement — initialise a key agreement exchange

**SYNOPSIS**

```
OM_uint32 gcs_key_agreement(
    OM_uint32                  *minor_status,
    gcs_session_context_t      *session_context,
    gcs_cc_t                   *caller_cc,
    gcs_cc_t                   *other_cc,
    gcs_bit_string_t            pdu_in,
    gcs_bit_string_t            pdu_out,
    gcs_cc_t                    *kak_cc
);
```

**DESCRIPTION**

This function initiates the transformation of a key agreement and associated information between the application and a remote peer. The key agreement is completed by exchanging the *pdu_out* and *pdu_in* with the remote peer and making to one or more subsequent calls to *gcs_key_agreement*( ).

The key agreement information is contained within or referenced by *kak_cc*. To complete the exchange of the key agreement, the *pdu_out* output from this function is sent as an opaque data item to the remote peer and the *pdu_in* is imported from the remote peer.

It returns a *kak_cc* in which the key agreement is built up with subsequent calls to *gcs_key_agreement*.

This service is provided to support key distribution services. The caller must possess the GCS_C_KEY_USAGE GCS authority or the call will fail.

If successful, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_key_agreement*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*caller_cc* (optional,opaque,in)
The *caller_cc* provides the private key of the caller. If not specified, the private key defaults to that established by the call to *gcs_initialise_session*( ) that established the current CSF session.

*other_cc* (opaque,in)
The *other_cc* provides the public key of the other party in the exchange.

*pdu_in* (in)
The partial protocol data unit sent from the remote peer. On the first call, *pdu_in* is a NULL pointer.

*pdu_out* (out)
The partial protocol data unit to be sent to the remote peer. This is an export mechanism-specific structure.

*kak_cc* (opaque,out)
The *kak_cc* maintains the intermediate state between subsequent calls to *gcs_key_agreement* and returns the enciphered and protected key agreement.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_COMPLETE_QCF]
Successful completion but *subject_cc* has the quasi compromised flag set in its key context.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_CONTINUE_NEEDED]
Subsequent call to *gcs_import_key_agreement* is needed.

[GCS_S_BAD_SUBJECT_CC]
One or more of *kak_cc*, *caller-cc*, or *another_cc* is not valid.

[GCS_S_BAD_EXPORT_MECH]
The export mechanism specified in *kak_cc* is not valid.

[GCS_S_BAD_PROTOCOL]
The partial_pdu_to_send supplied is not valid.

[GCS_S_INCORRECT_KEY_STATE]
The key state of one or more of *kak_cc*, or *caller_cc*, or *other_cc* does not permit the requested action.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required GCS authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

## NAME

gcs_list_cc — list crypto contexts stored in CSF

## SYNOPSIS

```
OM_uint32 gcs_list_cc(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    OM_unit32                     index_in_cc_list,
    gcs_buffer_t                  domain_id,
    gcs_buffer_t                  cc_name,
    gcs_cc_ref_t                 *cc_reference
);
```

## DESCRIPTION

This function returns a *cc_reference*, a *cc_name* or a *domain_id* from the list of CCs indexed by *index_in_cc_list*. The caller is then able to retrieve the CCs by calling *gcs_retrieve_cc*( ) for each *cc_reference*, *cc_name* or *domain_id* in turn. The list of CCs indexed contains only those CCs accessible to the caller.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_list_cc*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
An implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*index_in_cc_list* (in)
The index in the list of cryptographic contexts that the caller wishes to access.

*domain_id* (out)
The domain identity of the cryptographic context corresponding to the *index_in_cc_list* supplied. The *domain_id* may be NULL.

*cc_name* (out)
The name of the CC corresponding to the *index_in_cc_list*. The *cc_name* may be NULL.

*cc_reference* (opaque,out)
The cryptographic context reference corresponding to the *index_in_cc_list* supplied. The *cc_reference* may be NULL.

## RETURN VALUE

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion. There is no other element in the list if the function returns with GCS_S_COMPLETE.

[GCS_S_CONTINUE_NEEDED]
Another call to the function is required. There are other elements in the list if the function returns with GCS_S_CONTINUE_NEEDED.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_FAIL]
There are no elements in the cc list.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

**ERRORS**

No other errors are defined.

## NAME

gcs_protect_data — encipher data and generate a check value

## SYNOPSIS

```
OM_uint32 gcs_protect_data(
    OM_uint32              *minor_status,
    gcs_session_context_t  *session_context,
    gcs_buffer_t            input_data,
    gcs_buffer_t            IV,
    OM_uint32               chain_flag,
    gcs_cc_t               *confidentiality_cc,
    gcs_cc_t               *integrity_cc,
    gcs_buffer_t            intermediate_result,
    gcs_buffer_t           *output_data,
    gcs_buffer_t            check_value
);
```

## DESCRIPTION

This function transforms the cleartext submitted as *input_data* into cipher text, using the reversible cryptographic algorithm, key and related parameters as specified in *Confidentiality_cc* and the optional initialisation vector *IV*. It returns the checkvalue of the cleartext submitted as *input_data* computed using the cryptographic algorithms, key and related parameters as specified by *integrity_cc*. The checkvalue is computed for the purposes of integrity or data origin authentication.

Data greater in length than the maximum buffer size supported by an implementation may be transformed by successive calls to *gcs_protect_data*, passing *intermediate_result* from one call as input to the next call. The maximum buffer size may be determined by calling *gcs_csf_params*.

The lengths of the clear text and cipher text may or may not be the same.

The caller must possess the GCS_C_ENCIPHER_DECIPHER authority. If successful, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_protect_data*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*input_data* (in)
: The input clear text data to be enciphered and for which the check value is required.

*IV* (optional,in)
: The optional initialisation vector dependent upon the algorithm specified in *cc*. The IV block of random data is there to make each message unique. It can also be used as a confounder.

*chain_flag* (in)
: This argument can be set to GCS_FIRST, GCS_MIDDLE, GCS_LAST or GCS_ONLY.

*cc* (opaque,in/out)
: The cryptographic context supplied, from which the algorithm, key and related parameters

are taken to encipher the data input. The cryptographic context is returned, with key state updated as appropriate.

*intermediate_result* (in/out)
The intermediate results from the encipher calculation are returned with successive calls to *gcs_encipher_data*.

*output_data* (out)
The enciphered data output from the function. If the pointer and length within the *gcs_buffer_t* structure are GCS_NULL then the implementation allocates a buffer for the output of the ciphertext. If the pointer and length within the *gcs_buffer_t* structure are not GCS_NULL then the implementation will attempt to use the specified buffer when writing the ciphertext.

*check_value* (out)
If *chain_flag* is set to either GCS_LAST or GCS_ONLY, then a call to *gcs_protect_data*( ) returns the checkvalue in *check_value*.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_CONTINUE_NEEDED]
*gcs_protect_data* requires to be called again supplying the value returned in *intermediate_result* as an input parameter.

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_COMPLETE_QCF]
Successful completion but CC has quasi compromised flag set in key context.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BUFFER_OVERFLOW]
The check value or intermediate_result buffer length exceeds the maximum buffer size supported by the implementation.

[GCS_S_INCORRECT_KEY_STATE]
The key state in the CC supplied does not permit the requested action. ie., the key state must be active.

[GCS_S_BAD_SUBJECT_CC]
The subject cryptographic context supplied is not valid.

[GCS_S_IV_REQUIRED]
An initialisation vector is required and has not been supplied.

[GCS_S_BAD_FLAG]
The chaining flag specified is not valid.

[GCS_S_INCORRECT_KEY_STATE]
The key state in the CC supplied does not permit the requested action.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_release_bit_string — free storage allocated by the CSF

**SYNOPSIS**

```
OM_uint32 gcs_release_bit_string(
    OM_uint32                          *minor_status,
    gcs_bit_string_t                   *buffer
);
```

**DESCRIPTION**

The following APIs have a gcs_bit_string_t as output parameter: *gcs_get_cc*, *gcs_generate_random*, *gcs_export_key* and *gcs_get_csf_params*. Storage of the output data is allocated by the CSF. This function frees this storage area. In addition to freeing the associated storage, the function zeros the length field in the *buffer* argument. If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_release_bit_string*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*buffer* (in,out)
: The storage associated with the *buffer* is deleted. The gcs_bit_string_t object is not freed, but its length field is zeroed.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
: Successful completion.

[GCS_S_FAILURE]
: An implementation specific error or failure has occurred.

**ERRORS**

No other errors are defined.

## NAME

gcs_release_buffer — free storage associated with a buffer

## SYNOPSIS

```
OM_uint32 gcs_release_buffer(
    OM_uint32                        *minor_status,
    gcs_buffer_t                      buffer
);
```

## DESCRIPTION

This function frees storage associated with a buffer. The storage must have been allocated by a GCS-API function. In addition to freeing the associated storage, the function zeros the length field in the *buffer* argument. If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_release_buffer*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*buffer* (in,out)
: The storage associated with the *buffer* is deleted. The gcs_buffer_t object is not freed, but its length field is zeroed.

## RETURN VALUE

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
: Successful completion.

[GCS_S_FAILURE]
: An implementation specific error or failure has occurred.

## ERRORS

No other errors are defined.

## NAME

gcs_remove_cc — removes the specified cryptographic context from the CSF

## SYNOPSIS

```
OM_uint32 gcs_remove_cc(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    gcs_buffer_t                  domain_id
    gcs_buffer_t                 *cc_name
    gcs_cc_ref_t                 *cc_reference,
);
```

## DESCRIPTION

This function removes from the CSF a cryptographic context, previously made globally referenceable within the CSF by a call to the *gcs_store_cc* function. The cryptographic context reference input, *cc_reference*, specifies where the cryptographic context is stored. The caller must possess the GCS_C_SELECTION or the GCS_C_KEY_USAGE GCS authority

To remove a populated CC, or the GCS_C_KEY_USAGE GCS authority to remove a template CC.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_remove_cc*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*cc_reference* (optional,opaque,in)
: The optional reference to the stored cryptographic context that is to be removed. This function removes the global referenceability of the CC. If NULL, *cc_name* must be specified.

*domain_id* (optional,in)
: Th optional domain identifier. This is required if *cc_reference* is not defined.

*cc_name* (optional,in)
: The optional name of the cryptographic context to be removed. This is required if *cc_reference* is not defined.

## RETURN VALUE

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
: Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
: The session context supplied is not valid.

[GCS_S_INVALID_REFERENCE]
: The cryptographic context reference supplied does not refer to a valid cryptographic context.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_INVALID_CC_NAME]
The combination of Domain_ID and CC_Name supplied do not refer to a valid cryptographic context.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required GCS authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_retrieve_cc — retrieve a copy of the cryptographic context from CSF storage

**SYNOPSIS**

```
OM_uint32 gcs_retrieve_cc(
    OM_uint32                  *minor_status,
    gcs_session_context_t      *session_context,
    gcs_buffer_t                domain_id,
    gcs_buffer_t                cc_name,
    gcs_cc_ref_t               *cc_reference,
    boolean                     exclusive_update,
    gcs_cc_t                   *retrieved_cc
);
```

**DESCRIPTION**

This function returns a cryptographic context, *retrieved_cc*, to the caller using the cryptographic context reference, *cc_reference*, provided. As an alternative to a cc_reference, a domain_ID and CC_name may be specified to identify the CC. The cryptographic context reference was previously created by a call to *gcs_store_cc*. The function is responsible for allocating memory for the retrieved cc. *gcs_delete_cc* is used to delete the caller's copy of the cc and release the memory allocation.

The value returned in the *retrieved_cc* argument is not defined unless the function returns [GCS_S_COMPLETE].

The arguments for *gcs_retrieve_cc*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*domain_id* (optional,in)
: The optional domain identifier. This is required if *cc_reference* is not defined.

*cc_name* (optional,in)
: The optional name of the cryptographic context. This is required if *cc_reference* is not defined.

*cc_reference* (optional,opaque,in)
: Reference to the cryptographic context required. This is required if the *cc_name* and *domain_id* have not been specified.

*exclusive_update* (in)
: If the caller intends to update the CC retrieved by this call and then replace the stored copy then *exclusive_update* must be set to TRUE. This sets an exclusive access lock on the stored CC and any further calls on the CSF using this CC, except by this caller for the purposes of modifying the CC, shall fail until the exclusive access lock is released by a call to *gcs_store_cc*( ).

*retrieved_cc* (opaque,out)
: Cryptographic context corresponding to cc_reference.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_INVALID_REFERENCE]
The cryptographic context reference supplied does not refer to a valid cryptographic context.

[GCS_S_BAD_CC_NAME]
The combination of Domain_ID and CC_Name supplied do not refer to a valid cryptographic context.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_CC_BUSY]
The specified device is busy.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required GCS authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_store_cc — store the cryptographic context in the CSF

**SYNOPSIS**

```
OM_uint32 gcs_store_cc(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    gcs_buffer_t                  domain_id,
    gcs_buffer_t                  cc_name,
    gcs_cc_t                     *subject_cc,
    OM_uint32                    *storage_unit_class,
    OM_uint32                    *storage_unit_instance,
    gcs_cc_ref_t                 *cc_reference
);
```

**DESCRIPTION**

This function stores the cryptographic context, *subject_cc*, within the CSF on the optional storage unit device specified by *storage_unit_class* and returns to the caller a handle, *cc_reference*, by which it may be referenced. *cc_reference* may be exchanged between clients of the CSF and used to retrieve a copy of the cryptographic context for use in subsequent function calls on this same CSF.

The caller must possess the GCS_C_SELECTION GCS authority in order to store a populated CC, and the GCS_C_KEY_USAGE GCS authority in order to store a template CC.

A template or populated CC which has been retrieved with an exclusive lock and modified is stored as the original CC. The exclusive access lock is released after a successful call to *gcs_store_cc*( ).

A populated CC retrieved without a lock and modified is stored as a new populated CC. A template CC retrieved without a lock and modified is stored as a new template CC. When storing a CC previously retrieved without a lock and if the same *domain_id* and *cc_name* combination as an existing stored CC is provided then a call to gcs_store_cc*returns*( )[GCS_S_BAD_CC_NAME].

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_store_cc*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*domain_id* (optional,in)
The optional domain identity for the cryptographic context supplied.

*cc_name* (optional,in)
The optional name of the cryptographic context supplied.

*subject_cc* (opaque,in)
The cryptographic context to be stored.

*storage_unit_class* (optional,in / out)
The optional type of device on which the cryptographic context is to be stored. This may be

defined as GCS_C_DISK, GCS_C_MEMORY, GCS_C_CDROM, or GCS_C_SMARTCARD. If GCS_NULL is specified, the default device is used.

*storage_unit_instance* (optional,in/out)
The optional name of the device on which the cryptographic context is to be stored.

*cc_reference* (opaque,in,out)
The reference generated by the CSF to the cryptographic context stored by *gcs_store_cc*( ). The cryptographic context reference includes the storage unit class as part of the reference.

If the call is restoring a stored CC previously retrieved with an exclusive lock then the CC_reference may be used as an input to the stored CC.

If the call is restoring a stored CC previously restored without an exclusive lock the a new CC is created and a new CC_reference is generated.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
The cryptographic context supplied is not valid.

[GCS_S_BAD_DEVICE]
The device specified by*storage_unit_class* is not supported.

[GCS_S_DEVICE_BUSY]
The specified device is busy.

[GCS_S_BAD_DOMAIN_ID]
The supplied CC *domain_id* is not valid.

[GCS_S_BAD_CC_NAME]
The supplied *cc_name* is not valid, ie., if the CC was retrieved without an exclusive lock, and the *cc_name* supplied equals the original *cc_name*.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required GCS authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_terminate_session — terminate a session with the CSF

**SYNOPSIS**

```
OM_uint32 gcs_terminate_session(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context
);
```

**DESCRIPTION**

This function terminates a session between the caller and the CSF. If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_terminate_session*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque, in/out)
: An implementation specific parameter that defines the context of the current session between the caller and the CSF.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
: Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
: The session context supplied is not valid.

[GCS_S_FAILURE]
: An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
: An authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_verify_check_value — verify the checkvalue given against the checkvalue derived from the input data

**SYNOPSIS**

```
OM_uint32 gcs_verify_check_value(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    gcs_buffer_t                  input_data,
    gcs_buffer_t                  IV,
    gcs_buffer_t                  check_value,
    OM_uint32                     chain_flag,
    gcs_cc_t                     *cc,
    gcs_buffer_t                  intermediate_result
);
```

**DESCRIPTION**

This function verifies the check value against that derived from the input data contained in *input_data* and may authenticate the origin of a set of data, ie., prove the knowledge of the key used to generate the check value.

A caller may determine the maximum size of input data that may be handled by an implementation in a single call to this function by calling *gcs_get_csf_params*. Check values for data greater than this maximum size may be verified by successive invocations of *gcs_verify_check_value*.

The contents of *intermediate_result* generated by the previous invocation are re-input as *intermediate_result*. The *chain_flag* indicates if an invocation is the first, a middle, the last, or only invocation.

The *intermediate_result* needs to be protected by an implementation against disclosure in order to prevent a verified check value being used in an unauthorised way to generate a check value using a symmetric key.

If the check value is verified, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_verify_check_value*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*input_data* (in)
: The data for which the checkvalue is to be verified.

*IV* (optional,in)
: The optional initialisation vector dependent upon the type of algorithm used to verify the checkvalue.

*check_value* (in)
: The check value which is to be verified.

*chain_flag* (in)
This argument can be set to one of four values, indicating how the input data have been split. The values are GCS_FIRST, GCS_MIDDLE, GCS_LAST or GCS_ONLY.

*cc* (opaque,in/out)
The cryptographic context to be used to generate a check value on the input data. If the *chaining_flag* is set to either GCS_LAST or GCS_ONLY, then the cryptographic context with keys updated as required is returned.

*intermediate_result* (in/out)
The intermediate results from the check value calculation are returned with all successive calls to *gcs_verify_check_value*.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_COMPLETE_QCF]
Successful completion but *subject_cc* has quasi compromised flag set in key context.

[GCS_S_CONTINUE_NEEDED]
Another call to the function is required.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
The cryptographic context supplied is not valid.

[GCS_S_INCORRECT_KEY_STATE]
The key state in the CC supplied does not permit the requested action.

[GCS_S_FAILURE]
An implementation specific hardware or function failure has occurred.

[GCS_S_NO_CHECK]
The checkvalue input does not compare with that computed using the input data and the specified CC.

[GCS_S_BAD_SIZE]
The input buffer size exceeds maximum size that can be handled by the implementation.

[GCS_S_BAD_FLAG]
The chain flag specified is not valid.

**ERRORS**

No other errors are defined.

Chapter 5

# Advanced GCS-API Introduction

The increasing use of network services such as the Internet has enhanced awareness of the need for security in distributed computer systems, particularly in the light of the publicity surrounding successful breaches of security, for example, the *sniffing* of user identities and passwords passed in the clear over the Internet.

Security services such as authentication of identities, data-origin authentication, non-repudiation, data separation and confidentiality and integrity protection rely on underlying cryptographic services to provide protection. However, the wide-spread and common use of cryptography within applications is hindered by two things:

- the lack of agreed application programming interfaces
- legislative constraints on use and export of the technology

It has long been recognised that a standard application programming interface specification is needed for cryptographic services and this document addresses that need.

## 5.1 Callers of Cryptographic Services

The callers of cryptographic services may be classified according to the cryptographic awareness of the caller and the relative responsibility of the caller for cryptographic security policy. This is illustrated in Figure 5-1.

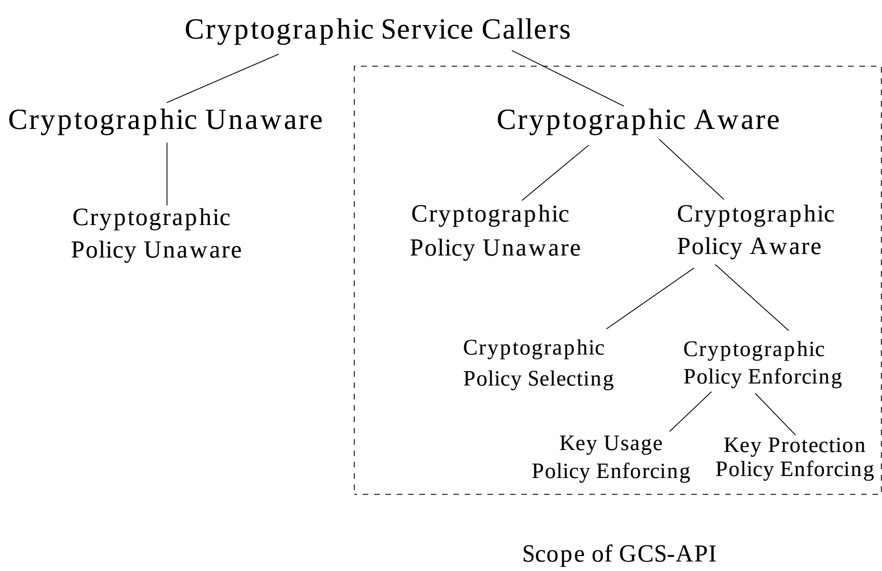


**Figure 5-1** Types of Caller of Cryptographic Services

### 5.1.1 Cryptographic Unaware Caller

Cryptographic services may be invoked on behalf of a caller that is unaware of any details of the cryptographic service. A cryptographic unaware caller invokes confidentiality or integrity protection services for an entity such as a file or message from an application infrastructure provider. The caller is unaware of how such protection is implemented, ie., the type of transform used, such as encipherment or checkvalue generation, nor the *cryptographic context* of the transform , see Chapter 7 on page 95, comprising the specific details of the cryptographic algorithms used such as whether symmetric or asymmetric and details of the cryptographic key.

### 5.1.2 Cryptographic Aware Caller

A cryptographic aware caller is aware of underlying aspects of the cryptographic service. It may therefore be aware of whether data are being enciphered or a checkvalue generated. A cryptographic aware caller may or may not be aware of details of the algorithm and keys used. A cryptographic aware caller may be further classed as either *cryptographic policy unaware* or *cryptographic policy aware*.

#### Cryptographic Policy Unaware

A cryptographic policy unaware caller invokes cryptographic services within a previously defined cryptographic context. That is, it is responsible for invoking appropriate cryptographic transforms, but is not responsible for the creation of the cryptographic context, such as the algorithm used, within which the transforms are made. Examples are application infrastructure supporting a secure RPC service and a secure messaging application.

#### Cryptographic Policy Aware

A cryptographic policy aware caller is responsible for the establishment of the cryptographic context of a set of operations through the selection of appropriate algorithm, generation of key and definition of key usage.

For the purposes of this specification a cryptographic policy aware caller is further categorised as being *cryptographic policy selecting* or *cryptographic policy enforcing*.

#### Cryptographic Policy Selecting Caller

A cryptographic policy selecting caller is a caller that is capable of selecting which of a set of predefined cryptographic contexts is to be used for a particular set of services. This type of caller is only permitted to modify such cryptographic contexts in a manner that reduces the scope of the permitted cryptographic operations and hence increases security.

#### Cryptographic Policy Enforcing Caller

A cryptographic policy enforcing caller is responsible for cryptographic policy. This specification distinguishes between two types of cryptographic policy enforcing callers:

- **Key Usage Policy Enforcing Callers**
  A key usage policy enforcing caller is responsible for key usage policy through the selection of appropriate algorithms and key usage parameters when creating a cryptographic context for a set of operations. However, it is not responsible for the integrity of the cryptographic service and the protection of key values. A key usage policy enforcing caller only handles keys in a protected, not a clear, format. Examples are a key distribution application and an authentication module.

- **Key Protection Policy Enforcing Callers**
  A key protection policy enforcing caller is responsible for the protection of the cryptographic service and the key values it generates and uses. A key protection policy enforcing caller may therefore handle keys in the clear, and may be responsible for the administration of the cryptographic services. Examples are a Master Key installation application and an authentication module that handles a user password or other such unprotected authentication credentials.

## 5.2 Scope

The scope of the current specification considers only services to support Cryptographic Aware callers. As such, the interface specification is provided for use by programmers who are cryptographic aware and who develop applications that rely on cryptographic services and key management services. Support for Cryptographic Unaware callers, that is a high level *protect* interface that supports the invocation of confidentiality protection, or integrity protection, or both to an entity without knowledge on the part of the caller of how such protection is provided is deferred to a later specification.

The objectives to be met by the interfaces defined in this specification may be categorised as functional and non-functional. In addition, legal constraints on the use of some cryptographic services need to be accommodated.

### 5.2.1 Functional Objectives

A common set of functions are required to support all types of Cryptographic Aware callers. These are termed *General Application Cryptographic Services* and comprise the following:

1. integrity checkvalue generation and verification
2. data encipherment and decipherment
3. production of irreversible hash of data
4. generation of random numbers
5. inquiry of available keys and key related data.

Cryptographic Policy Aware callers, such as key management applications require the following additional functions:

1. generation, derivation and deletion of keys, including public parameters
2. export and import of keys
3. storage and retrieval of keys and associated information.
4. archive and retrieval of keys and key related data.

The maintenance of an authenticated session previously established with the cryptographic service is an additional objective of this specification.

### 5.2.2 Non-Functional Objectives

The non-functional requirements to be supported by this specification are the requirements that make this specification *Generic* and include:

1. the API shall be cryptographic algorithm independent
2. the API shall be application independent
3. the API shall be cryptographic subsystem independent. (That is, appropriate to both hardware and software implementations)
4. the API shall not impose a particular placement of access control to cryptographic services within an operating system kernel
5. the API shall not constrain future extensibility.

### 5.2.3 Legal Constraints

Many governments currently place constraints on the export of products that include or invoke cryptographic services. Some additionally place constraints on the domestic supply and use of such products. These constraints include the types of algorithm, the length of keys used, and the type of use.

The existence of such constraints may result in:

1. potentially restricted encipherment and decipherment functions. Such restrictions may be implemented and enforced by providing:
   - functions that are available at run-time only to suitably privileged callers, implying authorisation functionality, or
   - functions that are available only at build time for incorporation in specific applications.
2. control on the usage of keys
3. control on the unauthorised replacement of algorithms
4. authentication of the cryptographic subsystem.

### 5.2.4 Functionality that is Out of Scope

The following areas are identified as out of scope of the current version of this specification:

1. The initial authentication of cryptographic service callers and user management are considered out of scope as these services are the application of a more general authentication service which should be developed separately to this specification. However, support for the continuity of such authentication once established is included.
2. Mechanisms for the setting of defaults (for example default CCs) is implementation defined and if individual per caller then defaults are set by *gcs_initialise_session*( ).
3. Enforcement of authorisation for the use of cryptographic services and hence provision of access control managers is required of an implementation but is implementation specific and therefore no specific measures are directly included in this specification. The only provision is recognition within the interface specifications of the possible failure of a call because of an authorisation failure.
4. The requirement by some governments to use a specific algorithm for password encryption and generation may be implemented as an authentication application and is considered out of scope of this specification.

5. Pre-sign functionality in support of the NIST Digital Signature Standard (DSS) is not exposed at the API and can be implemented as an optimisation below the API by an implementation. Invocation of pre-sign functionality implies specific cryptographic awareness on the part of a caller. This specification assumes no necessity for specific algorithm awareness and dependence.
6. High-level application interfaces supporting key distribution and information protection service interfaces for use by cryptographic unaware applications may be implemented by combinations of calls on the services within the scope of the specification. They are therefore considered out of scope of the current specification but could be included in future versions.
7. Certification authority services are an application of the cryptographic services supported by this specification and are therefore more appropriately specified separately to this specification.
8. Installation, initial configuration and subsequent reconfiguration of the cryptographic service itself, which has to be provided by an implementation.
9. Derivation of integrity or confidentiality seeds associated with exported or imported keys.

## 5.3 Layering of Cryptographic Service

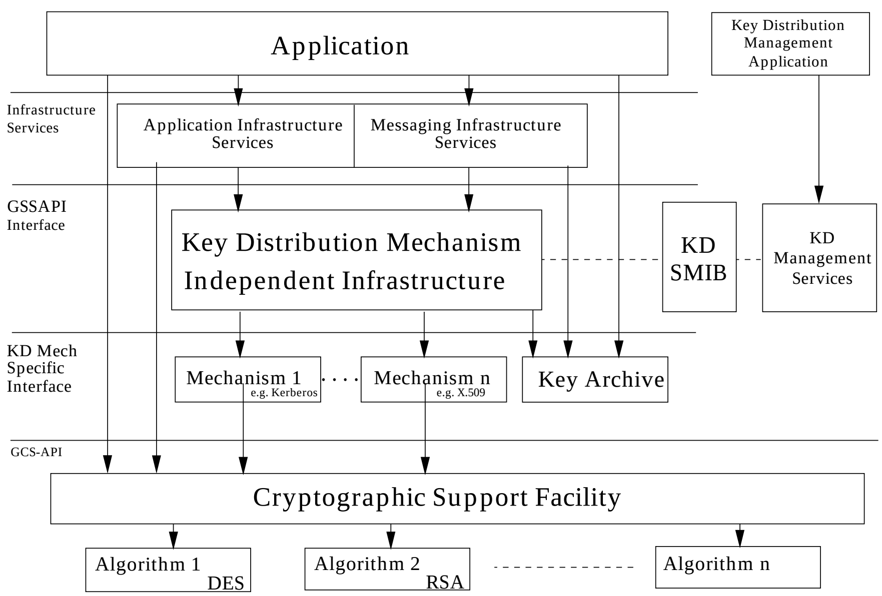


**Figure 5-2** Layering of Cryptographic Services

Figure 5-2 illustrates in some implementation detail the concepts of the layering of services.

At the highest level are applications that need to invoke data protection services via intermediate infrastructure services. These applications are generally cryptographic-unaware.

Next are application infrastructure services, for example RPC services and messaging services, that are responsible for handling the context of the operation, perhaps as a specified *Quality of*

*Protection*, but independent of any mechanism specific aspects. Such functionality is serviced by interfaces at the level of the GSS-API (Generic Security Service Application Program Interface).

The lower layers assume increasing responsibility for details of cryptographic security policy and hence establishment of cryptographic context. This progresses from mechanism independent key distribution services, as part of secure association creation, down through the selection of specific key distribution protocols and algorithms.

The services covered by this specification are shown as implemented within a *Cryptographic Support Facility (CSF)*, see Section 5.4.

The boundaries represented by the different layers of interface may be of particular significance. As discussed later in Section 5.4.2, the CSF interface represents a boundary that is non-bypassable and above which cryptographic keys are not stored or manipulated in the clear by unauthorised (non-cryptographic-enforcing) callers. Above the CSF interface, keys are referenced by a handle or are handled as opaque , cryptographically protected data.

## 5.4 Cryptographic Support Facility

A general Cryptographic Support Facility (CSF) provides a general set of cryptographic and key management service interfaces that sit on top of different algorithms and different implementations of those algorithms. The CSF service interface is capable of hiding any specific algorithm, in particular any key format related to the implementation of a chosen algorithm.

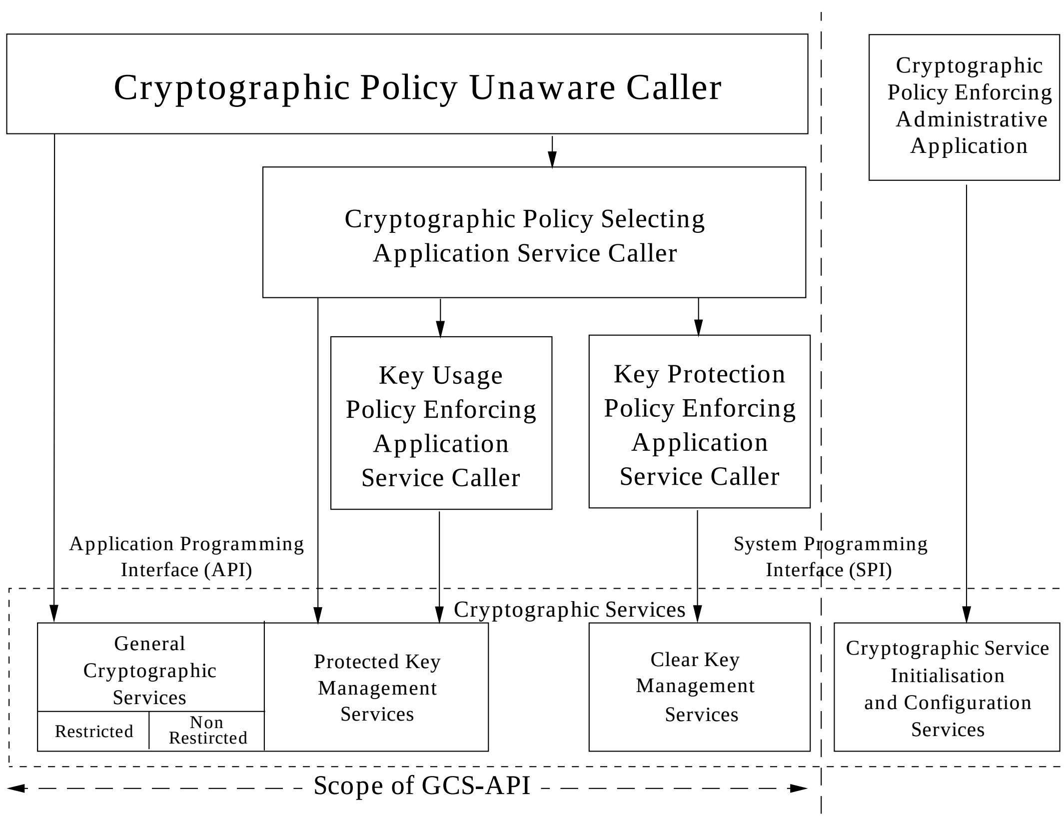


**Figure 5-3** Cryptographic Support Facility Callers and Services

The CSF provides support for applications and application infrastructure that:

- need to invoke a given cryptographic transformation or key management operation
- are not concerned about the details of the operation's implementation, nor whether the underlying technology is provided by software or hardware
- may, but need not, specify for a given operation the Quality of Protection needed
- may, but need not, specify for a given operation which particular cryptographic algorithm is used.

As illustrated in Figure 5-3 the CSF provides two programming interfaces, an *Application Program Interface* (API) and a *System Program Interface (SPI)*, between the various cryptographic aware callers and the following types of services:

**Application Program Interface (API)**

The Application Program Interface comprises interfaces to general cryptographic services and protected key management services:

- **General Cryptographic Services**
  these provide data encipherment, decipherment, production of checkvalues (seals or signatures), checkvalue verification, and are invoked both by callers of the CSF and internal CSF functions for Key Management Support.
- **Protected Key Management Support Services**
  these support cryptographic policy selecting callers and key usage policy enforcing callers by the provision of key generation, storage and distribution services.

**System Program Interface (SPI)**

The System program Interface comprises clear key management services:

- **Clear Key Management Support Services**
  these support key protection policy enforcing callers by the provision of clear key generation, storage and distribution services.

As key distribution protocols become standardised then the Protected Key Management Support Services will increasingly support mechanism dependent functionality. Currently there are no such standards and key distribution protocols are implemented externally to the CSF and require the provision of clear key management support services.

CSF services are identified in Chapter 4 and Chapter 10.

### 5.4.1 Authorisation Policy

The authorisation policy inherent in the GCS-API is defined in terms of authorisation to exercise GCS-API functions and authorisation to access and use specific keys.

Callers of the GCS-API are authorised to access any key created by the principal on whose behalf the caller is operating, or any key to which the creating principal has granted authorisation. The mechanisms by which this authorisation policy is enforced and managed are implementation specific and outside the scope of this specification. Support is included in this specification for the initialisation of a session between a caller and the CSF whereby the identity of the caller may be authenticated and any appropriate access control information established.

The functions a caller may perform on a key are determined by an authorisation policy based on a disjoint set of capabilities assigned to the callers of the GCS-API. These capabilities are associated with the caller itself rather than the principal on whose behalf the caller is acting. The caller may additionally enforce a policy of controlling which of the functions it is authorised to exercise are to be permitted to any individual principal invoking its services.

The capabilities defined for this specification are:

**GCS_C_ENCIPHER_DECIPHER**
The GCS_C_ENCIPHER_DECIPHER authority authorises a caller to utilise the *gcs_encipher_data*( ) and *gcs_decipher_data*( ) functions. The use of such functions may be restricted by an implementation to support legislative restraints on the supply and deployment of cryptographic services.

**GCS_C_SELECTION**
The GCS_C_SELECTION authority authorises a caller to use the Protected Key Management functions, excepting those that set or modify key usage policy.

**GCS_C_KEY_USAGE**
The GCS_C_KEY_USAGE authority authorises a caller to use the Protected Key Management functions that set or modify key usage policy.

**GCS_C_KEY_PROTECTION**
The GCS_C_KEY_PROTECTION authority authorises a caller to use the Clear Key Management functions.

All callers are authorised to exercise the general cryptographic service functions.

### 5.4.2 Security Considerations

Special controls must be applied to the use of cryptographic software due to its fundamental role in distributed system security, and also because of legislative constraints imposed by many countries on the export of software that invokes or contains and exposes cryptographic functions. For example, the USA Government International Traffic in Arms Regulations (ITAR) impose export constraints on products containing cryptographic services — in particular data confidentiality services. Furthermore some countries impose domestic supply and usage controls.

A CSF implementation must take into account a number of strict security requirements, which are summarised as follows:

- The CSF must prevent unauthorised access to cryptographic services.
- The CSF must prevent unauthorised access to underlying data such as private or secret keys.
- The CSF must verify any control information associated with keys (such as expiration information or usage constraints) before use.
- Depending on the policy enforced, the CSF might require its callers to have been authenticated before they can access its services. A cryptographic product can therefore include authentication and authorisation services, as well as the management and operational cryptographic services.
- Once deposited beneath the GCS-API, keys should never be referenced in the clear by unauthorised callers. Above the CSF interface operational keys are protected, for example by enciphering with the CSF Master-Key. Authorised callers are key distribution services that need to combine an operational key in the clear with other related information to create a mechanism-specific token. Also note that subversion of CSF access controls has more security significance for key management service interfaces than those related to general application cryptographic service interfaces.

Chapter 6

# Key Life Cycle

A key is used by cryptographic algorithms to control the transformations they perform. The longer a key is in use, the more susceptible it is to compromise; once a key is compromised, the protection provided by the key is lost. Thus, there is a need to protect keys, by changing them frequently enough to minimise the risk of compromise.

A key is thought of as having a key life cycle. It is created, used and then retired from use before it can effectively be compromised. A number of valid states in the key life cycle are defined for a key. Normal state transitions in the key life cycle, as illustrated in Figure 6-1 on page 88, are dependent on the period of validity associated with the key. The state determines the operations for which the key may be used.

A key may be held in various formats. For example, a different format may be used for a key that is in operational use to the format used for a key that is being exchanged. It is possible for copies of a key to exist in more than one format and storage media at any given time.

## 6.1 Key State

The basis of cryptographic protection is the use of a *key* as an input parameter to a cryptographic algorithm to control the transformation performed by the algorithm. The protection provided by the cryptographic transform depends upon the protection of the key. A key should not be used indefinitely for the following reasons:

- The longer a key is used the more likely it is to be compromised through discovery.
- The longer a key is used, the more data it protects, and thus the greater the potential loss if it is compromised.
- The more data protected by a key the greater the potential reward to the person discovering the key and hence the greater the temptation to expend the effort necessary.
- The risk that a key may be compromised increases the longer the key is used, and the more data it protects, as cryptanalysis is generally facilitated by the availability of more ciphertext encrypted with the same key.

Therefore a key is generally subject to a security policy governing its permitted uses and its permitted lifetime. As a consequence of such a policy a key may be considered to possess a state indicating its availability for operational use. The state of a key may be considered from the viewpoints of its *operational state* and *validity period* together with its *storage format*. These three aspects interact in a manner illustrated in Figure 6-2 on page 90.

### 6.1.1 Key States

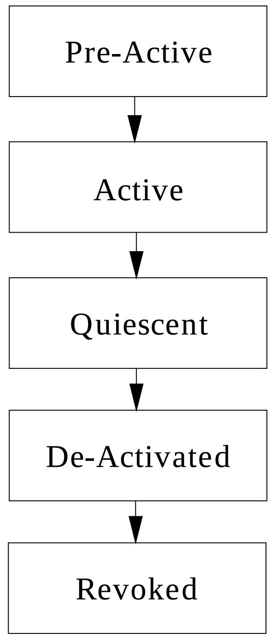


**Figure 6-1** Normal Key State Transitions

A key may possess a number of operational states during its lifetime. Some of these states may be assigned a *quasi-compromised flag* (QCF) which indicates that the key is in a suspicious state, but not yet confirmed as compromised. For example, the QCF might be set on a key if an unauthorised revocation request were received. The QCF indicates that further validation action may be required of the calling application before the key is used.

The following key states are defined:

**Pre-Active State**
A key that is in a pre-active state is not yet available for operational use

**Active State**
A key that is in an active state is available for operational use.

**Quiescent State**
A key that is in a quiescent state is available for a restricted usage. For example, a key in a quiescent state may typically be used to decipher data or verify a checkvalue but not to cipher data or generate a checkvalue.

**De-Activated State**
A key that is in a de-activated state is not available for use within cryptographic transforms.

**Revoked State**
A key that is in a revoked state has been withdrawn from operational use because it is known, or believed, to have been compromised. A revoked key is not available for operational use. Note the QCF does not apply to a revoked key.

### 6.1.2 Key State Operations

There are three basic functions that modify the key state. These are:

**Advance Key State**
This function can be used to step the key state forward.

**Revoke Key**
This function sets the state of a key to revoked thus inhibiting its further operational use. It is intended for use by an application when a key is found to have been compromised.

**Set Key Validity**
This is a function that is restricted to use by security policy aware callers. It supports operations that may change a key state against its normal lifecycle. For example, reseting a key from a de-activated or revoked state to a quiescent state for the purposes of verifying a historic checkvalue.

### 6.1.3 Key Validity Period

A cryptographic key has an associated validity period. The validity period defines the period of time during which the key may be used in cryptographic transforms and comprises:

**Activate Point**
The point in time at which the key is permitted to be fully operational.

**Quiescent Point**
The point at which a key is automatically transitioned from fully operational to partially operational. The quiescent point may be defined by a date and time or a number of cryptographic operations or a number of bytes of data processed. A key is typically placed in a quiescent state some time before it fully expires to facilitate a change of keys. In a quiescent state the range of operations for which the key may be used is restricted. For example, the key may be used to verify a cryptographic checksum but not to generate a cryptographic checksum.

**De-Activate Point**
The point at which a key is no longer permitted to be operational. The de-activate point may be defined by a date and time or a number of cryptographic operations or a number of bytes of data processed. (A de-activated key may be made operational again by an authorised application if permitted by the security policy.)

## 6.2 Key State Transitions

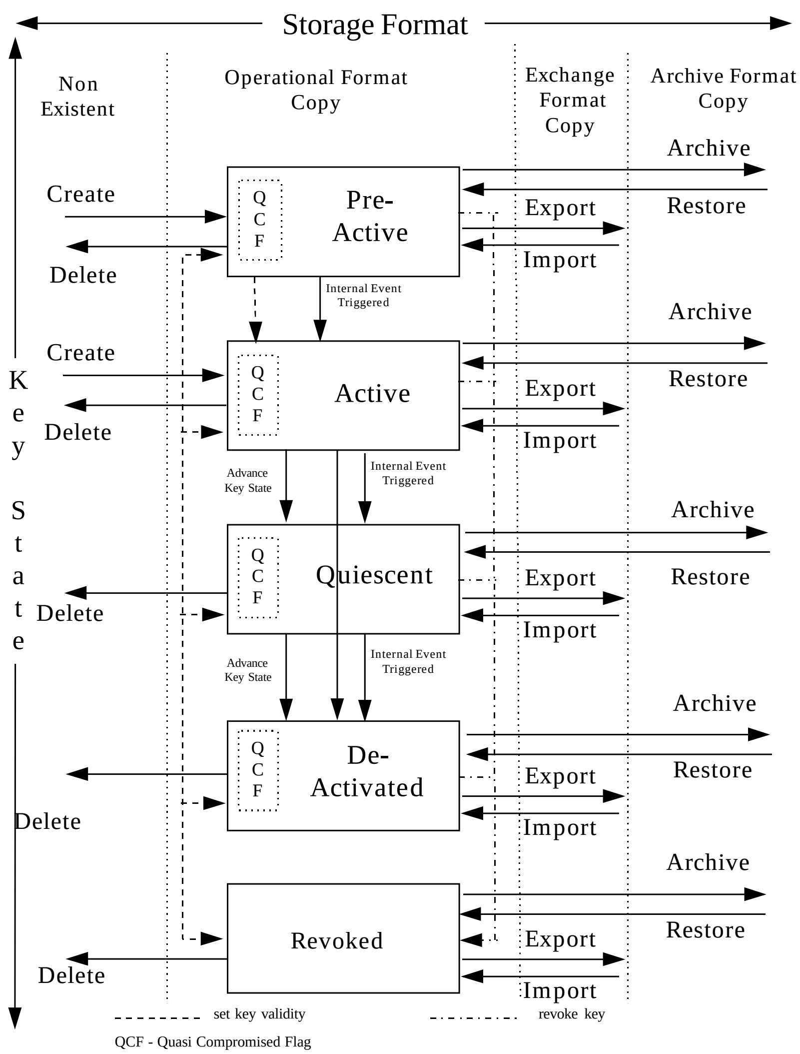


**Figure 6-2** Key Life Cycle

The normal operational key life cycle is to step between the following key states either automatically, by internal events, or by specifically invoked state change operations.

**Pre-Active -> Active**
A key changes from a pre-active to an active state by either:

- an internal event, for example based on the start of its validity date and time, or
- a specific caller invoked operation specifying the active state as the target state.

**Active -> Quiescent**
A key changes from an active to a quiescent state by either:

- an internal event, for example based on the quiescent date and time defined as part of its validity period, or

- a specific caller invoked operation specifying the quiescent state as the target state.

**Quiescent -> De-Activated**
A key changes from a quiescent state to a de-activated state by either:

- an internal event, for example based on the expiry date and time defined as part of its validity period, or
- a specific caller invoked operation specifying the de-activated state as the target state.

**Active -> De-Activated**
A key changes from an active state to a de-activated state by either:

- an internal event, for example based on the expiry date and time defined as part of its validity period, when no quiescent period is defined, or
- a specific caller invoked operation specifying the de-activated state as the target state.

Exceptional transitions are:

**De-Activated -> Quiescent or Active, and Revoked -> Quiescent or Active**
These key state transitions may be required for the purposes of performing a limited set of operations on some historic data. For example, verifying checkvalues used as the basis of a non-repudiation service.

## 6.3 Key Formats

A key may be stored in three formats with respect to the CSF: operational, exchange, and archive. Copies of a key may be present in all three representations concurrently.

**Operational Format**

A key in an operational format is held in a format that permits its use within cryptographic transforms. A key in this state may be held within the cryptographic support facility itself or may be held externally to the CSF. When held externally to the CSF it will be protected, for example enciphered under the CSF master key. The operational format is implementation defined.

**Exchange Format**

The purpose of the exchange format is to permit the exchange of a key between different CSFs for the purposes of key distribution. A key in an exchange format is typically protected under a Key-Encryption-Key (KEK). The exchange format will be dependent upon the key distribution protocol used to support the key exchange, for example X9.17. The definition of such protocols is outside the scope of this specification. A copy of a key in an exchange format will typically not retain control information associated with the key in an operational format unless the key exchange protocol specifically also provides for the exchange of such information. For example, a public key to be used only for validation of data protected under a private key should be set to the quiescent state when an operational format copy of the key is made from the exchange format copy.

**Archive Format**

Archive format is used by a CSF implementation for the long term storage of keys used by that CSF. A key in an archived storage format is typically protected under an archive Key-Encryption-Key (KEK) specific to the key archive system. The archive format is implementation defined.

## 6.4 Key Format Operations

The following operations create copies of a key in the different formats:

**Create**
Create a key in an operational format. The key state of a newly generated key may be pre-active or active.

**Export**
Export creates a copy of an operational key in an exchange format. A key in such a format may be exchanged between cryptographic support facilities by key distribution applications.

**Import**
Import creates a copy of a key in an operational format from a copy of the key in an exchange format.

**Archive**
Archive creates a copy of an operational key in an archive format for long term storage.

**Restore**
Restore creates a copy of a key in an operational format from a copy of the key in an archive format.

*Chapter 7*

# *GCS-API Data Structures*

In invoking a cryptographic operation it is insufficient for a caller to simply supply the input data and a key. Other information has to be assembled such as which algorithm is to be used and how it is to be used. For example:

- When a key is created then the security policy may require that the operations for which the key is to be used or the way in which is handled are to be restricted. This information needs to be bound to the key and the policy enforced by the CSF for each use of the key.
- As described in Chapter 6, a security policy is applied to control the period for which a key is available for use and that a key state is maintained and bound to the key.
- An algorithm may require a set of algorithm specific information to be supplied as well as a key.

To facilitate the specification and maintenance of this contextual and state information and its binding to a key, this specification represents this information and a key as a single logical data structure termed a *Cryptographic Context*, also referred to as a *CC*.

The physical internal structure of a CC is implementation defined. A CC is handled as an opaque object by callers of the CSF. The contents of a CC are potentially updated by the CSF each time it is used to reflect state changes. A Cryptographic context is therefore generally both an input and an output parameter to GCS-API functions. The CSF is responsible for maintaining the integrity of a CC as a whole, protecting it against unauthorised modification, and also for protecting the confidentiality of the key value it contains against unauthorised disclosure.

When created, a CC is a transient structure only accessible to the creating caller. A CC may be made persistent and globally accessible, subject to authorisation policy, by a call on the CSF. To support the handling and management of globally accessible CCs by applications facilities to associate both an internal name, a *CC_reference*, and caller defined name, *CC_name*, with a CC are supported.

## 7.1 Cryptographic Context

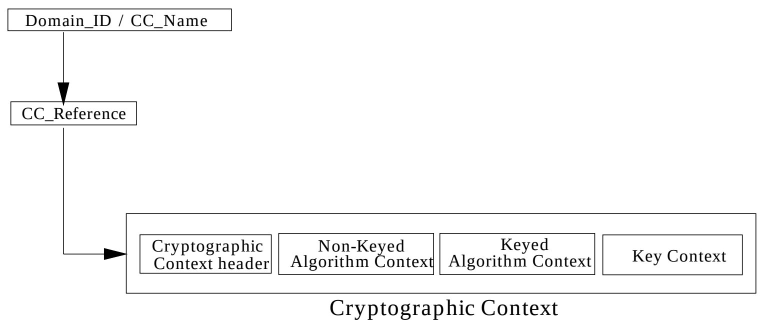


**Figure 7-1** Structure of the Cryptographic Context

Figure 7-1 illustrates the logical structure of the cryptographic context used to support all functions provided by this specification together with its relationship to a *CC_reference*. A CC comprises:

**Cryptographic Context Header**
This contains information pertaining to the context as a whole.

**Algorithm_Context(s)**
Contain information related to the cryptographic algorithm(s) used. This is information that is applicable to many key instances. Two such structures may be included: one for keyed algorithms and one for non-keyed algorithms, both of which may be used within a single context.

**Key_Context**
A Key Context also contains information related to a particular algorithm or mechanism. However, in this case the information is applicable to a specific instance of a cryptographic key.

**CC_Reference**
A CC_Reference is an internal CSF name assigned to a CC by which it may be referenced by callers other than the creator. The reference can be passed between processes sharing a single CSF.

**CC_Name**
As well as a CC_Reference a CC may also be assigned a caller defined name. The caller defined name may be used for ease of reference and an indication of its purpose when assigned to a CC that has been populated with a key. The caller defined name may be used to identify a preconfigured cryptographic policy or quality of protection when assigned to a CC that is unpopulated.

The following data structure definitions are logical definitions and do not imply a physical implementation. The contents of the CC defined in this specification are those necessary to comply with the specification. The inclusion of additional information in a CC by an implementation is not precluded.

## 7.2 Cryptographic Context Header

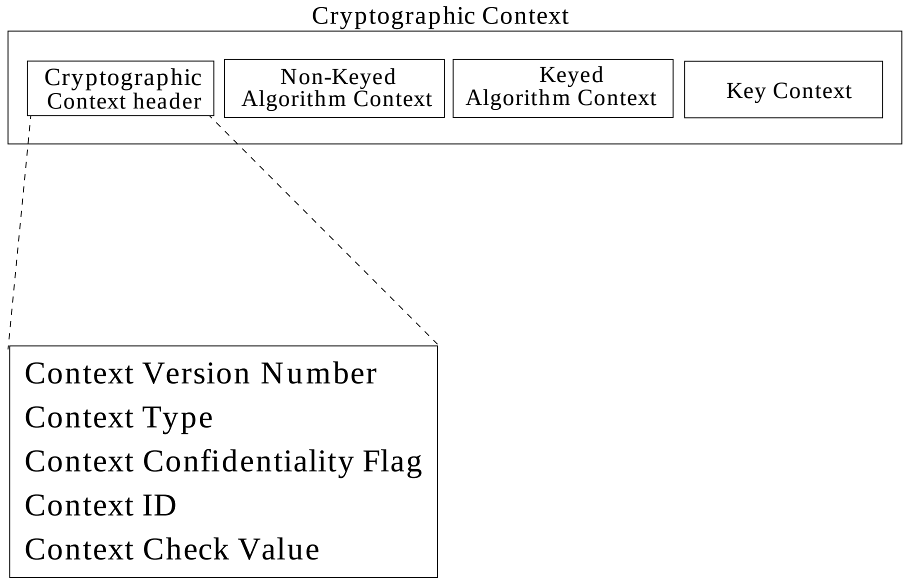


**Figure 7-2** Cryptographic Context Header

As illustrated in Figure 7-2, the CC_header comprises:

**Context_Version_Number**
Version number of the cryptographic context which may be of relevance for implementations of future versions of this specification. The Context Version Number defined by this specification is 0. The Context_version_Number is set by the CSF when a CC is created.

**Context_Type**
Specifies the type of algorithm context(s) included in the cryptographic context. That is, Keyed, Non-keyed or both. The value of this field is set by the CSF when the CC is created.

**Context_Confidentiality_Flag**
This field indicates whether or not the private or secret values held in the *key context* are to be protected for confidentiality. If they are not protected for confidentiality then the CC is only usable by callers possessing a GCS_C_KEY_PROTECTION authority.

- **YES** means that the private or secret values of the key_context shall be protected for confidentiality when populated with a key.
- **NO** means that the private or secret values of the key_context do not need to be protected for confidentiality, although they may be.

The value of this field is specified by a caller of *gcs_create_cc*( ) or *gcs_set_cc*( ).

**Context_ID and Context_Checkvalue**
The context identity and the context checkvalue are used internally by the CSF. The **Context_ID** is a unique identity assigned to a CC by the CSF when it is created. This

identity may be used by the CSF for the purposes of:

- maintaining consolidated usage statistics of a stored CC when retrieved and used by multiple callers concurrently,
- enforcing exclusive update access for modifying a CC,
- supporting access control. For example, it may be used to associate an ACL with the CC.

The **Context_Checkvalue** holds an internally generated and maintained checkvalue of the protected CC. The checkvalue is computed over all CC fields except the *Context_Checkvalue*. The method used to generate the checkvalue is implementation defined.

## 7.3 Algorithm_Context

An *Algorithm_Context* contains information related to a cryptographic algorithm to be used with a CC. This is algorithm specific information that is applicable to many key instances. Two such structures may be included in a CC: one for keyed algorithms and one for non-keyed algorithms, both of which may be used within a single context.

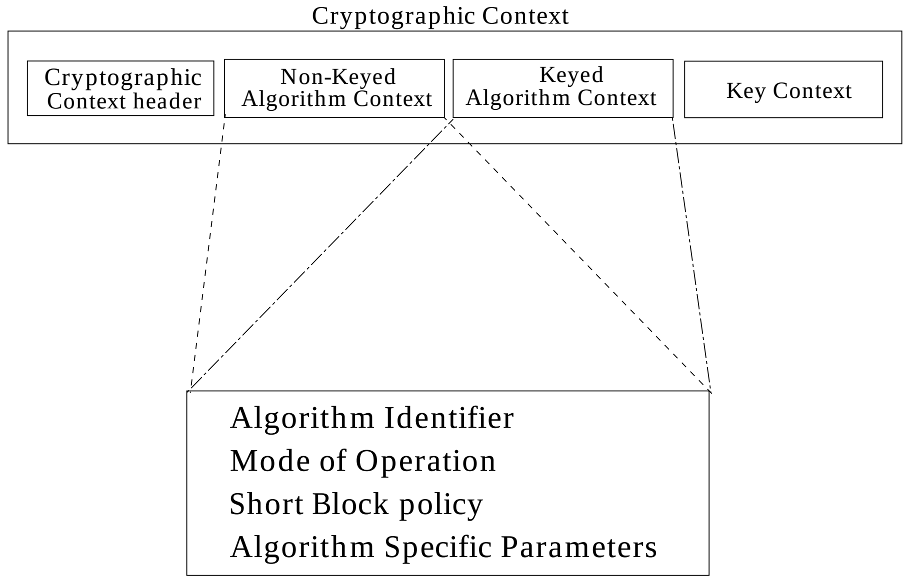


**Figure 7-3** Algorithm_Context

As illustrated in Figure 7-3, the algorithm_context comprises:

**Algorithm Identifier:**

This is defined constant that identifies the specific algorithm to be used. The algorithm ID may also identify the mode of operation, alternatively this may defined separately. Example algorithms are:

**Encipher/decipher algorithms**

- DES
- DES-MAC
- SKIPJACK
- CDMF
- IDEA
- RC(2,4,5)

**check_value algorithms**

- RSA

- DSA

**hash algorithms**

- SHA-1
- MD5

An initial set of Algorithm IDs are given in Section 9.3.4 on page 115.

**Mode of Operation**

The Mode Of Operation identifies the mode in which the selected algorithm is to be operated. The mode usually defines a feedback method and some other simple operations. The mode of operation may be indicated by the algorithm ID in which case the mode of operation can be set to NONE.

Examples of modes of operation are:

- Electronic Feedback Mode (ECB)
- Cipher Block Chaining Mode (CBC)
- Cipher feedback Mode (CFB)
- Output Feedback Mode (OFB)

**Short_Block_Policy:**

The Short_Block_Policy identifies the policy to apply if the caller submits a short block to a function call. Examples of Short Block Policies are:

- **None**
  Short blocks are not permitted. Input must be a multiple of block size.
- **X9.23**
  X9.23 uses byte padding. A short block is padded from 1 up to to 8 bytes. The last byte is the count of the number of bytes of padding.
- **IBM Information Protection System (IPS)**
  IBM IPS reciphers the last complete ciphertext block and re-enciphers and then XOR with plaintext for the required number of bytes. This acts like a psuedo one-time pad.
- **Cipher Text Stealing**
  Cipher Text Stealing encrypts normally up to the last few bytes. It then prepends ciphertext bytes to the remaining cleartext bytes to make up a complete block and then enciphers the complete block. This can also be used on the basis of bit-length as well as byte-length.
- **PKCS#1**
  Encryption block formatting as defined in PKCS#1.

**Algorithm_Specific_Parameters**

These are parameters required by the specific algorithm referenced by the algorithm context that are not specific to a single key to be used with the algorithm. The Algorithm Specific Parameters are defined by the standard that defines the Algorithm Object ID.

Examples of Algorithm Specific Parameters for some common algorithms are:

**DES**

- Key length - 64 bits

- Feedback length (for some block cipher modes)
- IV parameters (e.g.,length)

**RSA**

- Modulus length (this controls the size of the prime numbers, strength of the key)
- Optional User Group Parameters
  The following two parameters both have to be supplied if the values are shared between a group of users:
  — Group public exponent length
  — Group public exponent value

**DSA**

- Length of Prime P in bits (512 to 1024 bits, this controls the strength of the key)
- Optional User Group Parameters
  The following three parameters are all required to be supplied if the values are shared between a group of users:
  Prime **p**
  — Prime **q**
  — Generator **g**

**Diffie Hellman**

- Length of prime P in bits (512 to 1024 bits, this controls the strength of the key)
- Prime **P**
- Generator **G**, (1 < G < P )
- Number of Parties
- Derive (Spawn) Method
  Indicates how to interpret the input bit string to gcs_derive_key.
- Elliptic Curves-Diffie Hellman
  Elliptic curve parameters: curve parameters, curve order and generator point.

## 7.4 Key_Context

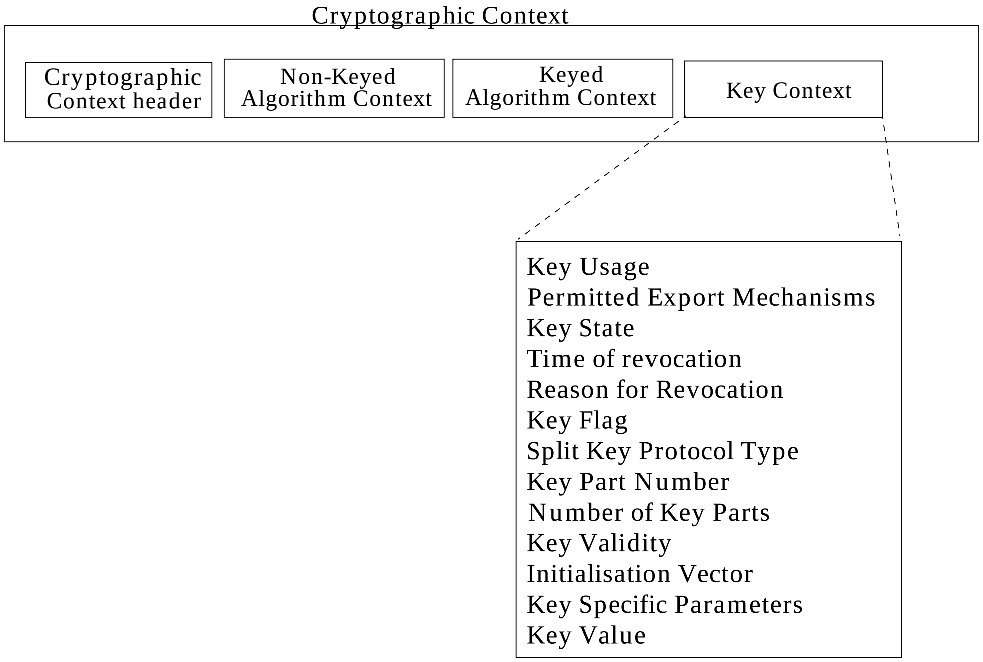


**Figure 7-4** Key_Context

As illustrated in Figure 7-4, the Key_Context comprises:

**Key_Usage**
This field defines for which functions the key that populates the CC may be used as the key for that cryptographic transform. A complete list of all functions subject to key usage constraints can be found in Section 3.8 on page 33. Once populated with a key the key_usage may only be reduced in scope.

**Permitted_Export_Mechanisms**
The Permitted_Export_Mechanism, identified by an Mechanism ID, defines which, if any, mechanisms may be used to transport the key contained in the CC between CSFs using *gcs_export_key*( ) and *gcs_import_key*( ) or *gcs_export_key_agreement*( ) and *gcs_import_key_agreement*( ). Examples that may be defined include:

- No export, the key is not permitted to be exported.
- X9.17
- Kerberos
- RSA - ANSII
- RSA - PKCS
- FORTEZZA Key-Wrap
- Control Vectors

- Diffie Hellman - X9.42 (dynamic case)
- Diffie Hellman - [Photuris]
- KEA

Many export mechanisms are the subject of draft standards and are under development. Specific examples with currently defined object Ids are listed in Section 9.3.9 on page 117.

**Key_State**
Identifies the current state of the key (pre_active, active, quiescent, de-activated or revoked) See Chapter 6.

**Time_of_Revocation**
Specifies the date and time at which the key was revoked. This is set by the CSF.

**Reason_For_Revocation**
This is a text string used to record the reason for which a key has been revoked. This is supplied by the caller revoking a key.

**Key_Flag**
Refines the state of the key and provides control of the functions to which the key may be a target.

- **IV_Needed** If set then a caller is required to supply an IV to the functions that provide for an IV input parameter, e.g., *gcs_encipher_date*( ) and *gcs_decipher_data*( ).
- **Split**
  Specifies whether or not the key is split.
- **Quasi Compromised (QCF)**
  Specifies whether the key is suspected of having been compromised but that this has not yet been authoratively confirmed.
- **Force_First_Key_Usage**
  Specifies by the first call whether the key is used for encryption/decryption, or for generating and/or verifying a check value. This provides support for X9.17 with ambiguous usage.

**Split_Protocol_Type**
If the CC contains a split key, this field defines the protocol used to split the key. This field is checked by *gcs_split_clear_key*( ). Examples of split protocol types are XOR and SHAMIR.

**Key_Part_Number**
If the CC contains a split key, this field defines the part number contained within the CC.

**Number_of_Key_Parts**
If the CC contains a split key, this field defines the total number of key parts into which the key has been split.

**Key_Validity**
The key validity data comprises:

- **activation time**

  This is the date/time after which the key is permitted to be used for cryptographic operations.

- **quiescent time**

  This defines the point in time after which the key is set to the quiescent state, that is it may be only used for a restricted set of operations. This point in time may be defined as a number of seconds after activation or a number of cryptographic operations or a number of bytes enciphered. This point in time may be defined using all three methods within a single CC.

- **deactivation time**

  This defines the point in time after which the key is set to the deactivated state, that is it may no longer be used for any cryptographic operations. This point in time may be defined as a number of seconds after activation or a number of cryptographic operations or a number of bytes enciphered. This point in time may be defined using all three methods within a single CC.

**Initialisation Vector**

This is a static IV value to be used by the CSF for all functions requiring an IV for which this CC is used unless overridden by a caller supplied IV parameter. A caller may be forced to supply an IV value to functions by setting the *IV_NEEDED* flag described above.

**Key_Specific_Parameters**

These are additional mechanism specific parameters that are associated with this key. Examples are: KEK_ID, Key_ID for ANSI X9.17, usage count, send counter, receive window, parity checked, parity set, etc.)i

**Key_Value**

The key value is implementation dependent and has a variable structure dependent upon the algorithm. (The key length is defined as an Algorithm Specific Parameter within the Algorithm Context.) Keys may have internal structure which is not visible to the API.

If the Context_Confidentiality_Flag is set then the private or secret values held within the *Key_Value* field have to be confidentiality protected by the CSF. This is typically done by enciphering under a CSF Master Key.

## 7.5 Cryptographic Context Reference

When created a CC is a transient structure only accessible to the creating caller. A CC may be made persistent and globally accessible, subject to authorisation policy, by a call on the CSF. This call stores a copy of the CC under the control of the CSF. To support the handling and management of such globally accessible CCs by applications a system defined name, a *CC_reference*, is associated with a stored CC. A CSF may be able to use different types of storage media in which to store CCs. The definition of a *CC_reference* supports the definition of the storage media and device by a caller. To improve usability a stored CC may also be aliased by a caller defined name.

The system defined name, the CC_reference, is defined as follows:

label
storage_unit_class [optional]
storage_unit_instance [optional]

Where:

**Label:**
Is the system defined name assigned to the cryptographic context stored in the operational storage unit maintained by the CSF. This is an internal machine-generated name and not a human-readable name.

**Storage_Unit_Class:**
Is an optional parameter which distinguishes the device on which the cryptographic context is stored. This parameter, which could have a default value, could be handled by the CSF implementation, or could be tuned by the caller. For example, non-volatile memory, disk, CD-ROM, smart_card.

**Storage_Unit_Instance:**
Is an optional parameter used to distinguish between different instances of the same storage unit class.

**Note:** the CC_reference is implementation-specific. It must be unique within an individual CSF domain.

## 7.6 Cryptographic Context Name

In addition to the use of the *CC_reference* to reference a CC this specification supports the assignment of a caller defined name to identify a stored CC. The caller defined name comprises two components:

**Domain_ID**
This identifies the security domain to which the **CC_name** relates. This may be defaulted.

**CC_name**
A name that must be unique within the domain *Domain_ID*.

This structure enables implementations to support:

- **Definition of Quality of Protection (QOP) Profiles**
  A set of unkeyed CCs may be created and stored to define the QOP policy within the identified domain. The domain may represent an interconnection security domain between two peers or may represent a storage domain, for example a backup service. The QOP represented by each CC may be represented by its *CC_name*.

- **Sharing of Keys between Callers of Different CSFs**
  A key that is distributed between different CSFs (via export and import operations) may be readily named and identified by co-operating callers of each CSF.

# Advanced GCS-API Services

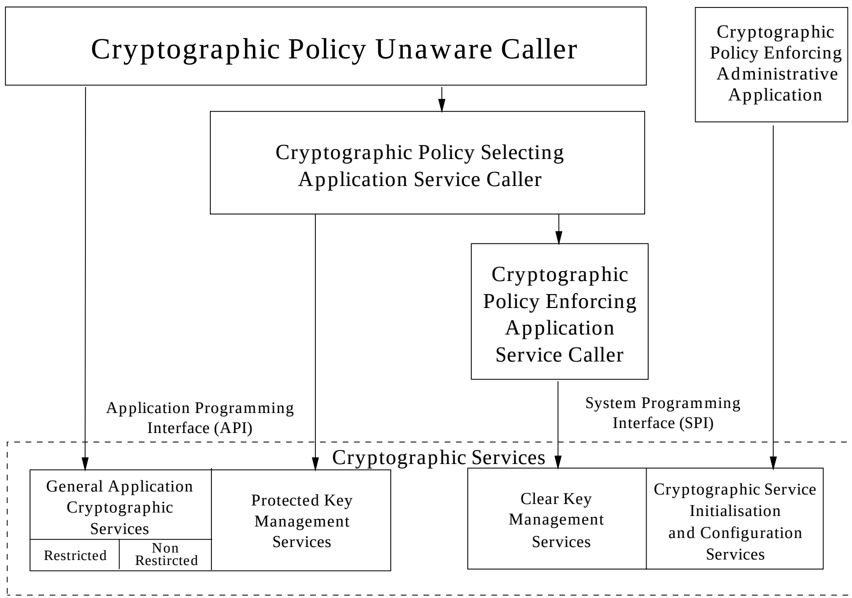


**Figure 8-1** CSF Services

The CSF services comprise both operational and management services and are illustrated in Figure 8-1.

They include the following categories:

- General Cryptographic Services (Part of the API)
- Protected Key Management Services (Part of the API)
- Clear Key Management Services (Part of the SPI)
- Cryptographic Service Initialisation and Configuration Services (Not within the current scope of this specification.)

As described in Section 5.4.1 on page 85 callers of the CSF are authorised to utilise CSF functions on the basis of a disjoint set of capabilities assigned to them.

This chapter describes the additional advanced services supported by the GCS-API. These fall within the protected key Management and Clear key Management Services illustrated in Figure 8-1. The basic services supported by the GCS-API are described in Chapters 1-4. Each subsection lists the functions supported and the GCS Authorities, if any, required by a caller in order to successfully invoke the function. GCS Authorities are described in detail in Chapter 2. In general these are assigned by administrative action and established on the initialisation of a session with the CSF. A detailed manual page for each of these functions is included.

## 8.1 Creation of CC

| Function | GCS Authorities |
|---|---|
| gcs_create_ac | - |
| gcs_delete_ac | - |
| gcs_set_ac | - |
| gcs_create_kc | - |
| gcs_delete_kc | - |
| gcs_set_kc | - |
| gcs_create_cc | GCS_C_KEY_USAGE |

**Table 8-1** Creation of a CC

A cryptographic context can only be created by authorised callers, ie., those that enforce cryptographic key usage policy. This is indicated by a caller being assigned a GCS_C_KEY_USAGE authority.

The specification only requires GCS_C_KEY_USAGE authority for *gcs_create_cc*( ) as this is the only interface that actually creates a CC. The other interfaces are supporting functions and are ineffective without the other one.

The cryptographic context is built up from one or two algorithm contexts and a key context in the following manner:

- Empty algorithm contexts and key contexts are created with calls to *gcs_create_ac*( ), and *gcs_create_kc*( ), respectively. Each of these functions allocates memory for the context as required.
- The created algorithm contexts and key contexts are filled by successive calls to *gcs_set_ac*( ) and *gcs_set_kc*( ), to set individual fields in each of the data structures. The key_value field of the key_context is not filled at this time.
- A cryptographic context is created by using the function *gcs_create_cc*( ), supplying it with appropriate algorithm and key contexts already created to define the policy represented by the CC. The cc_header fields are filled at this time. A caller of *gcs_create_cc*( ) is required to possess a GCS_C_KEY_USAGE authority.
- Once the CC has been created, the independent algorithm context and key context structures created to form the CC may be deleted and the memory occupied by them released by calls to *gcs_delete_ac*( ) and *gcs_delete_kc*( ).

This set of operations creates an template cryptographic context, which can either be populated for immediate use or stored in a library and made globally referencable.

## 8.2 Cryptographic Context Modification

| Function | GCS Authorities |
|---|---|
| gcs_get_cc | - |
| gcs_retrieve_cc | |
| gcs_set_cc | GCS_C_KEY_USAGE |
| gcs_store_cc | GCS_C_SELECTION or GCS_C_KEY_USAGE |

**Table 8-2** Cryptographic Context Inquiry

*gcs_get_cc* provides for the querying by a caller of the contents of a CC. Any CC content may be queried with the exception of the key value.

The CC contents may be modified using *gcs_set_cc*( ) to overwrite an individual field in the algorithm context(s) and key context in the cryptographic context. Multiple calls to *get_set_cc*( ) need to be made in order to modify several fields. A caller of *gcs_set_cc*( ) is required to possess a GCS_C_KEY_USAGE authority.

To modify a CC that has been stored then *gcs_retrieve_cc*( ) must be invoked with an exclusive lock set. This prevents any subsequent retrieval of the CC and also results in the failure of any cryptographic operations using a copy of the CC that has been previously retrieved.

A subsequent call by the caller that executed the exclusive lock to *gcs_store_cc*( ) using the CC on which the lock was obtained results in the stored copy of the CC being updated and the lock released. Any subsequent calls using a previously retrieved version of the CC result in that caller's private copy being updated as a consequence of the call it makes. The update of CC may result in a caller being unable to continue using the CC for operations it was previously capable of executing.

## 8.3 Additional Key Management Functions

| Function | GCS Authorities |
|---|---|
| gcs_combine_key | GCS_C_SELECTION |
| gcs_load_public_key | GCS_C_SELECTION |

**Table 8-3** Additional Key Management Functions

These functions provide additional facilities for the management of keys. *gcs_combine_key* provides for the combination of key parts into a single key. The individual key parts have to be imported to separate CCs and this function is then invoked to combine the individual key parts.

*gcs_store_cc* provides for the creation of a new CC that is identical to the original CC with the exception of its Context ID if the original CC was not retrieved with an exclusive lock set.

*gcs_load_public_key* provides for the loading of a public key supplied in clear text form, which by its nature does not require protection for confidentiality.

## 8.4 Key State Management

| Function | GCS Authorities |
|---|---|
| gcs_advance_key_state | GCS_C_SELECTION or GCS_C_KEY_USAGE |
| gcs_get_key_validity | - |
| gcs_reduce_key_usage | GCS_C_SELECTION or GCS_C_KEY_USAGE |
| gcs_revoke_key | GCS_C_SELECTION or GCS_C_KEY_USAGE |
| gcs_set_key_validity | GCS_C_KEY_USAGE |

**Table 8-4** Key State Management

The key state management functions provide for a caller to query and modify the key state and the parameters that control the key state.

*gcs_advance_key_state* provides for a caller to step the key state of a CC forward through its natural lifecycle in a manner that reduces the key's availability. For example, a CC with a key in an active state may be stepped forward to a quiescent or de-active state, but a CC with a key in a pre-active state cannot be made active.

*gcs_get_key_validity* provides for a caller to query the key validity parameters of the CC that control the points at which the CSF will trigger a key state change. *gcs_set_key_validity* provides for a caller to set the key validity parameters of a CC and therefore control when the CSF will trigger a key state change. This may be necessary in order to reactivate a key that has been previously de- activated or revoked for the purposes of deciphering or verifying historic data.

*gcs_reduce_key_usage* is used by a caller to reduce the cryptographic functions that a key may be used for. An example may be creation of a copy of a key (via *gcs_store_cc* with an exclusive lock) which is to be restricted to only decrypting data prior to making it available to other callers.

*gcs_revoke_key* provides for a caller to set a key into a revoked state when it has been found to be compromised.

## 8.5 Supplementary CC Management Functions

| Function | GCS Authorities |
|---|---|
| gcs_archive_cc | GCS_C_SELECTION or GCS_KEY_USAGE |
| gcs_restore_cc | GCS_C_SELECTION or GCS_KEY_USAGE |
| gcs_generate_key_pattern | GCS_C_SELECTION |
| gcs_verify_key_pattern | GCS_C_SELECTION |

**Table 8-5** Supplementary CC Management Functions

These functions provide supplementary services in support of the management of CCs. *gcs_archive_cc* and *gcs_restore_cc* provide for the long term storage of CCs. That is of both keys and the context in which they are used including key usage constraints. These services are not likely to be used for normal day to day operations but are required to support the recovery of historic keys and associated data.

*gcs_generate_key_pattern* and *gcs_verify_key_pattern* are provided in support of key derivation functions to enable CSF implementations that independently derive the same key from caller supplied data to check that the independently derived keys are identical and will reliably interwork.

## 8.6 System Programming Interface (SPI)

| Function | GCS Authorities |
|---|---|
| gcs_decipher_key | GCS_C_KEY_PROTECTION |
| gcs_encipher_key | GCS_C_KEY_PROTECTION |
| gcs_derive_clear_key | GCS_C_KEY_PROTECTION |
| gcs_generate_clear_key | GCS_C_KEY_PROTECTION |
| gcs_load_clear_key | GCS_C_KEY_PROTECTION |
| gcs_split_clear_key | GCS_C_KEY_PROTECTION |

**Table 8-6** System Programming Interface

The system programming interface supported by the GCS-API provides functions for the manipulation of clear keys by a caller. These types of functions are required to support the management of the CSF itself, for example the installation of initial keys and for support of key exchange protocols that require the manipulation of clear keys when such protocols have not been directly implemented by a CSF implementation.

*Chapter 9*

# *Advanced GCS-API Parameter Passing Conventions*

This chapter describes the additional data types, over and above those defined in Chapter 3 on page 25, used by the C-language versions of the advanced GCS-API functions. It also explains calling conventions for these functions.

## 9.1 Contexts

The **gcs_cc_t** data type contains a caller-opaque cryptographic context defined by the implementation. The cryptographic context holds the algorithm context and key context information.

**gcs_ac_t** data type contains an algorithm context defined by the implementation.

**gcs_kc_t** data type contains a key context defined by the implementation.

## 9.2 Cryptographic Reference

The **gcs_cc_ref_t** data type contains a handle to a caller-opaque cryptographic context defined by the implementation.

## 9.3 Constants

The tables below set out the constants defined by the specification, and the value to which they are set.

### 9.3.1 Register of GCS-API Constants

At the time of publication it is not possible for this specification to include the values of all constants that will be relevant to the GCS- API in the future. This is because cryptography is a developing technology and new algorithms, export mechanisms, etc., will continue to be developed and values to identify them within the GCS-API will need to be defined.

To provide for this extension of GCS-API constants a register of GCS-API constants is maintained by X/Open. The latest version of this may be accessed at the X/Open WWW Server at www.xopen.org by reference to the index at URL:

http://www.xopen.org/public/

or by anonymous ftp to:

ftp.xopen.co.uk

cd pub/GCS-API_Registry
get GCS-API_Constants.ps

To register a new a constant or range of constants an implementor should send a message via email to **GCS-API-Registry@xopen.co.uk**.

Registration of an algorithm ID requires the specification of the name of the algorithm together with a list of the Algorithm Specific Parameters and the format in which they have to be input. The modes of operation and applicable Short Block Policies shall also be defined. This information may be provided by reference to a standard or publicly accessible specification that defines the necessary information.

### 9.3.2 Optional Parameter Constants

| Name | Value | Meaning |
|---|---|---|
| [GCS_C_TRUE] | 1 | True |
| [GCS_C_FALSE] | 0 | False |
| [GCS_C_NULL] | NULL | Null |
| [GCS_C_EMPTY_BUFFER] | NULL | Empty buffer |
| [GCS_C_NO_BUFFER] | NULL | No buffer is supplied or returned |
| [GCS_C_NO_BIT_STRING] | NULL | The bit string supplied or returned is null |

**Table 9-1** Optional Parameter Constants

### 9.3.3 Context Types

| Context Type | Value | Meaning |
|---|---|---|
| Keyed | 0 | Keyed Algorithm Context |
| Non-Keyed | 1 | Non-Keyed Algorithm Context |
| Both | 2 | Keyed & Non-Keyed Algorithm Context |

**Table 9-2** Context Types

### 9.3.4 Algorithm Identifier

The following algorithm identifiers represent an initial list. Their inclusion in this document does not imply any conformance criteria for the supply of these particular algorithms.

An algorithm ID may also indicate a specific mode of operation, alternatively the mode of operation may be specified separately in a CC that uses the algorithm.

| Algorithm | Algorithm ID | Algorithm Parameters |
|---|---|---|
| GCS_C_DES_CBC | 1 | IV 64 bits |
| GCS_C_DES_MAC_32 | 2 | None |
| GCS_C_SKIPJACK_CBC_64 | 3 | IV 64 bits |
| GCS_C_RC2_CBC | 4 | IV or sequence RC2 version, IV |
| GCS_C_RC4 | 5 | None |
| GCS_C_RSA | 6 | Modulus length |
| GCS_C_DSA | 7 | |
| GCS_C_SHA_1 | 8 | |
| GCS_C_MD5 | 9 | |
| GCS_C_KEA | 10 | |
| GCS_C_DIFFIE | 11 | |

**Table 9-3** Algorithm IDs

### 9.3.5 Mode of Operation

The mode of operation qualifies how a particular algorithm is to be used and usually defines a feedback method and some simple operations.

| Name | Value | Mode of Operation |
|---|---|---|
| GCS_M_NONE | 0 | No mode appropriate or algorithm ID specifies |
| GCS_M_ECB | 1 | Electronic Code Book Mode |
| GCS_M_CBC | 2 | Cipher Block Chaining Mode |
| GCS_M_CFB | 3 | Cipher Feedback Mode |
| GCS_M_OFB | 4 | Output Feedback Mode |
| GCS_M_COUNTER | 5 | Counter Mode |
| GCS_M_BC | 6 | Block Chaining Mode |
| GCS_M_PCBC | 7 | Propagating Cipher Block Mode |
| GCS_M_CBCC | 8 | Cipher Block Chaining with Checksum |
| GCS_M_OFBNLF | 9 | Output Feedback with Non-Linear Function |
| GCS_M_CBCOFBM | 10 | CBC with OFB Masking |

**Table 9-4** Modes of Operation

### 9.3.6 Algorithm Specific Parameters

Algorithm specific parameters are defined by the standard that defines the algorithm ID. Algorithm specific parameters are to be represented in the algorithm context by a BER encoding of the format defined in the applicable standard. Examples of Algorithm Specific Parameters are included in Table 9-3 on page 115.

### 9.3.7 Short Block Policies

| Short Block Policy | Value | |
|---|---|---|
| GCS_SBP_NONE | 0 | Short Blocks Not Permitted |
| GCS_SBP_X9_23 | 1 | X9.23 byte padding |
| GCS_SBP_IPS | 2 | IBM Information Protection System |
| GCS_SBP_CTS | 3 | Cipher Text Stealing |
| GCS_SBP_PKCS_1 | 4 | Encryption block formatting as defined in PKCS#1 |
| GCS_SBP_DES_MAC | 5 | DES MAC Short Block Policy |
| GCS_SBP_PEM | 6 | PEM Short Block policy |

**Table 9-5** Short Block Policy Values

### 9.3.8 Key Usage

The key usage parameter defines for which GCS-API functions the CC may be used to provide the key to a cryptographic operation.

**Note:** The Key_Flag parameter controls the functions for which the CC may be the target of a GCS-API function.

| Key Value | Bit Mask Values | CSF Function |
|---|---|---|
| GCS_C_GENERATE_CV | "0x00000001" | gcs_generate_check_value |
| GCS_C_VERIFY_CV | "0x00000002" | gcs_verify_check_value |
| GCS_C_DERIVE_KEY | "0x00000004" | gcs_derive_key |
| GCS_C_ENCIPHER_DATA | "0x00000008" | gcs_encipher_data |
| GCS_C_DECIPHER_DATA | "0x00000010" | gcs_decipher_data |
| GCS_C_ARCHIVE_CC | "0x00000020" | gcs_archive_cc |
| GCS_C_RESTORE_CC | "0X00000040" | gcs_restore_cc |
| GCS_C_GENERATE_KEY_PATTERN | "0x00000080" | gcs_generate_key_pattern |
| GCS_C_VERIFY_KEY_PATTERN | "0x00000100" | gcs_verify_key_pattern |
| GCS_C_DECIPHER_KEY | "0x00000200" | gcs_decipher_key |
| GCS_C_ENCIPHER_KEY | "0x00000400" | gcs_encipher_key |
| GCS_C_EXPORT_KEY | "0x00000800" | gcs_export_key |
| GCS_C_EXPORT_KEY_AGREEMENT | "0x00001000" | gcs_export_key_agreement |
| GCS_C_IMPORT_KEY | "0x00002000" | gcs_import_key |
| GCS_C_IMPORT_KEY_AGREEMENT | "0x00004000" | gcs_import_key_agreement |

**Table 9-6** Key Usage Values

### 9.3.9 Permitted Export Mechanisms

These define which mechanisms, if any, can be used to transport the key contained in a CC between CSFs.

The Mechanism IDs specified as part of the GCS-API are separately maintained by X/Open and are accessible at the X/Open WWW site or ftp site. The following mechanism identifiers represent an initial list. Their inclusion in this document does not imply any conformance criteria for the supply of these particular algorithms.

If a proprietary or non-standardised mechanism is supported then an implementation may apply to X/Open for an mechanism ID for that mechanism.

| Export Mechanism | Bit Mask Values | Meaning |
|---|---|---|
| GCS_NO_EXPORT | "0x00000000" | the key cannot be exported |
| GCS_DH_PKCS3 | "0x00000001" | Diffie Hellman |
| GCS_DHKA_PKCS3_1 | "0x00000002" | Diffie Hellman Key Agreement |
| GCS_FORTEZZA_KEA | "0x00000004" | KEA |
| GCS_X917_1985 | "0x00000008" | X9.17 1985 |
| GCS_X917_1994 | "0x00000010" | X9.17 1994 |
| GCS_KERBEROS | "0x00000020" | Kerberos RFC 1510 |
| GCS_PCK51 | "0x00000040" | X9.44 |
| GCS_RSA_PKCS | "0x00000080" | RSA-PKCS X9.42 |
| GCS_FORTEZZA_KEY_WRAP | "0x00000100" | Fortezza Key Wrap |
| GCS_IBM_CV | "0x00000200" | Control Vectors IBM SC40-1675 |

**Table 9-7** Permitted Export Mechanism IDs

### 9.3.10 Key State Value

The key state value identifies the current state of the key.

| Key State | Value | Meaning |
|---|---|---|
| GCS_PRE_ACTIVE | 1 | pre-active key state |
| GCS_ACTIVE | 2 | key state active |
| GCS_QUIESCENT | 3 | key state quiescent |
| GCS_DEACTIVATED | 4 | key state de-activated |
| GCS_REVOKED | 5 | key revoked |

**Table 9-8** Key State Values

### 9.3.11 Key Flag

The key flag refines the state of the key.

| Key Flag | Bit Mask Value | Meaning |
|---|---|---|
| GCS_C_IV_NEEDED | "0x01" | caller must supply IV |
| GCS_C_SPLIT | "0x02" | if set, the key is split |
| GCS_C_QCF | "0x04" | if set, the key is suspected of having been compromised |
| GCS_C_FORCE_FIRST_USAGE | "0X08" | first usage specifies how key is used |

**Table 9-9** Key Flag Values

### 9.3.12 Split_Key_Protocol_Type

The following split key protocol types are defined:

| Key State | Value | Meaning |
|---|---|---|
| GCS_SKP_NONE | 0 | Key not Split (Default) |
| GCS_SKP_XOR | 1 | XOR split protocol |
| GCS_SKP_SHAMIR | 2 | Shamir split protocol |

**Table 9-10** Split Key Protocol Types

### 9.3.13 Key Validity Parameters

The following constants are defined for use as key validity parameters:

| Key State | Value | Meaning |
|---|---|---|
| GCS_C_TIME | 0 | Input is the number of seconds |
| GCS_C_COUNT | 1 | Input is the number of operations |
| GCS_C_BYTES | 2 | Input is the number of bytes |
| GCS_C_NOW | 0 | Zero offset from current time |
| GCS_C_INFINITE | "0xFFFFFFFF" | An infinite time |

**Table 9-11** Key Validity Values

### 9.3.14 Key Specific Parameters

These are additional mechanism specific parameters associated with the key. They are to be represented as BER encoded data.

### 9.3.15 Key Value

The formatting of key values is generally an internal implementation concern. An exception is the format of clear keys to be used with the functions *gcs_load_key*( ) and *gcs_load_public_key*( ).

For these functions:

- Clear Public Key values or Public/Private Key pair values shall be represented by DER encoding. (See PKCS #1)
- Clear keys for DES-CBC shall be formatted as a 64 bit string with the MSB in the lowest address bit.

### 9.3.16 CC Components

| Name | Value |
|---|---|
| GCS_C_CC_HEADER | 0 |
| GCS_C_KEYED_AC | 1 |
| GCS_C_NON_KEYED_AC | 2 |
| GCS_C_KC | 3 |

**Table 9-12** CC Components

### 9.3.17 Context Header Parameter Names

| Name | Value |
|---|---|
| GCS_C_CONTEXT_VERSION | 0 |
| GCS_C_CONTEXT_TYPE | 1 |
| GCS_C_CONFIDENTIALITY_FLAG | 2 |

**Table 9-13** Context Header Parameter Names

### 9.3.18 Algorithm Context Parameter Names

| Name | Value |
|---|---|
| GCS_C_ALGORITHM_ID | 0 |
| GCS_C_MODE_OF_OPERATION | 1 |
| GCS_C_SHORT_BLOCK_POLICY | 2 |
| GCS_C_ALGORITHM_SPECIFIC_PARAMETERS | 3 |

**Table 9-14** Algorithm Context Parameter Names

### 9.3.19 Key Context Parameter Names

| Name | Value |
|---|---|
| GCS_C_KEY_USAGE | 0 |
| GCS_C_PERMITTED_EXPORT_MECHANISM | 1 |
| GCS_C_KEY_STATE | 2 |
| GCS_C_KEY_FLAG | 3 |
| GCS_C_TIME_OF_REVOCATION | 4 |
| GCS_C_REASON_FOR_REVOCATION | 5 |
| GCS_C_SPLIT_PROTOCOL_TYPE | 6 |
| GCS_C_KEY_PART_NUMBER | 7 |
| GCS_C_NUMBER_OF_KEY_PARTS | 8 |
| GCS_C_KEY_VALIDITY_ACTIVATION_TIME | 9 |
| GCS_C_KEY_VALIDITY_QUIESCENT_TIME | 10 |
| GCS_C_KEY_VALIDITY_QUIESCENT_COUNT | 11 |
| GCS_C_KEY_VALIDITY_QUIESCENT_BYTES | 12 |
| GCS_C_KEY_VALIDITY_DEACTIVATE_TIME | 13 |
| GCS_C_KEY_VALIDITY_DEACTIVATE_COUNT | 14 |
| GCS_C_KEY_VALIDITY_DEACTIVATE_BYTES | 15 |
| GCS_C_IV | 16 |
| GCS_C_KEY_SPECIFIC_PARAMETERS | 17 |

**Table 9-15** Key Context Parameter Names

Chapter 10

# *Advanced CSF Application Program Interface (API)*

This chapter presents the functions comprising the advanced GCS-API. These are used by Cryptographic Policy Selecting Callers and Key Usage Policy Enforcing Callers.

In the majority of these definitions a cryptographic context is included as an input parameter providing information on the algorithm(s) and key(s) to be used in the function. A cryptographic context is also included as an output parameter because the CC may be modified by the call, eg., usage counts and key states may be modified any time the CC is used to provide a key used within a function. The check value of the CC and the validity period of a key within the CC are checked on each use of the CC.

**NAME**

gcs_advance_key_state — advances the key state of a cc

**SYNOPSIS**

```
OM_uint32 gcs_advance_key_state(
    OM_uint32                     *minor_status,
    gcs_session_context_t         *session_context,
    OM_uint32                     key_state,
    gcs_cc_t                      *subject_cc

);
```

**DESCRIPTION**

This function advances the key state of the cryptographic context, *subject_cc* thus permitting a caller to quiesce or deactivate a key before the transition is forced by the CSF based on time or number of cryptographic functions called. The function enables the caller to reduce key availability. The caller must possess the GCS_C_SELECTION authority, or the call will fail.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_advance_key_state*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*key_state* (in)
The required key state. Permitted values are GCS_QUIESCENT or GCS_DEACTIVATED.

*subject_cc* (opaque,in/out)
The cryptographic context of which the key state is to be advanced.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_COMPLETE_QCF]
Successful completion but *subject_cc* has quasi compromised flag set in key context.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
The *subject_cc* supplied is not valid.

[GCS_S_INCORRECT_KEY_STATE]
The *key_state* parameter value supplied is not one of the permitted values.

[GCS_S_INVALID_STATE_TRANSITION]
The key state transition requested is not permitted.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not recognised

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required GCS authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_archive_cc — transform a cryptographic context into an archive format

**SYNOPSIS**

```
OM_uint32 gcs_archive_cc(
    OM_uint32               *minor_status,
    gcs_session_context_t   *session_context,
    gcs_cc_t                *subject_cc,
    gcs_cc_t                *archive_kek_cc,
    gcs_bit_string_t         archive_string
);
```

**DESCRIPTION**

The *gcs_archive_cc* function transforms the cryptographic context, *subject_cc*, into an archive format as a bit string. The caller is responsible for storing the key, transformed by this function, in the archive. The caller must possess the GCS_C_KEY_USAGE GCS authority, or the call will fail.

If successful, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_archive_cc*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*subject_cc* (opaque,in)
The subject to be archived.

*archive_kek_cc* (optional,opaque,in/out)
The CC containing the key encryption key to be used in the archive process. If not defined, the CSF uses the default archive_kek.

*archive_string* (out)
The *subject_cc* is returned as an encrypted bit string for archive. The format of the bit string is defined by the implementation. The GCSAPI specification does not support the interoperability of archive formats between different implementations of the CSF.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_COMPLETE_QCF]
Successful completion but *archive_kek_cc* has quasi compromised flag set in key context.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_ARCHIVE_CC]
The *archive_kek_cc* supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
The *subject_cc* supplied is not valid.

[GCS_S_INCORRECT_KEY_STATE]
The *key_state* in the archive_kek_cc supplied does not permit the requested action.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required GCS authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

## NAME

gcs_combine_key — combine key parts

## SYNOPSIS

```
OM_uint32 gcs_combine_key(
    OM_uint32              *minor_status,
    gcs_session_context_t  *session_context,
    OM_uint32              key_part_flag,
    gcs_bit_string_t        key_part,
    gcs_cc_t               *kek_cc,
    gcs_cc_t               *combine_cc
);
```

## DESCRIPTION

This function is called recursively to build up a key in *combine_cc* The key part is in importable form protected by the *kek_cc*. *combine_cc* includes a split_protocol_type to indicate how the input bit string is encoded. The function returns the cc with the combined key values in the *combine_cc* supplied. The caller must possess the GCS_C_SELECTION GCS authority or the call will fail.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_combine_key*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*key_part_flag* (in)
: The *key_part_flag* specifies whether this is the first, subsequent last, or only call to the function. It may take on the values GCS_FIRST, GCS_MIDDLE, GCS_LAST or GCS_ONLY.

*key_part* (in)
: The part of the key to be combined with the key part contained in *combine_cc*.

*kek_cc* (opaque,in/out)
: The key encrypting key under which *key_part* is protected.

*combine_cc* (opaque,in/out)
: A cryptographic context supplied and into which the combined key parts are placed. The split protocol type is specified by *combine_cc*.

## RETURN VALUE

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
: Successful completion.

[GCS_S_CONTINUE_NEEDED]
: Another call to the function is required.

[GCS_S_BAD_SESSION_CONTEXT]
: The session context supplied is not valid.

[GCS_S_SUBJECT_CC]
: The *combine_CC* supplied is not valid.

[GCS_S_KEK_CC]
The *kek_cc* supplied is not valid.

[GCS_S_KEY_PART]
The key part supplied is not valid.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required GCS authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_create_ac — creates an empty algorithm context

**SYNOPSIS**

```
OM_uint32 gcs_create_ac(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    gcs_ac_t                     *ac
);
```

**DESCRIPTION**

This function creates an empty algorithm context which is returned in *ac* allocating memory as necessary.

Once created, its fields can be set, using *gcs_set_ac* and then supplied as a parameter to *gcs_create_cc* to create a cryptographic context.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_create_ac*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this context are required to support uses such as continuous I&A and authorisation.

*ac* (out)
: The algorithm context created.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
: Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
: The session context supplied is not valid.

[GCS_S_FAILURE]
: An implementation specific error or failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_create_cc — create a cryptographic context

**SYNOPSIS**

```
OM_uint32 gcs_create_cc(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    gcs_boolean_t                cc_confidentiality,
    gcs_ac_t                     *non_keyed_ac,
    gcs_ac_t                     *keyed_ac,
    gcs_kc_t                     *kc,
    gcs_cc_t                     *output_CC
);
```

**DESCRIPTION**

This function creates a cryptographic context from the input parameters supplied. The caller specifies the specific algorithm contexts, and key context required.

The cryptographic context created is returned in *output_CC*

The cryptographic context created is used in subsequent calls to the CSF. The caller must possess the GCS_C_KEY_USAGE GCS authority or the call will fail.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_create_cc*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*cc_confidentiality* (optional,in)
The flag specifying if the key used to eventually populate the cc is to be protected for confidentiality.

*non_keyed_ac* (in)
A non-keyed algorithm context previously created by *gcs_create_ac* and set by *gcs_set_ac*. NULL may be specified.

*keyed_ac* (in)
A keyed algorithm context previously created by *gcs_create_ac* and set by *gcs_set_ac*. NULL may be specified

*kc* (in)
An unkeyed key context previously created by *gcs_create_kc* and set with key context parameters by *gcs_set_kc*. NULL may be specified.

*output_CC* (opaque,out)
The resulting unkeyed cryptographic context.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_AC]
An algorithm context supplied is not valid.

[GCS_S_BAD_CONFIDENTIALITY_FLAG]
The confidentiality flag may be invalid.

[GCS_S_BAD_KC]
The key context supplied is not valid.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required GCS authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_create_kc — create an empty key context

**SYNOPSIS**

```
OM_uint32 gcs_create_kc(
    OM_uint32                 *minor_status,
    gcs_session_context_t     *session_context,
    gcs_kc_t                  *kc
);
```

**DESCRIPTION**

This function creates an empty key context which is returned in *kc*, allocating memory as necessary.

The key context may be set by *gcs_set_kc* and supplied as a parameter to *gcs_create_cc* to create a cryptographic context.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_create_kc*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this context are required for to support uses such as continuous I&A and authorisation.

*kc* (opaque,out)
The key context created.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_delete_ac — deletes an algorithm context

```
OM_uint32 gcs_delete_ac(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    gcs_ac_t                     *ac
);
```

**DESCRIPTION**

This function deletes the caller's copy of the algorithm context referred to as *ac*, frees the memory allocated to it and sets the *ac* pointer to GCS_NULL.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_delete_ac*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this context are required to support uses such as continuous I&A and authorisation.

*ac* (in/out)
: The algorithm context to be deleted.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
: Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
: The session context supplied is not valid.

[GCS_S_BAD_AC]
: The algorithm context supplied is not a valid algorithm context.

[GCS_S_FAILURE]
: An implementation specific error or failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_delete_kc — deletes a key context

**SYNOPSIS**

```
OM_uint32 gcs_delete_kc(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    gcs_kc_t                     *kc
);
```

**DESCRIPTION**

This function deletes the caller's copy of the key context input as *kc*, frees its memory allocation and sets the *kc* pointer to GCS_NULL.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_delete_kc*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this context are required to support uses such as continuous I&A and authorisation.

*kc* (in/out)
The key context to be deleted.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_KC]
The key context supplied is not a valid key context.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

**ERRORS**

No other errors are defined.

## NAME

gcs_generate_key_pattern — generate a test pattern for the supplied key

## SYNOPSIS

```
OM_uint32 gcs_generate_key_pattern(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    OM_uint32                    TPG_id,
    gcs_cc_t                     *subject_cc,
    gcs_buffer_t                  test_string
);
```

## DESCRIPTION

The *gcs_generate_key_pattern* function generates a key test pattern for the key contained within or referenced by *subject_cc*. The test pattern is used to verify the compatibility of keys derived by different implementations using the same input parameters. See Appendix E on page 221. The test pattern is output in *test_string*. The caller must possess the GCS_C_SELECTION GCS authority.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_generate_key_pattern*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*TPG_id* (in)
: The test pattern generator identifier.

*subject_cc* (opaque,in/out)
: The cryptographic context containing the key for which a key pattern is to be generated.

*test_string* (out)
: A character string containing the key pattern generated by the function.

## RETURN VALUE

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
: Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
: The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
: The cryptographic context *subject_cc* supplied is not valid.

[GCS_S_BAD_TPG]
: The test pattern generator identifier supplied is not valid.

[GCS_S_FAILURE]
: An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
An authorisation failure has occurred.

**ERRORS**

No other errors are defined.

## NAME

gcs_get_cc — get fields from the cryptographic context

## SYNOPSIS

```
OM_uint32 gcs_get_cc(
    OM_uint32                 *minor_status,
    gcs_session_context_t     *session_context,
    OM_uint32                  subject_container,
    OM_uint32                  parameter_name,
    gcs_cc_t                  *subject_cc,
    OM_uint32                 *parameter_integer_value,
    gcs_bit_string_t           parameter_bit_string_value
);
```

## DESCRIPTION

This function uses the *subject_container* field to determine from which of the *cc_header*, *non_keyed_ac*, *keyed_ac* or *key_context* sets of data a value is to be retrieved. It gets the value of the cryptographic context field specified by the *parameter_name* and places the value in *parameter_integer_value* or *parameter_bit_string_value* as appropriate.

Calls to *gcs_get_cc* only get a single field of the crypto context *subject_cc* per call. Any algorithm specific parameters returned are defined by BER encoding as specified in the standard that defines the object ID. This function does not return the key value. If successful, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_get_cc*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this context are required to support uses such as continuous I&A and authorisation.

*subject_container* (in)
A field specifying the data structure to be queried. It may be either the *crypto context header*, the *non-keyed algorithm context*, the *keyed algorithm context*, or the *key context*.

*parameter_name* (in)
The name of the field in the context specified by *input_container* to get.

*subject_cc* (opaque,in/out)
The cryptographic context to be queried.

*parameter_integer_value* (out)
The integer value of *parameter_name* retrieved by the call. This parameter is set to NULL if a bit_string value is returned.

*parameter_bit_string_value* (out)
The bit_string value of *parameter_name* retrieved by the call. This parameter is set to NULL if an integer value is returned.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_COMPLETE_QCF]
Successful completion but quasi-compromise flag is set in key context of *subject_cc*.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
The subject cc supplied is not valid.

[GCS_S_BAD_SUBJECT_CONTAINER]
The subject container supplied is not valid.

[GCS_S_BAD_PARAMETER]
The parameter name supplied is not valid.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required GCS authority, or some other authorisation failure has occurred. For example, the caller has requested a modification to a field that the caller is not authorised to set.

**ERRORS**

No other errors are defined.

**NAME**

gcs_get_key_validity — get key validity information.

**SYNOPSIS**

```
OM_uint32 gcs_get_key_validity
    OM_uint32              *minor_status,
    gcs_session_context_t  *session_context,
    gcs_cc_t               *subject_cc,
    OM_uint32              validity_format,
    OM_uint32              *activation_value,
    OM_uint32              *quiescent_value,
    OM_uint32              *deactivation_value
   );
```

**DESCRIPTION**

This function returns the key validity values held within the key context of the *subject_cc*.

If successful, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_get_key_validity* ( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this context are required to support uses such as continuous I&A and authorisation.

*subject_cc* (opaque,in/out)
The cryptographic context supplied for which the key validity is required.

*validity_format* (in)
Specified whether the *quiescent_value* and the *deactivation_value* are set in terms of:

- GCS_C_TIME number of seconds from current time (an absolute time value may be converted to a relative time by subtracting current time from it), or
- GCS_C_COUNT of cryptographic functions called (overwrite existing values.)
- GCS_C_BYTES, number of bytes of data processed by cryptographic function calls.

For a populated CC RELATIVE_TIME is relative to the current CSF time. For a template CC RELATIVE_TIME is relative to the time of population with a key.

*activation_value* (out)
For a populated CC the number of seconds relative to current CSF time after which the key state is to be set to GCS_ACTIVE. For a template CC the number of seconds relative to the subsequent time of population of the CC after which the key state is to be set to GCS_ACTIVE. GCS_C_NOW and GCS_C_INFINITE may be specified. This parameter may only be input as a number of seconds.

*quiescent_value* (out)
The number of seconds, or the number of calls to cryptographic functions using *subject_cc*, or the number of bytes processed by calls using *subject_cc* after which the key state is to be set to GCS_QUIESCENT. GCS_C_NOW and GCS_C_INFINITE may be specified.

*deactivation_value* (out)
The number of seconds, or the number of calls to cryptographic functions using *subject_cc*,

or the number of bytes processed by calls using *subject_cc* after which the key state is to be set to GCS_DEACTIVATED. GCS_C_NOW and GCS_C_INFINITE may be specified.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_COMPLETE_QCF]
Successful completion but *subject_cc* has quasi compromised flag set in key context.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
The cryptographic context supplied is not valid.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
An authorisation failure has occurred.

**ERRORS**

No other errors are defined.

## NAME

gcs_load_public_key — load a clear public key or key part

## SYNOPSIS

```
OM_uint32 gcs_load_public_key(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    gcs_cc_t                     *subject_cc,
    gcs_bit_string_t              input_key_part,
    OM_uint32                     key_part_type
);
```

## DESCRIPTION

The *gcs_load_public_key* function loads a clear public key, or key part, into *subject_cc*.

A separate call to *gcs_store_cc* needs to be made if the key is to be retained within the CSF. The caller must possess the GCS_C_KEY_USAGE GCS authority.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_load_public_key* ( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*subject_cc* (opaque,in, out)
The template CC, or partially populated cryptographic context into which the key, or key part, is to be loaded. The *subject_CC* includes the split protocol type indicating which mechanism is to be used to combine key parts, if the key is loaded in parts. The function returns the cryptographic context with key value updated as appropriate.

*input_key_part* (in)
The key part.

*key_part_type* (in)
This may be defined as GCS_FIRST, GCS_MIDDLE, GCS_LAST or GCS_ONLY.

## RETURN VALUE

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
The cryptographic context subject_cc supplied is not valid.

[GCS_S_BAD_PART]
The key part specified is not valid.

[GCS_S_INCORRECT_KEY_STATE]
The key state in the cc supplied does not permit the requested action.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required GCS authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_reduce_key_usage — reduce usage of the cryptographic context

**SYNOPSIS**

```
OM_uint32 gcs_reduce_key_usage(
    OM_uint32                     *minor_status,
    gcs_session_context_t         *session_context,
    OMuint32                      key_usage,
    gcs_cc_t                      *subject_cc
);
```

**DESCRIPTION**

This function reduces the usage of the cryptographic context *subject_cc* supplied. The original *key_usage* bit mask can be retrieved from *subject_cc* by a call to *gcs_get_cc*( ) and then modified prior to being reinput to this function. The caller must possess the GCS_C_SELECTION GCS authority.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_reduce_key_usage*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*key_usage* (in)
The usage to which the key is put. It is used to modify the cryptographic context.

*subject_cc* (opaque,in/out)
The cryptographic context supplied. It is returned with a modified key usage.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_BAD_SESSION_CC]
The session_context supplied is not valid.

[GCS_S_BAD_SUBJECT_CONTEXT]
The cryptographic context supplied is not valid.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_BAD_KEY_USAGE]
The key usage supplied is not valid.

[GCS_S_AUTHORISATION_FAILURE]
An authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_restore_cc — transform an archive bit string to a cryptographic context

**SYNOPSIS**

```
OM_uint32 gcs_restore_cc(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    gcs_cc_t                     *archive_kek,
    gcs_bit_string_t              archive_string ,
    gcs_cc_t                     *restored_cc
);
```

**DESCRIPTION**

The *gcs_restore_cc* function transforms the input archive string, *archive_string*, decrypted from an archive format bit string to an operational format cryptographic context, *restored_cc*. The caller must possess the GCS_C_SELECTION or GCS_C_KEY_USAGE GCS authority, or the call will fail.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_restore_cc*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*archive_kek* (optional, opaque, in/out)
The CC that contains the key encryption key to be used to process the input bit string. If not defined, the CSF uses the default archive-kek.

*archive_string* (in)
The bit string in archive format to be restored using *archive_kek.*

*restored_cc* (opaque,out)
The cryptographic context represented by the input archive string is output in an operational format.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_COMPLETE_QCF]
Successful completion but *archive_kek* has quasi compromised flag set in key context.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_ARCHIVE_CC]
The archive key encryption key supplied is not valid.

[GCS_S_BAD_ARCHIVE_STRING]
The archive string supplied could not be used to restore a CC.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required GCS authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_revoke_key — change the key state to revoked

**SYNOPSIS**

```
OM_uint32 gcs_revoke_key(
    OM_uint32                   *minor_status,
    gcs_session_context_t       *session_context,
    gcs_cc_t                    *subject_cc,
    gcs_buffer_t                 reason
);
```

**DESCRIPTION**

This function changes the key state in the cryptographic context supplied to REVOKED for the *reason* for revocation supplied. It is used when a key is found to be compromised. The cryptographic context for which the key is revoked is disabled. After this call the time of revocation in the key context is set to the time of invocation of this function and the reason for revocation is set to the string given in *reason*. Note that the *reason* is restricted to a maximum length of 80 characters.

The caller must possess either or both of GCS_C_SELECTION or GCS_KEY_USAGE authorities. If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_revoke_key*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*subject_cc* (opaque,in, out)
: The cryptographic context for which the key is to be revoked.

*reason* (in)
: The reason why the key is to be revoked which is constrained to be less than 80 characters in length.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
: Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
: The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
: The cryptographic context reference subject_cc supplied does not refer to a valid cryptographic context.

[GCS_S_FAILURE]
: An implementation specific error or failure has occurred.

[GCS_S_BAD_REASON]
: The reason given for revocation is not valid.

[GCS_S_INCORRECT_KEY_STATE]
The key state is already revoked.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required GCS authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_set_ac — set fields in the algorithm context

**SYNOPSIS**

```
OM_uint32 gcs_set_ac(
    OM_uint32                      *minor_status,
    gcs_session_context_t          *session_context,
    OM_uint32                       parameter_name,
    OM_uint32                       parameter_integer_value,
    gcs_bit_string_t                parameter_bit_string_value,
    gcs_ac_t                       *ac
);
```

**DESCRIPTION**

This function sets or overwrites the algorithm context field specified by the *parameter_name* to the value specified in *parameter_integer_value* or *parameter_bit_string_value*.

Algorithm specific parameters need to be defined by BER encoding as specified in the standard that defines the object ID.

Several calls to *gcs_set_ac* are required to set each field of the algorithm context. If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_set_ac*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
An implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*parameter_name* (in)
The name of the field in the algorithm context to set. All algorithm specific parameters must be supplied in a single call to *gcs_set_ac*( ). These are interpreted in the context of the algorithm identity which must have been set by a previous call to *gcs_set_ac*( ).

*parameter_integer_value* (in)
The integer value to which the *parameter_name* is to be set. This parameter is set to NULL if a bit string value is to be set.

*parameter_bit_string_value* (in)
The bit-string to which the *parameter_name* is to be set. This parameter is set to NULL if an integer value is to be set.

*ac* (opaque,in/out)
The algorithm context to be populated.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_AC]
The algorithm context supplied is not valid.

[GCS_S_BAD_PARAMETER]
The parameter name supplied is not valid.

[GCS_S_BAD_PARAM_VALUE]
The parameter value supplied is not consistent with the parameter value supplied or with the existing contents of the algorithm context.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

**ERRORS**

No other errors are defined.

## NAME

gcs_set_cc — set fields in the cryptographic context

## SYNOPSIS

```
OM_uint32 gcs_set_cc(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    OM_uint32                    subject_container,
    OM_uint32                    parameter_name,
    OM_uint32                     parameter_integer_value,
    gcs_bit_string_t              parameter_bit_string_value,
    gcs_cc_t                     *subject_cc
);
```

## DESCRIPTION

This function uses the *subject_container* field to determine which of the *cc_header*, *non_keyed_ac*, *keyed_ac* or *key_context* sets of data is to be modified. It sets the cryptographic context field specified by the *parameter_name* to the value specified in *parameter_integer_value* or *parameter_bit_string_value*.

Calls to *gcs_set_cc* only set a single field of the crypto context *subject_cc*. Algorithm specific parameters and key specific parameters need to be defined by BER encoding as specified in the standard that defines the object ID.

The caller must possess the GCS_C_KEY_USAGE GCS authority. If successful, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_set_cc*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this context are required to support uses such as continuous I&A and authorisation.

*subject_container* (in)
: A field specifying the data structure to populate. It may be either the *crypto context header*, the *non-keyed algorithm context*, the *keyed algorithm context*, or the *key context*.

*parameter_name* (in)
: The name of the field in the context specified by *input_container* to set.

*parameter_integer_value* (in)
: The integer value to which the *parameter_name* is to be set. This parameter is set to NULL if a bit string value is to be set.

*parameter_bit_string_value* (in)
: The bit_string to which the *parameter_name* is to be set. This parameter is set to NULL if an integer value is to be set.

*subject_cc* (opaque,in/out)
: The cryptographic context to be modified.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_COMPLETE_QCF]
Successful completion but quasi-compromise flag is set in key context of *subject_cc*.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
The subject cc supplied is not valid.

[GCS_S_BAD_SUBJECT_CONTAINER]
The subject container supplied is not valid.

[GCS_S_BAD_PARAMETER]
The parameter name supplied is not valid.

[GCS_S_BAD_PARAM_VALUE]
The parameter value supplied is not consistent with the parameter name supplied.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required GCS authority or some other authorisation failure has occurred. For example, the caller has requested a modification to a field that the caller is not authorised to set.

**ERRORS**

No other errors are defined.

## NAME

gcs_set_kc — set fields in the key context

## SYNOPSIS

```
OM_uint32 gcs_set_kc(
    OM_uint32              *minor_status,
    gcs_session_context_t  *session_context,
    OM_uint32              parameter_name,
    OM_uint32               parameter_integer_value,
    gcs_bit_string_t         parameter_bit_string_value,
    gcs_kc_t               *kc
);
```

## DESCRIPTION

This function sets the key context field specified by the *parameter_name* to the value specified in *parameter_integer_value* or *parameter_bit_string_value*.

Calls to *gcs_set_kc* only set a single field of the key context per call. Key specific parameters need to be defined by BER encoding as specified in the standard that defines the algorithm.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_set_kc*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this context are required to support uses such as continuous I&A and authorisation.

*parameter_name* (in)
: The name of the field in the key context to set. All key specific parameters must be supplied in a single call to *gcs_set_kc*( ) in a BER encoded format.

*parameter_integer_value* (in)
: The integer value to which the *parameter_name* is to be set. If the parameter value required is a bit_string then this parameter is to be set to NULL.

*parameter_bit_string_value* (in)
: The bit_string value to which the *parameter_name* is to be set. If the parameter value required is an integer_value then this parameter is to be set to NULL.

*kc* (in/out)
: The key context to be populated.

## RETURN VALUE

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
: Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
: The session context supplied is not valid.

[GCS_S_BAD_KC]
: The key context supplied is not valid.

[GCS_S_BAD_PARAMETER]
The parameter name supplied is not valid.

[GCS_S_BAD_PARAM_VALUE]
The parameter value supplied is not consistent with the parameter value supplied.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

**ERRORS**

No other errors are defined.

## NAME

gcs_set_key_validity — set the key validity information.

## SYNOPSIS

```
OM_uint32 gcs_set_key_validity
    OM_uint32              *minor_status,
    gcs_session_context_t  *session_context,
    OM_uint32               validity_format,
    OM_uint32               activation_value,
    OM_uint32               quiescent_value,
    OM_uint32               deactivation_value,
    gcs_cc_t               *subject_cc
);
```

## DESCRIPTION

The *gcs_set_key_validity*( ) function changes the key validity values held within the key context of the *subject_cc*. The caller requires the GCS_C_KEY_USAGE GCS authority.

This call may be used to modify the key validity policy of a locally referenced CC including reactivating a deactivated key, for example when restored from an archive for the purposes of verifying a signature on some historic information.

The key validity values of a stored CC are not modified unless the caller possesses an exclusive access lock to the CC and makes a subsequent call to *gcs_store_cc*( ) to update the stored CC and release the exclusive access lock.

If successful, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_set_key_validity*( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this context are required to support uses such as continuous I&A and authorisation.

*validity_format* (in)
Specifies whether the *quiescent_value* and the *deactivation_value* supplied are in terms of:

- GCS_C_TIME number of seconds from current time (an absolute time value may be converted to a relative time by subtracting current time from it), or
- GCS_C_COUNT of cryptographic functions called (overwrite existing values.)
- GCS_C_BYTES, number of bytes of data processed by cryptographic function calls.

For a populated CC RELATIVE_TIME is relative to the current CSF time. For a template CC RELATIVE_TIME is relative to the time of population with a key.

*activation_value* (in)
For a populated CC the number of seconds relative to current CSF time after which the key state is to be set to GCS_ACTIVE. For a template CC the number of seconds relative to the subsequent time of population of the CC after which the key state is to be set to GCS_ACTIVE. GCS_C_NOW and GCS_C_INFINITE may be specified. This parameter may only be input as a number of seconds.

*quiescent_value* (in)
The number of seconds, or the number of calls to cryptographic functions using *subject_cc*, or the number of bytes processed by calls using *subject_cc* after which the key state is to be set to GCS_QUIESCENT. GCS_C_NOW and GCS_C_INFINITE may be specified.

*deactivation_value* (in)
The number of seconds, or the number of calls to cryptographic functions using *subject_cc*, or the number of bytes processed by calls using *subject_cc* after which the key state is to be set to GCS_DEACTIVATED. GCS_C_NOW and GCS_C_INFINITE may be specified.

*subject_cc* (opaque,in/out)
The cryptographic context supplied is returned with the key validity values changed as specified in *key_state*. If appropriate the key state will also have been changed if the new key validity values are inconsistent with the initial key state when the call is made.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_COMPLETE_QCF]
Successful completion but *subject_cc* has quasi compromised flag set in key context.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_PARAMETER]
The time offset or one or more validity periods is invalid, or both.

[GCS_S_BAD_SUBJECT_CC]
The cryptographic context supplied is not valid.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

## NAME

gcs_verify_key_pattern — verify the supplied key against a key test pattern string

## SYNOPSIS

```
OM_uint32 gcs_verify_key_pattern(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    gcs_buffer_t                  test_string,
    OM_uint32                    *TPG_id,
    gcs_cc_t                     *subject_cc
);
```

## DESCRIPTION

The *gcs_verify_key_pattern* function verifies a key contained within or referenced by *subject_CC* against the specified key test pattern, *test_string*. The caller must possess the GCS_C_SELECTION GCS authority.

If the key pattern is verified, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_verify_key_pattern* ( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*test_string* (in)
: A character string containing the key pattern generated by a previous call to *gcs_generate_key_pattern*.

*TPG_id* (in)
: The test pattern generator identifier to be used.

*subject_cc* (opaque,in/out)
: The cryptographic context containing the key for which a key pattern is to be verified.

## RETURN VALUE

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
: Successful completion.

[GCS_S_COMPLETE_QCF]
: Successful completion but *subject_cc* has quasi compromised flag set in key context.

[GCS_S_BAD_SESSION_CONTEXT]
: The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
: The cryptographic context supplied is not valid.

[GCS_S_BAD_TPG]
: The test pattern generator identifier supplied is not valid.

[GCS_S_NO_VERIFY]
The key pattern verification has failed.

[GCS_S_INCORRECT_KEY_STATE]
The key state in the CC supplied does not permit the requested action.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required GCS authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

*Chapter 11*

# *Advanced CSF System Programming Interfaces (SPIs)*

This chapter presents those functions that are restricted to use by Cryptographic Policy Enforcing callers.

## NAME

gcs_decipher_key — decipher a key

## SYNOPSIS

```
OM_uint32 gcs_decipher_key(
    OM_uint32              *minor_status,
    gcs_session_context_t  *session_context,
    gcs_cc_t               *kek_cc,
    gcs_buffer_t            enciphered_key,
    gcs_buffer_t            IV,
    gcs_buffer_t            clear_key
);
```

## DESCRIPTION

The *gcs_decipher_key* function is used to transform an enciphered key and key related data input as *enciphered_key* and output as *clear_key* using the algorithm and key specified by *kek_cc*. It is distinguished from *gcs_decipher_data* by constraints on the size of data that may be deciphered, or the speed at which it may be deciphered.

The *gcs_decipher_key* function is provided to support existing key distribution implementations. It is only needed if the caller cannot achieve key transport or key agreement using *gcs_export_key* and *gcs_import_key* or *gcs_export_key_agreement* and *gcs_import_key_agreement*.

Applications may need to prefix keys with confounders according to the appropriate protocol.

It is up to the caller to protect clear keys. The caller must possess the GCS_C_KEY_PROTECTION and the GCS_C_ENCIPHER_DECIPHER authorities.

If successful, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_decipher_key*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contenst of this session context are required to support uses such as continuous I&A and authorisation.

*kek_cc* (opaque,in, out)
: The cryptographic context containing the key encryption key algorithms and other key information needed to decipher the key.

*enciphered_key* (in)
: The enciphered key to be deciphered.

*IV* (in)
: The optional initialisation vector.

*clear_key* (out)
: The key is deciphered and returned in clear form in *clear_key*.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_COMPLETE_QCF]
Successful completion but kek_cc has the quasi compromised flag set in its key context.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_KEK_CC]
The kek_cc cryptographic context supplied is not valid.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

## NAME

gcs_derive_clear_key — derive a secret key from the key string supplied

## SYNOPSIS

```
OM_uint32 gcs_derive_clear_key(
    OM_uint32                    *minor_status,
    gcs_session_context_t        *session_context,
    gcs_bit_string_t              key_string,
    gcs_cc_t                     *kgk_cc,
    gcs_cc_t                     *subject_cc
);
```

## DESCRIPTION

The *gcs_derive_clear_key* function derives a secret key from *key_string*.

The algorithm, key size, key usage and other parameters associated with the cryptographic context are specified in *subject_cc*.

The derived key will be unprotected. If the context confidentiality flag is not set to "NO", the call will fail. The key is output within the key context part of *subject_cc*.

Note that the caller is responsible for the protection of clear keys.

The caller must possess the GCS_C_KEY_PROTECTION authority. If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_derive_clear_key*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*key_string* (in)
: The key string used as the basis for deriving a secret key.

*kgk_cc* (optional, in, out)
: When supplied this references the cryptographic context used to derive a key using the derivation mechanism specified in the algorithm context of *kgk_cc*.

*subject_cc* (opaque,in/out)
: The subject_cc cryptographic context supplied is populated to include the secret key derived by *gcs_derive_clear_key* and returned.

## RETURN VALUE

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
: Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
: The session context supplied is not valid.

[GCS_S_BAD_KGK_CC]
: The key generating key cryptographic context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
The cryptographic context subject_cc supplied is not valid.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_encipher_key — encipher a key

**SYNOPSIS**

```
OM_uint32 gcs_encipher_key(
    OM_uint32                  *minor_status,
    gcs_session_context_t      *session_context,
    gcs_cc_t                   *kek_cc,
    gcs_buffer_t                key_bit_string,
    gcs_buffer_t                IV,
    gcs_buffer_t                enciphered_key
);
```

**DESCRIPTION**

The *gcs_encipher_key* function is used to transform a clear key and key related data input in *key_bit_string* to an *enciphered_key* using the algorithm and key specified by *kek_cc*. It is distinguished from *gcs_encipher_data* by constraints on the size of data that may be enciphered, or the speed at which it may be enciphered.

The *gcs_encipher_key* function is provided to support existing key distribution implementations. It is only needed if the caller cannot invoke suitable key transport or key agreement services using *gcs_export_key* and *gcs_import_key* or *gcs_export_key_agreement* and *gcs_import_key_agreement*. That is to say, they are not supported export mechanisms of the CSF.

Applications may need to prefix keys with confounders according to the appropriate protocol. The caller must possess the GCS_C_KEY_PROTECTION and the GCS_C_ENCIPHER_DECIPHER authority.

If successful, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_encipher_key*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contenst of this session context are required to support uses such as continuous I&A and authorisation.

*kek_cc* (opaque,in, out)
: The cryptographic context containing the key encryption key algorithms and other key information needed to encipher the key.

*key_bit_string* (in)
: The clear key bit string to be enciphered.

*IV* (in)
: The optional initialisation vector.

*enciphered_key* (out)
: The enciphered key is returned.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_COMPLETE_QCF]
Successful completion but kek_cc has quasi compromised flag set in its key context.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_KEK_CC]
The kek_cc cryptographic context supplied is not valid.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

## NAME

gcs_generate_clear_key — generate a secret key or a public and private key pair in the clear

## SYNOPSIS

```
OM_uint32 gcs_generate_clear_key(
    OM_uint32                  *minor_status,
    gcs_session_context_t      *session_context,
    gcs_cc_t                   *subject_cc
);
```

## DESCRIPTION

The *gcs_generate_clear_key* function generates a secret key or public and private key pair in the clear and outputs them in *subject_cc*. The algorithm, key size, key usage, and other associated parameters are specified by the input *subject_cc*.

Note that the caller is responsible for the protection of clear keys.

The call will fail if the context confidentiality flag in the *subject_cc* is not set to "NO". The caller must possess the GCS_C_KEY_PROTECTION authority.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_generate_clear_key*( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*subject_cc* (opaque,in/out)
: The cryptographic context defining algorithm, key size, and key usage. The cryptographic context is returned populated with the clear key.

## RETURN VALUE

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
: Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
: The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
: The subject_cc cryptographic context supplied is not valid.

[GCS_S_RNG_NOT_INITIALISED]
: The CSF random number generator has not been initialised.

[GCS_S_FAILURE]
: An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
: The caller does not possess the required authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

**NAME**

gcs_load_key — load a clear key or key part

**SYNOPSIS**

```
OM_uint32 gcs_load_key(
    OM_uint32                   *minor_status,
    gcs_session_context_t       *session_context,
    gcs_cc_t                    *subject_cc,
    gcs_bit_string_t             input_key_part,
    OM_uint32                    key_part_type
);
```

**DESCRIPTION**

The *gcs_load_key* function loads a clear key, or key part, into *subject_cc*.

A separate call to *gcs_store_cc* needs to be made if the key is to be retained within the CSF. The caller must possess the GCS_C_KEY_PROTECTION authority.

If successful, the function returns [GCS_S_COMPLETE].

The arguments for *gcs_load_public_key* ( ) are:

*minor_status* (out)
An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*subject_cc* (opaque,in, out)
The unpopulated, or partially populated cryptographic context into which the key, or key part, is to be loaded. The function returns the cryptographic context with key value updated as appropriate.

*input_key_part* (in)
The key part.

*key_part_type* (in)
This may be defined as GCS_FIRST, GCS_MIDDLE, GCS_LAST or GCS_ONLY.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
The cryptographic context subject_cc supplied is not valid.

[GCS_S_BAD_PART]
The key part specified is not valid.

[GCS_S_INCORRECT_KEY_STATE]
The key state in the cc supplied does not permit the requested action.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

## NAME

gcs_split_clear_key — split a clear key into several parts

## SYNOPSIS

```
OM_uint32 gcs_split_key(
    OM_uint32               *minor_status,
    gcs_session_context_t   *session_context,
    gcs_cc_t                *subject_cc,
    OM_uint32                n,
    OM_uint32                k,
    OM_uint32                split_protocol_type,
    gcs_bit_string_set_t     output_key
);
```

## DESCRIPTION

The *gcs_split_clear_key* function splits the input key contained in *subject_cc* into a number of parts specified in *n* and returns the split key in *output_key*. The maximum number of parts needed to reconstitute the key, *k*, and the maximum number of parts, *n*, are defined by the implementation. n >= k.

Note that the caller is responsible for the protection of the key parts. The caller must possess the GCS_C_KEY_PROTECTION authority

If successful, the function returns [GCS_S_COMPLETE] or [GCS_S_COMPLETE_QCF].

The arguments for *gcs_split_clear_key* ( ) are:

*minor_status* (out)
: An implementation specific return status that provides additional information when [GCS_S_FAILURE] is returned by the function.

*session_context* (opaque,in)
: The implementation specific parameter that defines the context of the current session between the caller and the CSF. The contents of this session context are required to support uses such as continuous I&A and authorisation.

*subject_cc* (opaque,in)
: The cryptographic context containing the key to be split.

*n* (in)
: The number of parts into which the key is to be split. The implementation defines the maximum size of n.

*k* (in)
: The number of parts needed to reconstitute the key. This is defined by the implementation, where n >=k.

*split_protocol_type* (in)
: The split protocol type. For example GCS_C_XOR and GCS_C_SHAMIR.

*output_key* (out)
: The key output as a set of n strings.

**RETURN VALUE**

The following GCS status codes shall be returned:

[GCS_S_COMPLETE]
Successful completion.

[GCS_S_COMPLETE_QCF]
Successful completion but subject_cc has the quasi compromised flag set in the key context.

[GCS_S_BAD_SESSION_CONTEXT]
The session context supplied is not valid.

[GCS_S_BAD_SUBJECT_CC]
The cryptographic context subject_cc supplied is not valid.

[GCS_S_BAD_SIZE]
The number of parts specified exceeds the implementation defined maximum.

[GCS_S_BAD_PROTOCOL]
The split protocol specified is not valid.

[GCS_S_INCORRECT_KEY_STATE]
The key state in the cc supplied does not permit the requested action.

[GCS_S_FAILURE]
An implementation specific error or failure has occurred.

[GCS_S_AUTHORISATION_FAILURE]
The caller does not possess the required authority or some other authorisation failure has occurred.

**ERRORS**

No other errors are defined.

Chapter 12

# Conformance Statement

## 12.1 GCS-API (Base) Conformance

This section defines conformance criteria for implementations of the GCS-API, and also mechanism-independent use of the GCS-API by applications.

The following GCS-API implementation conformance levels are defined:

- Basic GCS-API Minimal Implementation Conformance

  All Basic GCS-API functions but excluding user data encipherment functions.

  A minimally conforming implementation that supports multiple principals or separation of CCs shall provide support for an administrator to configure default behaviour to limit access to populated CCs to the principal or group of principals on whose behalf the CC has been populated.

- Basic GCS-API Restricted User Data Encipherment Option.

  User data encipherment is supported but using restricted strength algorithms.

- Basic GCS-API Unrestricted User Data Encipherment Option.

  User data encipherment is supported using full strength algorithms.

- Advanced GCS-API Option

  This includes the Advanced GCS-API functions excluding key test pattern and clear key management functions.

- Advanced GCS-API Key Test Pattern Option

  Support for key test pattern generation and verification is optional.

- Advanced GCS-API Clear Key Management Option

  This provides additional support for CSF management applications.

An implementation is required to specify identities of supported algorithms and export mechanisms. This should include the identity of standards (if any) in which they are defined. If these are proprietary and not otherwise defined in a referencable document, then the algorithm specific parameters and export mechanism protocols must be defined.

### 12.1.1 GCS-API (Base) Minimal Implementation

All conforming GCS-API (Base) implementations **shall** support the following interfaces:

| | |
|---|---|
| gcs_delete_key | gcs_derive_key |
| gcs_export_key | gcs_generate_check_value |
| gcs_generate_hash | gcs_generate_key |
| gcs-generate_random_number | gcs_get_csf_params |
| gcs_import_key | gcs_initialise_session |
| gcs_key_agreement | gcs_list_cc |
| gcs_release_buffer | gcs_remove_cc |
| gcs_retrieve_cc | gcs_store_cc |
| gcs_terminate_session | gcs_verify_checkvalue |
| gcs_release_bit_string | |

### 12.1.2 GCS-API (Base) Restricted User Data Encipherment Option

All conforming implementations that support the Restricted User Data Encipherement Option **shall** additionally support:

| | |
|---|---|
| gcs_decipher_data | gcs_decipher_verify |
| gcs_encipher_data | gcs_protect_data |

The conformance statement for an implementation shall state the restrictions to which these functions are subject within an implementation.

### 12.1.3 GCS-API (Base) Unrestricted User Data Encipherment Option

All conforming implementations that support the Unrestricted User Data Encipherement Option **shall** support:

| | |
|---|---|
| gcs_decipher_data | gcs_decipher_verify |
| gcs_encipher_data | gcs_protect_data |

These interfaces shall be unencumbered by any restrictions.

### 12.1.4 GCS-API (Base) Advanced Service Option

All conforming implementations that support the Advanced Service Option **shall** support the following interfaces:

| | |
|---|---|
| gcs_advance_key_state | gcs_archive_cc |
| gcs_clone_cc | gcs_combine_key |
| gcs_create_ac | gcs_create_cc |
| gcs_create_kc | gcs_delete_ac |
| gcs_delete_kc | gcs_get_cc |
| gcs_get_key_validity | gcs_load_public_key |
| gcs_reduce_key_usage | gcs_restore_cc |
| gcs_revoke_key | gcs_set_ac |
| gcs_set_cc | gcs_set_kc |
| gcs_set_key_validity | |

### 12.1.5 GCS-API (Base) Key Test Pattern Option

All conforming implementations that support the Test Pattern Option **shall** support:

| | |
|---|---|
| gcs_generate_key_pattern | gcs_verify_key_pattern |

### 12.1.6 GCS-API (Base) Clear key Management Option

All conforming implementations that support the Clear Key Management Option **shall** support:

| | |
|---|---|
| gcs_decipher_key | gcs_derive_clear_key |
| gcs_encipher_key | gcs_generate_clear_key |
| gcs_load_key | gcs_split_clear_key |

*Appendix A*

# *CSF Implementation Considerations*

## A.1 Legislative Constraints

Chapter 5 has introduced the concept of legislative constraints on the export or use of products containing cryptographic functions. This appendix describes some of the implementation implications of complying with such legislation.

Figure A-1 illustrates alternative placements for the legislative enforcing functions.

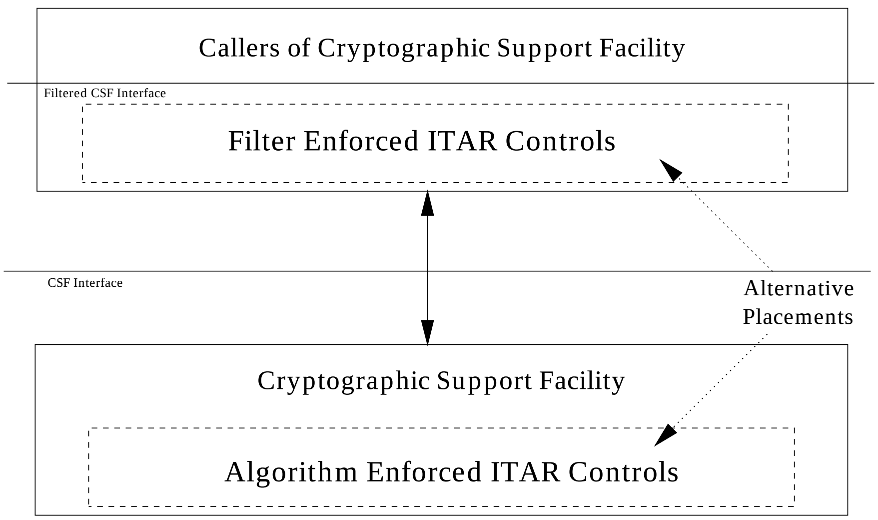


**Figure A-1** Legislative Controls within Cryptographic Support Facility

Legislative enforcing functions may be incorporated in the CSF or within callers of the CSF. If implemented within the CSF, the CSF need not provide the restricted services to callers at all, or it may impose limits on their use. If implemented within the caller of the CSF services, the CSF provides all cryptographic services to its callers, which are then trusted only to utilise the restricted services in an authorised manner. A combination of these alternatives may be deployed, dependent upon one of the following:

- a run-time determination of the caller's authorisation to use the restricted services
- a build-time constraint by restricting the availability of the libraries providing the interfaces to the restricted CSF services to developers of trusted applications.

Some consequences of these requirements are that:

- Conformance to a CSF specification must be compatible with achieving conformance to known domestic and export controls, although different CSF interface profiles may apply for different regulatory environments. For example, a confidentiality service could be an optional service that would not be supported by some conforming implementations.

- The operational CSF services identified in the document are split into:
  — general cryptographic services
  — protected key management services
  — clear key management services
- CSF management services are not within the scope of the document.
- For those CSF services identified to be legislative sensitive it should be possible to achieve compliance with export or usage rules through internal CSF controls (for example, by binding usage controls into algorithms), or through controls at the CSF interface layer (for example, a *legislative filter* as shown in Figure A-1 on page 177).
- Depending on the policy enforced, the CSF might require its callers to have been authenticated before they can access its services. A cryptographic product can therefore include authentication and authorisation services, as well as the management and operational cryptographic services.
- Once deposited beneath the CSF API, keys should never be referenced in the clear by unauthorised callers. Above the CSF interface operational keys are protected by enciphering with the CSF Master-Key. Authorised callers are trusted key distribution services that require to combine an operational key in the clear with other related information to create a mechanism-specific token. Also note that subversion of CSF access controls is more important for services related to key management than those related to applications.

## A.2 Technical Constraints

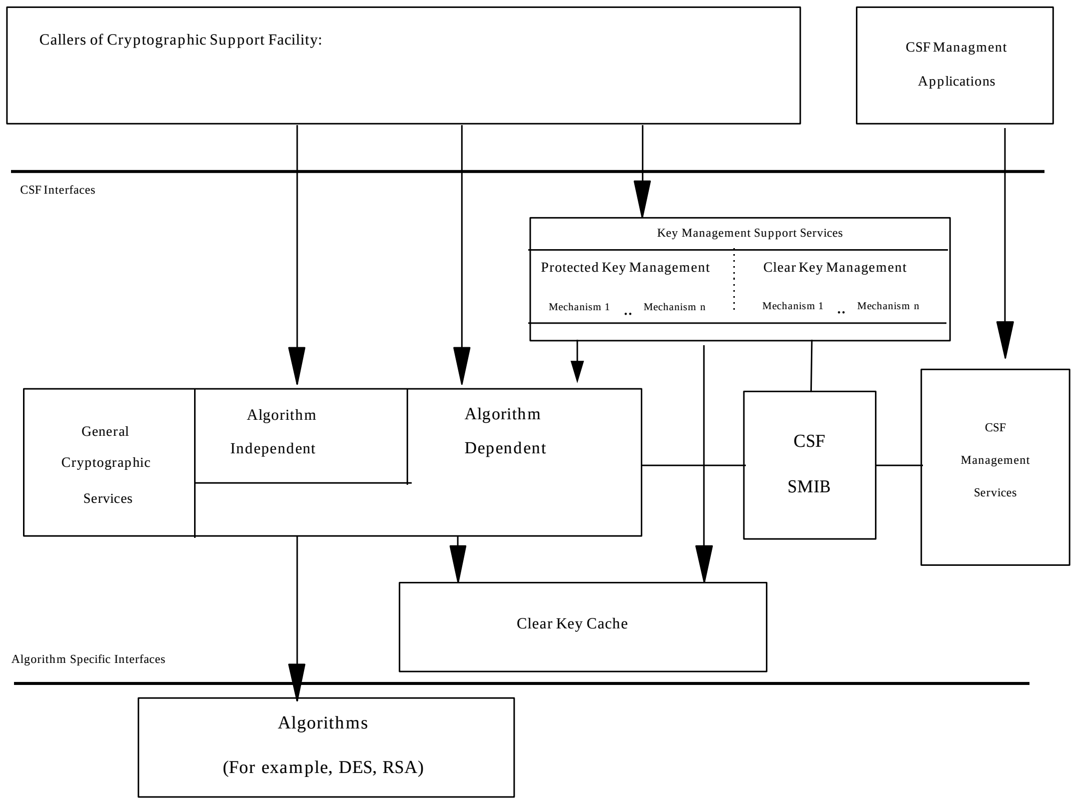


**Figure A-2** Cryptographic Support Facility

A logical structuring of a CSF is illustrated in Figure A-2. The CSF is implemented over interfaces to different algorithms and different implementations of those algorithms.

As cryptographic interfaces are often implemented in hardware, the CSF interfaces and constructs should not require implementations to maintain internal state information across API invocations.

The CSF services could be achieved by means of stateless transactions in which state information is provided as parameters (either by value or by reference) of the current API invocation, and not based on information retained from previous API invocations.

The CSF SMIB may be implemented within the unit that implements the CSF or may be implemented externally to the unit provided it is suitably protected.

In the case when a CSF supports the concurrent retrieval of a populated CC, stored by the CSF, for concurrent use by multiple callers then the usage statistics must be accumulated over all uses of the key within the stored CC. This may result in the triggering of a key state change arising from one callers use of the CC that results in a subsequent failure of a call by another caller using a copy of the same stored CC.

## A.3 Threat Model

### A.3.1 Types of threats

1. **Outsider/Insider**
Is the adversary an outsider or is the adversary a valid user of the system in some way. Thwarting insider threats is more difficult than thwarting outsider threats. Some outsider threats are passive, such as a wiretap monitoring the data, while others are active, such as use of a LAN sniffer to attempt to interject or modify data. In general insiders are assumed to have all the capabilities of outsiders and in addition insiders may attempt to manipulate an interface, such as a user interface, an application programming interface (API), a system programming interface (SPI), or a microcode or hardware interface.

2. **Compute Power/Storage Capability**
The capability of the adversary to do large amounts of computation and/or store large amounts of data is usually translated into monetary terms, with a trend of decreasing cost of computation and storage. For example, total key exhaustion of a 56-bit DES key requires 2 or 3 blocks of known plaintext/ciphertext pairs and the ability to test 2**56 keys (over 72 quadrillion trials).

3. **Read/Write/Modify/Delay/Replay/Insert/Delete Data**

   - Must the data remain a secret?
   If yes, the data must be scrambled when potentially exposed. The sender enciphers the data and the receiver deciphers it.

   - Must the information remain authentic, that is, as the sender sent it?
   If yes, data must have an integrity checksum when potentially exposed. The sender calculates the checksum and the receiver verifies it.

   - **Must the receiver be able to detect stale (non-current) data?**
   If yes, a time variant parameter containing one or more of the following must be used:

     1. Timestamp appended by sender and verified by receiver. This implies synchronised clocks exist on the sender and receiver.

     2. Unpredictable nonce generated by receiver and sent to sender, then appended to message by sender and verified by receiver.

   - **Must the receiver be able to detect when data has be duplicated, inserted, or deleted?**
   If yes, a time variant parameter containing one or more of the following must be used:

     1. Monotonically increasing sequence number appended by sender and verified by receiver.

     2. 2) Unpredictable nonce generated by receiver and sent to sender, then appended to message by sender and verified by receiver.

   - **Recover key from Ciphertext Only/Known Plaintext/Chosen Plaintext**
   The assumptions regarding what an adversary knows regarding enciphered messages. If the adversary only knows the ciphertext on the link, this is called a ciphertext only attack and is the hardest to perform. If the adversary knows some plaintext and matching ciphertext, this is called a known plaintext attack and this knowledge can often be used to develop an improved attack.

   Historical experience shows that systems should be designed to resist known plaintext attacks to recover the key. If the adversary can determine the ciphertext for arbitrary plaintext, this is known as a chosen plaintext attack. This is one of the most powerful assumptions to make regarding the capabilities of an adversary and is often not a realistic threat. However, it is very desirable design a system to resist a chosen plaintext attack to recover the key, if possible.

- **Recover plaintext using a dictionary**
  If known plaintext exists, a dictionary matching the plaintext to the ciphertext may be built, which may allow recovery of all or part of an enciphered message, without necessarily recovering the key. This includes the possibility of an outsider building a dictionary for a personal key or an insider for a system key.

- **Requirements on cryptographic keys**
  It is important to remember that the requirements for cryptographic keys are varied. A secret symmetric key or a private asymmetric key must have its secret values remain secret, its key values maintained with integrity, and the system must allow usage of the key only to an authorised entity. To support the non-repudiation of digitally signed messages, it must be possible for an authorised caller to use a private asymmetric key but it must not be possible for the caller to determine the value of the key, otherwise the caller could disclose the value to another party and then claim that the other party digitally signed the message.

  A public asymmetric key has no values that must remain secret but it key values must be maintained with integrity and information regarding the owner of the associated private asymmetric key must be coupled to the key, for example, by a certificate.

- **Random number generator/key generator**
  A pseudorandom number generator (PRNG) is often used to generate symmetric keys and used for input to the generation of asymmetric keys. The quality of the PRNG must be such that an adversary cannot succeed in breaking the PRNG with less cost than to break a key.

- **Physical security**
  One goal is to deny access to an adversary any area where data is in its "raw" unprocessed form. This can be as simple as locking a door or as extreme as rendering a device unusable on detection of tampering, as is specified in FIPS 140-1.

  One way to measure appropriate physical security is to consider the value of what is being protected, as measures appropriate for small value data will likely be inappropriate for large value data.

- **Standard security methods** Access controls on cryptographic services and keys and well as audit of usage of same with alerts for suspicious activity are appropriate.

*Appendix B*

# Example Template CCs

This appendix presents a set of example Template CCs for a number of common algorithms and uses and example sets of Template CCs that may be assembled as a means of packaging cryptographic algorithms.

## B.1 Example Sets of Template CCs

### B.1.1 FULL RSA

| **Functionality** | **Template CC** |
|---|---|
| Encryption | RSA-RC2-CBC<br>RSA-RC4 |
| Signature | RSA-SIGN-SHA-1<br>RSA-VERIFY-SHA-1<br>RSA-SIGN-MD5<br>RSA-VERIFY-MD5 |
| Hash | MD5-HASH<br>SHA-1-HASH |
| Key Exchange | RSA-EXPORT<br>RSA-IMPORT |

### B.1.2 SIGNATURE RSA

| **Functionality** | **Template CC** |
|---|---|
| Signature | RSA-SIGN-SHA-1<br>RSA-VERIFY-SHA-1<br>RSA-SIGN-MD5<br>RSA-VERIFY-MD5 |
| Hash | MD5-HASH<br>SHA-1-HASH |
| Key Exchange | RSA-EXPORT<br>RSA-IMPORT |

### B.1.3 FORTEZZA

| **Functionality** | **Template CC** |
|---|---|
| Encryption | SKIPJACK |
| Signature | DSS-SIGN<br>DSS-VERIFY |
| Hash | SHA-1-HASH |
| Key Exchange | KEA-EXPORT<br>KSA-IMPORT |

### B.1.4 DSS

| **Functionality** | **Template CC** |
|---|---|
| Signature | DSS-SIGN<br>DSD-VERIFY |
| Hash | SHA-1-HASH |

### B.1.5 MS-MAIL

| **Functionality** | **Template CC** |
|---|---|
| Encryption | CAST |
| Signature | RSA-SIGN-MD5<br>RSA-VERIFY-MD5 |
| Hash | MD5-HASH |
| Key Exchange | RSA-EXPORT<br>RSA-IMPORT |

### B.1.6 Default SSL

| **Functionality** | **Template CC** |
|---|---|
| Encryption | DES-CBC |
| Signature | RSA-SIGN-SHA-1<br>RSA-VERIFY-SHA-1 |
| Hash | SHA-1-HASH |
| Key Exchange | RSA-EXPORT<br>RSA-IMPORT |

## B.2 Example Template CCs

### B.2.1 DES-CBC

| **Field** | : | **Value** |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | Keyed |
| Context_Confidentiality_Flag | : | Yes |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | GCS_A_DES |
| Mode_of_operation | : | GCS_M_CBC |
| Short_Block_Policy | : | X9.23 |
| Algorithm_Specific_Parameters | : | Key length |
| | : | Feedback Length |
| | : | IV Parameter Length |
| **Key_Context** | | |
| Key_Usage | : | GCS_C_ENCIPHER_DATA |
| | : | GCS_C_DECIPHER_DATA |
| Permitted_Export_Mechanisms | : | Site and purpose specific values |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | - |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | - |
| Key_Value | : | - |

## B.2.2 RSA-RC2-CBC

| **Field** | : | **Value** |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | KEYED |
| Context_Confidentiality_Flag | : | YES |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | RSA-RC2 |
| Mode_of_operation | : | |
| Short_Block_Policy | : | GCS_SBP_PKCS#1 |
| Algorithm_Specific_Parameters | : | Feedback length |
| | : | Block length |
| **Key_Context** | | |
| Key_Usage | : | GCS_C_ENCIPHER_DATA |
| | : | GCS_C_DECIPHER_DATA |
| Permitted_Export_Mechanisms | : | Site and purpose specific values |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | - |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | - |
| Key_Value | : | - |

### B.2.3 RSA-RC4

| **Field** | : | **Value** |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | KEYED |
| Context_Confidentiality_Flag | : | YES |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | RSA-RC4 |
| Mode_of_operation | : | GCS_M_NONE |
| Short_Block_Policy | : | PKCS#1 |
| Algorithm_Specific_Parameters | : | Feedback length |
| | : | Block length |
| **Key_Context** | | |
| Key_Usage | : | GCS_C_ENCIPHER_DATA |
| | : | GCS_C_DECIPHER_DATA |
| Permitted_Export_Mechanisms | : | Site and purpose specific values |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | - |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | - |
| Key_Value | : | - |

### B.2.4 SKIPJACK

| **Field** | : | **Value** |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | KEYED |
| Context_Confidentiality_Flag | : | YES |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | SKIPJACK |
| Mode_of_operation | : | GCS_M_CBC |
| Short_Block_Policy | : | SKIPJACK |
| Algorithm_Specific_Parameters | : | Key Length |
| | : | Feedback Length |
| | : | IV Length |
| **Key_Context** | | |
| Key_Usage | : | GCS_C_ENCIPHER_DATA |
| | : | GCS_C_DECIPHER_DATA |
| Permitted_Export_Mechanisms | : | GCS_FORTEZZA_KEA |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | - |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | - |
| Key_Value | : | - |

### B.2.5 CAST

| **Field** | : | **Value** |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | KEYED |
| Context_Confidentiality_Flag | : | YES |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | CAST |
| Mode_of_operation | : | GCS_M_CBC |
| Short_Block_Policy | : | |
| Algorithm_Specific_Parameters | : | Key Length |
| | : | Feedback Length |
| | : | IV Length |
| **Key_Context** | | |
| Key_Usage | : | GCS_C_ENCIPHER_DATA |
| | : | GCS_C_DECIPHER_DATA |
| Permitted_Export_Mechanisms | : | Site and purpose specific values |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | - |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | - |
| Key_Value | : | - |

### B.2.6 RSA-SIGN-SHA-1

| Field | : | Value |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | BOTH |
| Context_Confidentiality_Flag | : | YES |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | RSA |
| Mode_of_operation | : | - |
| Short_Block_Policy | : | PKCS#1-Format1 |
| Algorithm_Specific_Parameters | : | Modulus Size |
| | : | Exponent Value |
| **Non_Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | SHA-1 |
| Mode_of_Operation | : | - |
| Short_Block_Policy | : | - |
| Algorithm_Specific_Parameters | : | - |
| **Key_Context** | | |
| Key_Usage | : | GCS_C_GENERATE_CV |
| Permitted_Export_Mechanisms | : | GCS_NO_EXPORT |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | - |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | - |
| Key_Value | : | - |

### B.2.7 RSA-VERIFY-SHA-1

| **Field** | : | **Value** |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | BOTH |
| Context_Confidentiality_Flag | : | YES |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | RSA |
| Mode_of_operation | : | - |
| Short_Block_Policy | : | PKCS#1-Format1 |
| Algorithm_Specific_Parameters | : | Modulus size |
| | : | Exponent value |
| **Non_Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | SHA-1 |
| Mode_of_Operation | : | - |
| Short_Block_Policy | : | - |
| Algorithm_Specific_Parameters | : | - |
| **Key_Context** | | |
| Key_Usage | : | GCS_C_VERIFY_CV |
| Permitted_Export_Mechanisms | : | GCS_NO_EXPORT |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | - |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | - |
| Key_Value | : | - |

### B.2.8 RSA-SIGN-MD5

| **Field** | : | **Value** |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | BOTH |
| Context_Confidentiality_Flag | : | YES |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | RSA |
| Mode_of_operation | : | - |
| Short_Block_Policy | : | PKCS#1-Format1 |
| Algorithm_Specific_Parameters | : | Modulus size |
| | : | Exponent value |
| **Non_Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | MD5 |
| Mode_of_Operation | : | - |
| Short_Block_Policy | : | - |
| Algorithm_Specific_Parameters | : | - |
| **Key_Context** | | |
| Key_Usage | : | GCS_C_GENERATE_CV |
| Permitted_Export_Mechanisms | : | GCS_NO_EXPORT |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | - |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | - |
| Key_Value | : | - |

### B.2.9 RSA-VERIFY-MD5

| **Field** | : | **Value** |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | BOTH |
| Context_Confidentiality_Flag | : | YES |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | RSA |
| Mode_of_operation | : | - |
| Short_Block_Policy | : | PKCS#1-Format1 |
| Algorithm_Specific_Parameters | : | Modulus size |
| | : | Exponent value |
| **Non_Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | MD5 |
| Mode_of_Operation | : | - |
| Short_Block_Policy | : | - |
| Algorithm_Specific_Parameters | : | - |
| **Key_Context** | | |
| Key_Usage | : | GCS_C_VERIFY_CV |
| Permitted_Export_Mechanisms | : | GCS_NO_EXPORT |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | - |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | - |
| Key_Value | : | - |

### B.2.10 RSA-EXPORT

| **Field** | : | **Value** |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | KEYED |
| Context_Confidentiality_Flag | : | YES |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | RSA |
| Mode_of_operation | : | - |
| Short_Block_Policy | : | PKCS#1-Format1 |
| Algorithm_Specific_Parameters | : | Modulus size |
| | : | Exponent value |
| **Key_Context** | | |
| Key_Usage | : | GCS_C_EXPORT_KEY |
| Permitted_Export_Mechanisms | : | GCS_NO_EXPORT |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | - |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | - |
| Key_Value | : | - |

### B.2.11 RSA-IMPORT

| **Field** | : | **Value** |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | KEYED |
| Context_Confidentiality_Flag | : | YES |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | RSA |
| Mode_of_operation | : | - |
| Short_Block_Policy | : | PKCS#1-Format1 |
| Algorithm_Specific_Parameters | : | Modulus size |
| | : | Exponent length |
| **Key_Context** | | |
| Key_Usage | : | GCS_C_IMPORT_KEY |
| Permitted_Export_Mechanisms | : | GCS_NO_EXPORT |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | - |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | - |
| Key_Value | : | - |

### B.2.12 DSS-SIGN

| Field | : | Value |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | Both |
| Context_Confidentiality_Flag | : | YES |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | DSS |
| Mode_of_operation | : | - |
| Short_Block_Policy | : | - |
| Algorithm_Specific_Parameters | : | Size of p |
| | : | Values of p, q and g |
| **Non_Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | SHA-1 |
| Mode_of_Operation | : | - |
| Short_Block_Policy | : | - |
| Algorithm_Specific_Parameters | : | - |
| **Key_Context** | | |
| Key_Usage | : | GCS_C_GENERATE_CV |
| Permitted_Export_Mechanisms | : | GCS_NO_EXPORT |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | NULL |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | - |
| Key_Value | : | - |

### B.2.13 DSS-VERIFY

| **Field** | : | **Value** |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | Both |
| Context_Confidentiality_Flag | : | Yes |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | DSS |
| Mode_of_operation | : | - |
| Short_Block_Policy | : | - |
| Algorithm_Specific_Parameters | : | Size of p |
| | : | Values of p, q and g |
| **Non_Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | SHA-1 |
| Mode_of_Operation | : | - |
| Short_Block_Policy | : | - |
| Algorithm_Specific_Parameters | : | - |
| **Key_Context** | | |
| Key_Usage | : | GCS_GC_VERIFY_CV |
| Permitted_Export_Mechanisms | : | GCS_NO_EXPORT |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | - |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | - |
| Key_Value | : | - |

### B.2.14 KEA-EXPORT

| **Field** | : | **Value** |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | KEYED |
| Context_Confidentiality_Flag | : | YES |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | KEA |
| Mode_of_operation | : | - |
| Short_Block_Policy | : | - |
| Algorithm_Specific_Parameters | : | Size of p |
| | : | Values of p. q and g |
| **Key_Context** | | |
| Key_Usage | : | GCS_C_EXPORT |
| Permitted_Export_Mechanisms | : | GCS_NO_EXPORT |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | - |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | - |
| Key_Value | : | - |

### B.2.15 KEA-IMPORT

| **Field** | : | **Value** |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | KEYED |
| Context_Confidentiality_Flag | : | YES |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | KEA |
| Mode_of_operation | : | - |
| Short_Block_Policy | : | - |
| Algorithm_Specific_Parameters | : | Size of p |
| | : | Values of p, q and g |
| **Key_Context** | | |
| Key_Usage | : | GCS_C_IMPORT |
| Permitted_Export_Mechanisms | : | GCS_NO_EXPORT |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | - |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | - |
| Key_Value | : | - |

### B.2.16 DES-X9.17

| **Field** | : | **Value** |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | KEYED |
| Context_Confidentiality_Flag | : | YES |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | DES |
| Mode_of_operation | : | GCS_M_ECB |
| Short_Block_Policy | : | GCS_SBP_NONE |
| Algorithm_Specific_Parameters | : | - |
| **Key_Context** | | |
| Key_Usage | : | GCS_C_EXPORT_KEY |
| | : | GCS_C_IMPORT_KEY |
| | : | GCS_C_COMBINE_KEY |
| Permitted_Export_Mechanisms | : | Site and purpose specific values |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | GCS_SPLIT |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | Receive_Count =1 |
| | : | Send_count = 1 |
| | : | My_Node = site specific |
| | : | Your_node = site specific |
| Key_Value | : | - |

### B.2.17 DES-MAC

| **Field** | : | **Value** |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | KEYED |
| Context_Confidentiality_Flag | : | YES |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | DES |
| Mode_of_operation | : | GCS_M_DES_MAC |
| Short_Block_Policy | : | GCS_SBP_DES_MAC |
| Algorithm_Specific_Parameters | : | - |
| **Key_Context** | | |
| Key_Usage | : | GCS_C_GENERATE_CV |
| | : | GCS_C_VERIFY_CV |
| | : | GCS_C_COMBINE_CC |
| Permitted_Export_Mechanisms | : | Site and purpose specific values |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | - |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | - |
| Key_Value | : | - |

### B.2.18 DIFFIE-HELLMAN-EXPORT

| **Field** | : | **Value** |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | KEYED |
| Context_Confidentiality_Flag | : | YES |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | DH |
| Mode_of_operation | : | - |
| Short_Block_Policy | : | - |
| Algorithm_Specific_Parameters | : | Size of p |
| | : | Values of p, q and g |
| **Key_Context** | | |
| Key_Usage | : | GCS_C_EXPORT_KEY |
| Permitted_Export_Mechanisms | : | GCS_NO_EXPORT |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | - |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_of_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | - |
| Key_Value | : | - |

### B.2.19 DIFFIE-HELLMAN-IMPORT

| **Field** | : | **Value** |
|---|---|---|
| **CC_Header** | | |
| Context_Type | : | KEYED |
| Context_Confidentiality_Flag | : | YES |
| **Keyed_Algorithm_Context** | | |
| Algorithm_ID | : | DH |
| Mode_of_operation | : | - |
| Short_Block_Policy | : | - |
| Algorithm_Specific_Parameters | : | Size of p |
| | : | Values of p, q and g |
| **Key_Context** | | |
| Key_Usage | : | GCS_C_IMPORT |
| Permitted_Export_Mechanisms | : | GCS_NO_EXPORT |
| Key_State | : | - |
| Time_of_Revocation | : | - |
| Reason_for_Revocation | : | - |
| Key_Flag | : | - |
| Split_Protocol_Type | : | - |
| Key_Part_Number | : | - |
| Number_of_Key_Parts | : | - |
| Key_Validity | : | Site Specific Values |
| Initialisation_Vector (IV) | : | - |
| Key_Specific_Parameters | : | - |
| Key_Value | : | - |

# Example Walkthroughs

## C.1 ANSI X9.17 Key Distribution Protocol

```
 Scenario:  Party A wishes to send a MAC KD and an
        encryption key KD to Party B.  The keys will be notarized.
        The minimum conformant implementation for a point
        to point environment requires the exchange of two
        Cryptographic Service Messages (CSMs): a Key Service
        Message (KSM) to transfer the encrypted KDs, and a
        Response Service Message (RSM) to acknowledge receipt of
        the KD.   Message authentication is provided by the
        MAC field of each CSM.

Notes:
 A. Error handling is not addressed.
 B. The applications construct the formatted CSMs.
 C. Notation:
  1. KDm denotes a data MAC, KDe denotes a data
     encrypting key, KK denotes a key-encrypting-key.
     KK1 and KK2 represent the first and second parts of a split KK.

  2. Assuming a and b are pointer arguments, the
     notation a => b indicates that b is set to the
     same address as a.  a <= b indicates the reverse operation.

Process Summary

  A. Each party initializes their system by loading a manually
     installed KK that is shared with the other party.

     1. Each party creates two CCs to contain the split components
        of the manually installed KK.
     2. Each party populates the two CCs with the respective
        split components of KK, and combines them to form a
        third CC containing the final KK value.
     3. X9.17 requires each party to maintain two counters
        associated with the KK: an origination count and a
        reception count.  These counters are set to 1 when
        the manually distributed KK is loaded.  There is also
        the (optional) setting of the size of the reception
        window, as some distribution methods do not guarantee the
        order of delivery is the same as the order of initiation
        of transmission.  For simplicity, the window is set to 1.

  B. Party A generates a data MAC key KDm and a data encrypting key KDe
     and sends them to Party B in an X9.17 Key Service Message (KSM):
```

```
      1. Create/Retrieve a CC and populate it with the KDm.
      2. Create/Retrieve a CC and populate it with the KDe.
      3. Create/Retrieve a CC and combine KDm and KDe to form
         the CSM MAC key.
      4. Export KDm, sealed by the shared KK.
      5. Export KDe, sealed by the shared KK.
      6. Increment the origination count associated with KK.
      7. Construct a partial KSM ASCII string containing KDn and KDe.
         E.g.: (MCL/KSM RCV/PartyB ORG/PartyA KD/[Key Value]
                EDK/[Key Activation Date] CTP/1
      8. Generate a MAC on the partial KSM using the CSM MAC key.
      9. Complete the KSM ASCII string by appending the MAC field.
     10. Transmit the completed KSM to Party B.

   C. Party B receives and verifies the KSM:

      1. Verify the counter field in the KSM with the reception count
         in the KK.  As the window size is 1, they must be equal.
      2. Create/Retrieve a CC for the KDm.
      3. Create/Retrieve a CC for the KDe.
      4. Create/Retrieve a CC for the CSM MAC key.
      5. Import KDm using the shared KK.
      6. Import KDe using the shared KK.
      7. Combine KDm and KDe to form the CSM MAC key.
      8. Verify the MAC on the KSM using the CSM MAC key.
      9. Increment the reception counter associated with KK.

   D. Party B generates an X9.17 Response Service Message (RSM) and
      sends it to Party A:

      1. Construct a partial RSM. E.g., (MCL/RSM RCV/PartyA ORG/PartyB...
      2. Generate a MAC on the partial RSM using the CSM MAC key.
      3. Delete the CC for the CSM MAC key.
      4. Form the complete RSM by appending the MAC to the partial RSM.
      6. Send the complete RSM to Party A.

   E. Party A receives and verifies the RSM:

      1. Extract the MAC field from the received RSM.
      2. Verify the MAC on the partial RSM using the CSM MAC key.
      3. Delete the CC for the CSM MAC key.

   F. Party A and Party B now share KDm and KDe.

========================================================================

 Pseudocode example:
   Note that not all parameters are listed.

   A. Each party initializes their system by loading a manually
      installed KK that is shared with the other party.
```

```
     1. Each party creates two CCs to contain the split components
        of the manually installed KK.

gcs_create_ac ( ac => AC_KK1 );

gcs_set_ac ( ac <= AC_KK1 );
/* One call is made to gcs_set_ac for each of the parameter
name/value pairs in the following table:  */
parameter_name       parameter_value
--------------       ---------------
ALGORITHM_ID         DES
ALGORITHM_CLASS_ID   SYMMETRIC
MODE_OF_OPERATION    DES_CBC
SHORT_BLOCK_POLICY   X9.23

gcs_create_kc ( kc => KC_KK1 );
gcs_set_kc ( kc <= KC_KK1 );
/* One call is made to this function for each of the parameter
name/value pairs in the following table: */
parameter_name       parameter_value
--------------       ---------------
KEY_USAGE            GCS_C_EXPORT_KEY
KEY_USAGE            GCS_C_IMPORT_KEY
KEY_USAGE            GCS_C_COMBINE_CC
KEY_FLAG             GCS_SPLIT
KEY_LIFETIME         GCS_INFINITE
The BER encoding value of RECEIVE_COUNT, SEND_COUNT, MY_NODE and
YOUR_NODE.

gcs_create_cc ( keyed_ac <= AC_KK1,
                kc => KC_KK1,
                output_CC => CC_KK1 );

The same process is repeated substituting KK2 for KK1.

/*****************************************************************
     2. Each party populates the two CCs with the respective
        split components of KK, and combines them to form a
        third CC containing the final KK value.
     3. X9.17 requires each party to maintain two counters
        associated with the KK: an origination count and a
        reception count.  These counters are set to 1 when
        the manually distributed KK is loaded.  There is also
        the (optional) setting of the size of the reception
        window, but this is set to 1.
*****************************************************************/

gcs_load_key ( subject_cc <= CC_KK1,
               input_key_part = KK1,
               key_part_type = GCS_FIRST );

gcs_load_key ( subject_cc <= CC_KK2,
```

```
               input_key_part = KK2,
               key_part_type = GCS_LAST );

gcs_combine_cc ( cc_list <= CC_KK1 CC_KK2,
                 skeleton_cc => CC_KK );

gcs_delete_cc ( subject_cc <= CC_KK1 );
gcs_delete_cc ( subject_cc <= CC_KK2 );

Party B also does the same processes for KK1 and KK2, except that
MY_NODE = PartyB and YOUR_NODE = PartyA.

/*****************************************************************
  B. Party A generates a data MAC key KDm and a data encrypting key KDe
     and sends them to Party B in an X9.17 Key Service Message (KSM):

     1. Create/Retrieve a CC and populate it with the KDm.
*****************************************************************/

gcs_create_ac ( ac => AC_KDm );

gcs_set_ac ( ac <= AC_KDm );
/* One call is made to this function for each of the parameter
name/value pairs in the following table: */
parameter_name      parameter_value
--------------      ---------------
ALGORITHM_ID        DES
ALGORITHM_CLASS_ID  SYMMETRIC
MODE_OF_OPERATION   DES-MAC
SHORT_BLOCK_POLICY  DES-MAC

gcs_create_kc ( kc => KC_KDm );

gcs_set_kc ( kc <= KC_KDm );
/* One call is made to this function for each of the parameter
name/value pairs in the following table: */
parameter_name      parameter_value
--------------      ---------------
KEY_USAGE           GCS_C_GENERATE_CV
KEY_USAGE           GCS_C_VERIFY_CV
KEY_USAGE           GCS_C_COMBINE_CC
KEY_FLAG            GCS_EXPORTABLE
KEY_LIFETIME        GCS_INFINITE

gcs_create_cc ( keyed_ac <= AC_KDm,
                kc <= KC_KDm,
                output_cc => CC_KDm );

gcs_generate_key ( subject_cc <= CC_KDm );

/*   2. Create/Retrieve a CC and populate it with the KDe. */
```

```
gcs_create_ac ( ac => AC_KDe );

gcs_set_ac ( ac <= AC_KDe );
/* One call is made to this function for each of the parameter
name/value pairs in the following table: */
parameter_name       parameter_value
--------------       ---------------
ALGORITHM_ID          DES
ALGORITHM_CLASS_ID    SYMMETRIC
MODE_OF_OPERATION     DES_CBC
SHORT_BLOCK_POLICY    X9.23
ALG_SPEC_PARMS        IV_REQUIRED

gcs_create_kc ( kc => KC_KDe );

gcs_set_kc ( kc <= KC_KDe );
/* One call is made to this function for each of the parameter
name/value pairs in the following table: */
parameter_name       parameter_value
--------------       ---------------
KEY_USAGE            GCS_C_ENCIPHER
KEY_USAGE            GCS_C_DECIPHER
KEY_USAGE            GCS_C_COMBINE_CC
KEY_FLAG             GCS_EXPORTABLE
KEY_LIFETIME         GCS_INFINITE

gcs_create_cc ( keyed_ac <= AC_KDe,
                kc <= KC_KDe,
                output_cc => CC_KDe );

gcs_generate_key ( subject_cc <= CC_KDe );

/* Create a CC and combine KDm and KDe to form the CSM MAC key. */
gcs_create_ac ( ac => AC_KDcm );

gcs_set_ac ( ac <= AC_KDcm );
/* One call is made to this function for each of the parameter
name/value pairs in the following table: */
parameter_name       parameter_value
--------------       ---------------
ALGORITHM_ID         DES
ALGORITHM_CLASS_ID   SYMMETRIC
MODE_OF_OPERATION    DES-MAC
SHORT_BLOCK_POLICY   DES-MAC

gcs_create_kc ( kc => KC_KDcm );

gcs_set_kc ( kc <= KC_KDcm );
/* One call is made to this function for each of the parameter
name/value pairs in the following table: */
parameter_name       parameter_value
--------------       ---------------
```

```
KEY_USAGE            GCS_C_GENERATE_CV
KEY_USAGE            GCS_C_VERIFY_CV
KEY_LIFETIME         GCS_INFINITE

gcs_create_cc ( keyed_ac <= AC_KDcm,
                kc <= KC_KDcm,
                output_cc => CC_KDcm );

gcs_combine_cc ( subject_cc <= CC_KDm,
                 subject_cc <= CC_KDe,
                 target_cc => CC_KDcm );

 /* 4. Export KDm, sealed by the shared KK. */

gcs_export_key ( subject_cc <= CC_KDm,
                 kek_cc <= CC_KK,
                 export_mech = KDm components of KSM );

 /* 5. Export KDe, sealed by the shared KK. */

gcs_export_key ( subject_cc <= CC_KDe,
                 kek_cc <= CC_KK,
                 export_mech = GCS_X9.17_NOTARIZE,
                 partial_PDU = KDe components of KSM );

 /*  6. Increment the origination count associated with KK. */

gcs_get_cc ( subject_cc <= CC_KK,
             subject_container <= kc,
             parameter_name = ORG_COUNT,
             parameter_integer_value = ocount );

gcs_set_kc ( kc <= KC_KK,
              parameter_name = ORG_COUNT,
              parameter_integer_value = [ocount + 1] );

/*   7. Construct a partial KSM ASCII string containing KDn and KDe. */
/*      E.g.: (MCL/KSM RCV/PartyB ORG/PartyA KD/[Key Value]          */
/*             EDK/[Key Activation Date] CTP/1                       */
/*   8. Generate a MAC on the partial KSM using the CSM MAC key.     */

gcs_generate_check_value ( cc <= CC_KD,
                           input_buffer = partial KSM,
                           chaining_flag = GCS_ONLY,
                           check_value => MAC );
/*   9. Complete the KSM ASCII string by appending the MAC field.  */
/*  10. Transmit the completed KSM to Party B.                     */

/* C. Party B receives and verifies the KSM:                         */
/*   1. Verify the counter field in the KSM with the reception count */
/*      in the KK.  As the window size is 1, they must be equal.     */
```

```
gcs_get_cc ( subject_cc <= CC_KK,
             subject_container <= kc,
             parameter_name = RCV_COUNT,
             parameter_integer_value = ocount );

/*   2. Create a CC for the KDm.                                  */
/*   3. Create a CC for the KDe.                                  */
/*   4. Create a CC for the KDcm.                                 */
 See above.

/*   5. Import KDm using the shared KK.                        */

gcs_import_key ( subject_cc <= CC_KDm,
                 kek_cc <= CC_KK,
                 export_mech = GCS_X9.17_NOTARIZE,
                 partial_PDU = KDm component from KSM );

/*   6. Import KDe using the shared KK.                         */

gcs_import_key ( subject_cc <= CC_KDe,
                 kek_cc <= CC_KK,
                 export_mech = GCS_X9.17_NOTARIZE,
                 partial_PDU = KDe component from KSM );

/*   7. Combine KDm and KDe to form the CSM MAC key KDcm.       */

gcs_combine_cc ( subject_cc <= CC_KDm,
                 subject_cc <= CC_KDe,
                 target_cc => CC_KDcm );

/*    8. Verify the MAC on the KSM using the CSM MAC key.       */

gcs_verify_check_value ( cc <= CC_KDcm,
                         input_buffer = partial KSM,
                         check_value = MAC from KSM,
                         chaining_flag = GCS_ONLY );

 /* 9. Increment the reception counter associated with KK.     */

gcs_get_cc ( subject_cc <= CC_KK,
             subject_container = kc,
             parameter_name = RCV_COUNT,
             parameter_integer_value = rcount );

gcs_set_key ( kc <= KC_KD,
              parameter_name = REC_COUNT,
              parameter_integer_value = [rcount + 1] );

/* D. Party B generates an X9.17 Response Service Message (RSM) and */
/*    sends it to Party A:                                          */
/*                                                                  */
/*    1. Construct a partial RSM: (MCL/RSM RCV/PartyA ORG/PartyB    */
```

```
/*     2. Generate a MAC on the partial RSM using KDcm.              */

gcs_generate_check_value ( cc <= CC_KDcm,
                           input_buffer = PARTIAL_RSM,
                           chaining_flag = GCS_ONLY,
                           check_value => RSM_MAC );

/*     3. Delete the CC for the CSM MAC key.                         */

gcs_delete_cc ( subject_cc <= CC_KDcm );

/*     4. Form the complete RSM by appending the MAC to partial RSM. */
/*     5. Send the complete RSM to Party A.                          */

/* E. Party A receives and verifies the RSM:                         */
/*    1. Extract the MAC field from the received RSM.                */
/*    2. Verify the MAC on the partial RSM using the CSM MAC key.    */

gcs_verify_check_value ( cc <= CC_KD,
                         input_buffer = PARTIAL_RSM,
                         check_value = RSM_MAC,
                         chaining_flag = GCS_ONLY );

/*    3. Delete the CC for the CSM MAC key.                          */

gcs_delete_cc ( subject_cc <= CC_KDcm );

/*   F. Party A and Party B now share KDm and KDe.                   */
```

## C.2 Fortezza Public Key Exchange

This section provides a mapping of of the GCS-API public key exchange calls onto their Fortezza counterparts

```
************************************************************************
P.S. function prototypes
************************************************************************

gcs_key_agreement(
      OM_uint32                  *minor_status,
      gcs_session_context_t      *session_context,
      gcs_cc_t                   *caller_cc,
      gcs_cc_t                   *other_cc,
      gcs_bit_string_t           *pdu_in,
      gcs_bit_string_t           *pdu_out,
      gcs_cc_t                   *kak_cc);

************************************************************************
                    DESCRIPTION
************************************************************************

The following shows the GCS-API calls necessary to implement a key
exchange between an initiator and a recipient, using the Key Exchange
Algorithm as exists on the Fortezza card.  Therefore, Fortezza is the
hardware cryptomodule that underlies this GCS example.  Initiator
pseudocode is shown first, followed by recipient pseudocode.  In each
case, the listing of GCS and Fortezza calls is shown before the
pseudocode, so one can eyeball the whole process in about a paragraph,
before expanding these calls with all their parameters.

The goal is to have initiator and recipient establish a session key,
(or MEK, a Message Encryption Key), to protect a direct-connected
session.  This is done by each agreeing on a Token Encryption Key (TEK)
via the key exchange.  Then the initiator encrypts the MEK with the TEK
and sends it to the recipient.  The TEK is used only to protect the MEK
enroute to the recipient.  The MEK is used to protect the rest of the
session.  In KEA, random quantities must be exchanged between initiator
and recipient.  Since this is a session application, it is assumed that
this exchange takes place as a first step in processing by the key
exchange function calls (gcs_export_key_agreement and
gcs_import_key_agreement).

Generally, the initiator is the "exporter" of the TEK, and the
recipient is the "importer".   So the initiator performs the key
exchange using gcs_key_agreement, and prepares the MEK
for transmission to the recipient using gcs_export_key.  The
recipient uses gcs_key_agreement and gcs_import_key in
doing the corresponding actions.  The main difference between initiator
side processing and recipient side processing is that the recipient
does not need to compute the random session key.
```

For purposes of this example, it is assumed that templates of all necessary cryptographic context (CC) data structures reside in the CC library. Therefore, necessary CCs are "checked out" of the CC library, and when appropriate, copies are made and populated for the application's use.

This pseudocode is meant to be an excerpt of GCS code. Therefore, only calls directly affecting the key exchange are shown. This is true for both the GCS calls and the Fortezza calls. The Fortezza calls that the GCS-API calls use for implementation are shown after the GCS-API calls, and are indented. To help keep straight whether parameters are inputs, outputs, or both, the following key is used: "->" means input, "<-" means output, and "<->" means input and output.

```
************************************************************************
                    INITIATOR SIDE
************************************************************************

************************************************************************
Initiator Side Call Mapping and Overview
************************************************************************

------------------------------------------------------------------------
Fortezza calls                   GCS-API calls
------------------------------------------------------------------------

CI_CheckPIN                gcs_initialise_session
CI_SetPersonality
                           gcs_retrieve_cc     (initiator's private key CC)
                           gcs_retrieve_cc (recipient's public key CC)
                           gcs_load_public_key (load recipient's public key)
                           gcs_retrieve_cc     (TEK CC)

CI_GenerateRa              gcs_generate_random_number (initiator's
                                                 random quantity)
                           <send initiator's random to recipient>
                           <receive recipient's random number>
CI_GenerateTEK                   gcs_export_key_agreement (form TEK)
                           gcs_retrieve_cc (MEK CC)

CI_GenerateMEK                   gcs_generate_key (generate random MEK)
CI_WrapKey                 gcs_export_key (wrap MEK by TEK)
                           <transmit TEK-wrapped MEK to recipient>
                           <use MEK to protect session>

************************************************************************
Initiator-side pseudocode
(Underlying Fortezza calls are interleaved, and are indented)
************************************************************************

/* Establish a session context */
```

```
gcs_initialise_session
       <-  (minor_status,
       <-> session_context,
        -> initialise_parameters)

       /* To log onto Fortezza card, must pass a PIN check. */
       CI_CheckPIN
               -> (PINType <- CI_USER_PIN,      /* subject is USER, not SSO */
               -> pPIN,                  /* pointer to PIN */
              <-  return value)

       /* Do other Fortezza processing to determine the card slot that */
       /* has the correct set of keys for this application.  Put that */
       /* value in variable "PersonalityIndex". This call affects which */
       /* cryptographic contexts will be retrieved below for each side */
       /* in the key agreement call. */
       CI_SetPersonality
               -> (CertificateIndex <- PersonalityIndex,
              <-  return value)

/* Retrieve the initiator’s private key CC.  This is a case where the*/
/* CC is retrieved and directly used, rather than being copied and */
/* populated. This is because the initiator’s private key is long-term, */
/* and not used by anyone else.                              */
gcs_retrieve_cc
       <-  (minor_status,
        -> session_context,
        -> domain_id,
        -> cc_name <- GCS_CC_NAME_FORTEZZA_KEA_PRIVATE,
        -> cc_reference <- GCS_NULL,
       <-  retrieved_cc <- initiator_cc);

/* Retrieve a copy of a recipient’s public key CC, and copy it. */
gcs_retrieve_cc
       <-  (minor_status,
        -> session_context,
        -> domain_id,
        -> cc_name <- GCS_CC_NAME_FORTEZZA_KEA_RECIP_PUBLIC,
        -> cc_reference <- GCS_NULL,
       <-  retrieved_cc <- recipient_cc);

/* Assuming a bit string holding the recipient’s public key has been*/
/* obtained and placed into variable "recipient_public_key", load that*/
/* key into the recipient’s public key CC.                         */
gcs_load_public_key
       <-  (minor_status,
        -> session_context,
        -> subject_cc <- recipient_cc,
        -> input_key_part <- recipient_public_key,
        -> key_part_type <- GCS_ONLY);

/*
```

```
/* Retrieve a copy of a TEK CC */
gcs_retrieve_cc
       <-  (minor_status,
        -> session_context,
        -> domain_id,
        -> cc_name <- GCS_CC_NAME_FORTEZZA_TEK,
        -> cc_reference <- GCS_NULL,
       <-  retrieved_cc <- temp_cc);
gcs_generate_random_number
       <-  (minor_status,
       ->  session_context,
       ->  GCS_C_FORTEZZA_KEA_RANDOM_SIZE,
       <-  initRand);

       /* Fortezza:  Generate initiator's random quantity. */
       CI_GenerateRa
               -> (none,
              <-  pRa <- initRand, return value)


/* Perform the key exchange.  On this, the initiator's side, the main*/
/* inputs are the initiator's private key, and the recipient's public*/
/* key.  The result is the TEK, held in a CC.                      */
gcs_key_agreement /* form TEK */
       <-  (minor_status,
        -> session_context,
        -> caller_cc <- initiator_cc,
        -> other_cc <- recipient_cc,
        -> pdu_in <- recipRand,
        <-  pdu_out <- initRand,
       <-  kak_cc <- TEK_cc);

       /* Fortezza: Perform initiator-side KEA key agreement algorithm. */
       CI_GenerateTEK
               -> (Flags <- CI_INITIATOR_FLAG,  /* initiator side */
               -> RegisterIndex <- TEKIndex,          /* slot to put TEK */
               -> Ra <- initRand, Rb <- recipRand,    /* random quantities */
               -> Size <- recipPubSize, pY <- recipPub,/* other's public */
              <-  return value);

/* Retrieve a copy of an MEK CC. */
gcs_retrieve_cc
       <-  (minor_status,
        -> session_context,
        -> domain_id,
        -> cc_name <- GCS_CC_NAME_FORTEZZA_MEK, /* assumed name of an MEK CC */
        -> cc_reference <- GCS_NULL,
       <-  retrieved_cc <- MEK_cc);


/* Generate a random MEK and place in MEK_cc */
gcs_generate_key
```

```
        <-  (minor_status,
         -> session_context,
        <-> kak_cc <- MEK_cc);

        /* Fortezza: Generate a random MEK and place in a Fortezza slot */
        CI_GenerateMEK
                -> (RegisterIndex <- MEKIndex, /* assumed this is free slot */
                -> Reserved,
               <-  return value);

/* Wrap (encrypt) MEK by TEK, and place in a CC */
gcs_export_key
        <-  (minor_status,
         -> session_context,
        <-> subject_cc <- MEK_cc,
        <-> kek_cc <- TEK_cc,
        <-> export_data <- exportedMEK);

        /* Fortezza Wrap (encrypt) MEK by TEK */
        CI_WrapKey
                -> (WrapIndex <- TEKIndex,      /* slot where TEK is */
                -> KeyIndex <- MEKIndex,  /* slot where MEK is */
               <-  pKey <- wrappedKey,    /* key ready for export */
               <-  return value);

/* Delete unnecessary CCs */
/* Transmit TEK-wrapped MEK to recipient */
/* Begin protecting session using the MEK */

************************************************************************
                    RECIPIENT SIDE
************************************************************************

************************************************************************
Recipient-side Call Mapping and Overview
************************************************************************

------------------------------------------------------------------------
Fortezza calls                   GCS-API calls
------------------------------------------------------------------------

                           <receive random number from initiator>
CI_CheckPIN                gcs_initialise_session
CI_SetPersonality
                           gcs_retrieve_cc     (recipient’s private key CC)
                           gcs_retrieve_cc (initiator’s public key CC)
                           gcs_load_public_key (load initiator’s public key)
                           gcs_retrieve_cc     (TEK CC)

CI_GenerateRa              gcs_generate_random_number (recipient’s
                                                  random quantity)
                           <send random number to initiator>
```

```
                           <receive TEK-wrapped MEK from initiator>
CI_GenerateTEK                    gcs_key_agreement (form TEK)
CI_UnwrapKey               gcs_import_key (unwrap TEK-wrapped MEK)
                           <use MEK to protect session>

************************************************************************
Recipient-side pseudocode
(Underlying Fortezza calls are interleaved, and are indented)
************************************************************************

gcs_initialise_session
       <-  (minor_status,
       <-> session_context,
        -> initialise_parameters)

       /* To log onto Fortezza card, must pass a PIN check. */
       CI_CheckPIN
               -> (PINType <- CI_USER_PIN,      /* subject is USER, not SSO */
               -> pPIN,                  /* pointer to PIN */
              <-  return value)

       /* Do other Fortezza processing to determine the card slot that */
       /* has the correct set of keys for this application.  Put that */
       /* value in variable "PersonalityIndex". This call affects which */
       /* cryptographic contexts will be retrieved below for each side */
       /* in the key agreement call. */
       CI_SetPersonality
               -> (CertificateIndex <- PersonalityIndex,
              <-  return value)

gcs_retrieve_cc            /* recipient's private key CC */
       <-  (minor_status,
        -> session_context,
        -> domain_id,
        -> cc_name <- GCS_CC_NAME_FORTEZZA_KEA_PRIVATE,
        -> cc_reference <- GCS_NULL,
       <-  retrieved_cc <- recipient_cc);
gcs_retrieve_cc            /* initiator's public key CC */
       <-  (minor_status,
        -> session_context,
        -> domain_id,
        -> cc_name <- GCS_CC_NAME_FORTEZZA_KEA_INITIATOR_PUBLIC,
        -> cc_reference <- GCS_NULL,
       <-  retrieved_cc <- initiator_cc);

gcs_load_public_key        /* load initiator's (previously obtained) public key */
       <-  (minor_status,
        -> session_context,
        -> subject_cc <- recipient_cc,
        -> input_key_part <- initiator_public_key,
        -> key_part_type <- GCS_ONLY);
gcs_retrieve_cc            /* TEK CC */
```

```
       <-  (minor_status,
        -> session_context,
        -> domain_id,
        -> cc_name <- GCS_CC_NAME_FORTEZZA_TEK,
        -> cc_reference <- GCS_NULL,
       <-  retrieved_cc <- TEK_cc);
gcs_key_agreement /* form TEK, using the Fortezza KEA algorithm */
       <-  (minor_status,
        -> session_context,
        -> caller_cc <- recipient_cc,
        -> other_cc <- initiator_cc,
        -> pdu_in <- initRand,
        <-  pdu_out <- recipRand,
       <-  kak_cc <- TEK_cc);

       CI_GenerateRa
               -> (none,
              <-  pRa <- initRand, return value)

       CI_GenerateTEK      /* have received recipient's random number */
               -> (Flags <- CI_RECIPIENT_FLAG,/* recipient side */
               -> RegisterIndex <- TEKIndex,    /* where to place result */
               -> Ra <- initRand, Rb <- recipRand, /* random quantities */
               -> Size <- initPubSize, pY <- initPub, /* initiator's public */
              <-  return value);

gcs_retrieve_cc            /* MEK CC */
       <-  (minor_status,
        -> session_context,
        -> domain_id,
        -> cc_name <- GCS_CC_NAME_FORTEZZA_MEK,
        -> cc_reference <- GCS_NULL,
       <-  retrieved_cc <- MEK_cc);

/* By now, must have received TEK-wrapped MEK from initiator. */
/* Assume it's brought into a variable called "importedMEK" */

gcs_import_key             /* unwrap TEK-wrapped MEK */
       <-  (minor_status,
        -> session_context,
        -> kek_cc <- TEK_cc,
       <-> import_data <- importedMEK);
       <-> subject_cc <- MEK_cc);

       CI_UnwrapKey
               -> (UnwrapIndex <- TEKIndex,
               -> KeyIndex <- MEKIndex,
              <-  pKey <- wrappedKey,
              <-  return value);

/* Delete unnecessary CCs */
/* Begin using MEK to protect session */
```

# Appendix D: Future Directions

Functionality that may be encompassed by the scope of a future release of this specification includes:

- Data protection services
- CSF initialisation and management services
- CSF identification and authentication services
- CSF access control management
- Public key management services
- Key escrow support services

*Appendix E*

# *Generate Test Pattern and Verify Test Pattern Examples*

## E.1 Generate Test Pattern

```
Input:
1. 128-bit key K - This is either (1) a 64-bit key followed on the right
                   with 64 bits of binary zeros or (2) a 128-bit key.

Output:
1. 128-bit test pattern TP

Notation:
Let eK(X) denote DES encryption of 64-bit data X using key K.
Let KL denote the leftmost 64 bits of the input 128-bit key.
Let KR denote the rightmost 64 bits of the input 128-bit key.
Let TPL denote the leftmost 64 bits of the calculated test pattern.
Let TPR denote the rightmost 64 bits of the calculated test pattern.
Let X1, X2, X3, K2 denote 64-bit internal variables.
Let K1 denote X'4545454545454545'.

Process:
1. Set TPL to a 64-bit newly-generated random number.
2. Compute TPR:
   a. Set X1 to eK1(KL)
   b. Set K2 to X1 XOR KL
   c. Set X2 to KR XOR TPL
   d. Set X3 to eK2(X2)
   e. Set TPR to X2 XOR X3
3. Output test pattern as TPL concatenated to TPR.
```

## E.2 Verify Test Pattern

```
Input:
1. 128-bit key K - This is either (1) a 64-bit key followed on the right
                   with 64 bits of binary zeros or (2) a 128-bit key.
2. 128-bit trial test pattern TTP

Output:
1. Return code indicating trial test pattern verified or did not verify.

Notation:
Let eK(X) denote DES encryption of 64-bit data X using key K.
Let KL denote the leftmost 64 bits of the input 128-bit key.
Let KR denote the rightmost 64 bits of the input 128-bit key.
Let TPL denote the leftmost 64 bits of the calculated test pattern.
Let TPR denote the rightmost 64 bits of the calculated test pattern.
```

```
Let X1, X2, X3, K2 denote 64-bit internal variables.
Let K1 denote X'4545454545454545'.
Let TTPL denote the leftmost 64 bits of the trial test pattern.
Let TTPR denote the rightmost 64 bits of the trial test pattern.

Process:
1. Set TPL to TTPL
2. Compute TPR:
   a. Set X1 to eK1(KL)
   b. Set K2 to X1 XOR KL
   c. Set X2 to KR XOR TPL
   d. Set X3 to eK2(X2)
   e. Set TPR to X2 XOR X3
3. Check TPR for equality with TTPR
   a. If equal: success, test pattern verified
   b. If unequal: failure, test pattern did not verify
```

Implementation note: Steps 2 a through e of the Generate Test Pattern and Verify Test Pattern services are the same.

# Discussion on Key Parity

The NIST DES FIPS 46 originally stated that the 8th bit in each byte shall be used for parity. The ANSI X3.92 DEA stated that the 8th bit in each byte may be used for parity. FIPS 46-2 has been updated to use the word "may."

The use of the word "may" has some subtle implications. A conforming system may set parity and require parity to be set. Call such a system an SR system. A conforming system may ignore parity altogether. Call such a system an II system. A conforming system may try to compromise and set parity but not require it. Call such a system an SI system.

Let's see what kind of systems can talk to each other.

```
Sender  Receiver SR   II   SI
-------- --------------------
    SR   |        Y    Y    Y
    II   |        N    Y    Y
    SI   |        Y    Y    Y
```

The sender is the system that creates the key and the receiver is the system that wants to use the key, for example for decryption of a message encrypted at the sender system.

The point is that there is an incompatibility between valid conforming systems when an II system wants to send a key to an SR system. Another way to look at this is that an SR system is a universal sender (but not a universal receiver), an II system is a universal receiver (but not a universal sender) and an SI system is a universal sender and receiver. So if you want to design a system that can talk with all other conforming systems, a first thought is to do an SI system. However, things are not that simple.

ANSI X9.17 allows the specification of a "P" as a subparm which means that the key has parity and if it does not, you should fail. The lack of a "P" means that the parity should be ignored. That is, even if the parity is wrong, the operation should proceed. Now a system can decide to implement a portion of the standard and be conforming to that portion.

Let us see how each system can handle the P or no-P parm in a message as a sender or as a receiver.

| | send P | send no-P | receive P | receive no-P |
|---|---|---|---|---|
| SR | Y | Y | Y | N |
| II | N | Y | N | Y |
| SI | Y | Y | N | Y |

The interesting insight is that there is no solution in the above systems that handles all situations. This means that if you want to handle all valid conforming implementations, your system must do more than just set and ignore parity, it must process crypto service messages with keys using specific handling options.

In the most general case, one wants to allow the user during key export or key import to be able to specify: 1) IGNORE parity, 2) ENFORCE parity, or 3) ADJUST parity. Let's see how such functions could allow talking to anyone from any system, that is, let's see how the problems are handled.

An SR system cannot receive a No-P message. If you are on an II system exporting to an SR system, say ADJUST and send a P message. If you are on an SR system importing from an II system, say ADJUST.

An II system cannot send a P message. If you are on an II system exporting to an SR system, say ADJUST and send a P message. If you are on an SR system importing from an II system, say ADJUST.

An II or SI system cannot receive a P message. If you are on an II system importing from an SR or SI system, say ENFORCE. If you are on an SR or SI system exporting to an II system, send a no-P message.

Of course, you say IGNORE when you really do not care, such as when you are on an II system and are exporting to an II system.

# Glossary

**access control**

The prevention of unauthorised use of a resource including the prevention of use of a resource in an unauthorised manner (see ISO / IEC 7498-2).

**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 (see POSIX.0).

**algorithm context (ac)**

The definition of the algorithm(s) used by an implementation. The algorithm context may be keyed or non-keyed. See *cryptographic algorithm*.

**algorithm identifier**

An object ID that identifies the specific algorithm included in the *algorithm context*.

**algorithm specific parameters**

These are the parameters required by the algorithm specified in the algorithm context which are not specific to a single key to be used with the algorithm. Examples include key length and optional user group parameters for asymmetric algorithms.

**authenticated identity**

An identity of a principal that has been assured through authentication (see ISO / IEC 10081-2).

**authentication**

Verify claimed identity; see data origin authentication, and peer entity authentication (see ISO / IEC 7498-2).

**authorisation**

The granting of rights, which includes the granting of access based on access rights (see ISO / IEC 7498-2).

**authorisation policy**

A set of rules, part of an access control policy, by which access by security subjects to security objects is granted or denied. An authorisation policy may be defined in terms of access control lists, capabilities or attributes assigned to security subjects, security objects or both (see ECMA TR / 46).

**availability**

The property of being accessible and usable upon demand by an authorised entity (see ISO / IEC 7498-2).

**capability**

Users of the GCS-API are assigned *capabilities* which determine the authority they can exercise in use of the GCS-API functions. Four capabilities are defined, GCS_C_SELECTION, GCS_C_KEY_USAGE, GCS_C_KEY_PROTECTION, and GCS_C_ENCIPHER_DECIPHER.

**CC**

See *cryptographic context*.

**CC name**

The name for a cryptographic context which is unique within its *domain*.

**CC_reference**

The handle to a globally accessible and persistent *cryptographic context*. It comprises a *label*, a *storage unit class* and *storage unit instance*, a *domain identifier* and a *name*.

**ciphertext**

Data produced through the use of encipherment. The semantic content of the resulting data is not available (see ISO / IEC 7498-2).

**Note:** Ciphertext may itself be input to encipherment, such that super-enciphered output is produced.

**clear text**

Intelligible data, the semantic content of which is available (see ISO / IEC 7498-2).

**compromise**

A key is said to be *compromised* if its confidentiality is suspect. The threat of a key to compromise increases the longer the key is in use.

**confidentiality**

The property that information is not made available or disclosed to unauthorised individuals, entities, or processes (see ISO / IEC 7498-2).

**confounder**

Random information placed in front of cleartext before encipherment by a block cipher to prevent common header information included in the cleartext always being enciphered to the same ciphertext. (See also Initialisation Vector.)

**context confidentiality flag**

Indicates whether the private or secret values held in the *key context* are protected for confidentiality.

**context check value**

The *context check value* is a CSF internally generated and maintained check value for the protected *cryptographic context*.

**context id**

A unique identity assigned to a *cryptographic context* by the CSF when it is created.

**contextual information**

Information derived from the context in which an access is made (for example, time of day) (see ISO / IEC 10081-3).

**context type**

Specifies the type of algorithm context(s) included in the *cryptographic context*, ie., keyed, unkeyed, or both.

**context version number**

The version number of the *cryptographic context*. This specification defines the *context version number* as 0.

**credentials**

Data that is transferred to establish the claimed identity of an entity (see ISO / IEC 7498-2).

**cryptanalysis**
The analysis of a cryptographic system and its inputs and outputs to derive confidential variables and/or sensitive data including clear text.

**cryptographic algorithm**
A method of performing a cryptographic transformation (see cryptography) on a data unit. Cryptographic algorithms may be based on:

- symmetric key methods (the same key is used for both encipher and decipher transformations), or
- on asymmetric key methods (different keys are used for encipher and decipher transformations), or
- one way functions, which may or may not utilise a key, for the generation of a cryptographic hash value of input data.

**cryptographic aware**
Used to differentiate callers of the CSF. Cryptographic aware callers are those which are aware of the cryptographic policies used by the implementation.

**cryptographic checkvalue**
Information that is derived by performing a cryptographic transformation (see cryptography) on a data unit.

**Note:** The derivation of the checkvalue may be performed in one or more steps and is a result of a mathematical function of the key and data unit. It is usually used to check the integrity of a data unit.

**cryptographic context**
The cryptographic context is the set of information that defines the environment within which a particular cryptographic transform takes place. The information represents the cryptographic policy applicable and includes details of the permitted functions, algorithm(s) to be used, the key to be used and its current state. Within this specification the cryptographic context is deemed to comprise a header, a keyed and/or non-keyed algorithm context and a key context.

**cryptographic policy aware**
The name given to callers of the GCS-API who are responsible for establishing the cryptographic context of a set of operations through the selection of appropriate algorithm, generation of key and definition of key usage. These users are further categorised into *cryptographic policy selecting* or *cryptographic policy enforcing* users.

**cryptographic policy enforcing**
The name given to callers of the GCS-API who are responsible for enforcing cryptographic policy. Users may be *key usage policy enforcing* or *key protection policy enforcing* and have the GCS_C_KEY_USAGE or GCS_C_KEY_PROTECTION capabilities respectively. See *capability*.

**cryptographic policy selecting**
The name given to callers of the GCS-API who are capable of selecting which of a set of predefined cryptographic contexts is to be used for a particular set of services. These users have the GCS_C_SELECTION capability. See *capability*.

**cryptographic policy unaware**
The name given to callers of the GCS-API who are permitted to invoke cryptographic services within a previously defined cryptographic context.

**cryptographically strong random number**
A cryptographically strong number is one that does not have a period, is random, and might repeat.

**cryptographic unaware**
Used to differentiate callers of the CSF. Cryptographic unaware callers have no knowledge or understanding of the underlying cryptographic policies supported by the implementation of the CSF.

**cryptography**
The discipline that embodies principles, means, and the methods for the transformation of data in order to hide its information content, prevent its undetected modification and/or prevent its unauthorised use. (see ISO/IEC 7498-2).

**Note:** The choice of cryptography mechanism determines the methods used in encipherment and decipherment. An attack on a cryptographic principle, means or methods is cryptanalysis.

**CSF**
The Cryptographic Support Facility.

**data integrity**
The property that data has not been altered or destroyed in an unauthorised manner (see ISO/IEC 7498-2).

**data origin authentication**
The corroboration that the entity responsible for the creation of a set of data is the one claimed.

**decipherment**
The reversal of a corresponding reversible encipherment.

**decryption**
See decipherment.

**digital signature**
Data appended to, or a cryptographic transformation (see cryptography) of, a data unit that allows a recipient of the data unit to prove the source and integrity of the data unit and protect against forgery for example, by the recipient.

**encipherment**
The cryptographic transformation of data (see cryptography) to produce ciphertext.

**Note:** Encipherment may be irreversible, in which case the corresponding decipherment process cannot feasibly be performed. Such encipherment may be called a one-way-function or cryptochecksum.

**encryption**
See encipherment (see ISO/IEC 7498-2).

**end-to-end encipherment**
Encipherment of data within or at the source end system, with the corresponding decipherment occurring only within or at the destination end system (see ISO/IEC 7498-2).

**identification**
The assignment of a name by which an entity can be referenced. The entity may be high level (such as a user) or low level (such as a process or communication channel.

**initiator**
An entity (for example, human user or computer based entity) that attempts to access other entities (see ISO / IEC 10081-3).

**initialisation vector (IV)**
The *initialisation vector* is an algorithm specific parameter required for some symmetric key algorithms when used in a block cipher mode of operation. (See also confounder.) A static IV value may be defined as part of a key context (see *key context*) in which case the same value is used each time an IV is required. Alternatively, a caller may specify an IV value as an input parameter of those functions for which an IV is appropriate in which case a different IV value may be used for each call.

**integrity**
See Data Integrity (see ISO / IEC 7498-2).

**ITAR**
The US Government's International Traffic in Arms Regulations. This imposes constraints on the export of products containing cryptographic services, especially data confidentiality.

**key**
A sequence of symbols that controls the operations of encipherment and decipherment (see ISO / IEC 7498-2).

**KAK**
Key Archive Key.

**KEK**
Key Encryption Key.

**key context**
The *key context* contains information related to the use of a particular key instance. It comprises *key usage*, *permitted export mechanisms*, *key state*, *time of revocation*, *reason for revocation*, *key flag*, *key lifetime*, *initialisation vector*, *key specific parameters*, *split_protocol_type*, *key_part_number*, *number_of_key_parts* and *key value*.

**key flag**
The *key flag* refines the state of the key and provides control of the functions to which the key may be a target.

**key lifecycle**
A sequence of key states defined by the specification for a cryptographic key. These progress from pre-active, to active, active to quiescent, quiescent to de-activated, and deactivated to revoked. Other transitions can be effected by authorised callers of the CSF.

**key lifetime**
Defines the lifetime of the key.

**key management**
The generation, storage, distribution, deletion, archiving and application of keys in accordance with a security policy (see ISO / IEC 7498-2).

**key protection policy enforcing**
The name given to callers of the GCS-API who are responsible for the protection of cryptographic service and the key values it generates and uses. They may handle keys in the clear, and are assigned the GCS_C_KEY_PROTECTION capability. See *capability*.

**key specific parameters**
Additional mechanism specific parameters associated with the key.

**key state**
A defined set of states which can be assigned to a key. (see key lifecycle).

**key usage policy enforcing**
The name given to callers of the GCS-API who are responsible for key usage policy through the selection of appropriate algorithms and key usage parameters in creating CCs. They possess the GCS_C_KEY_USAGE capability. See *capability*

**key validity**
The key validity defines the period over which a key may be used for cryptographic transforms.

**key value**
The value of the key is implementation dependent.

**label**
The system defined name assigned to the cryptographic context stored in the operational storage unit maintained by the CSF.

**masquerade**
The unauthorised pretence by an entity to be a different entity (see ISO / IEC 7498-2).

**master key**
A cryptographic key used to protect other cryptographic keys during operational use. The Master Key is used to encipher the operational keys when they are handled or stored outside of the protected CSF environemnt.

**messaging application**
An application based on a store and forward paradigm; it requires an appropriate security context to be bound with the message itself.

**password**
Confidential authentication information, usually composed of a string of characters (see ISO / IEC 7498-2).

**physical security**
The measures used to provide physical protection of resources against deliberate and accidental threats (see ISO / IEC 7498-2).

**policy**
See security policy (see ISO / IEC 7498-2).

**principal**
An entity whose identity can be authenticated (see ISO / IEC 10081-2).

**private key**
A key used in an asymmetric algorithm. Possession of this key is restricted, usually to only one entity (see ISO / IEC 10081-1).

**permitted export mechanisms**
Defines which, if any, mechanisms may be used to transport the key contained in the CC between CSFs.

**public key**
The key, used in an asymmetric algorithm, that is publicly available (see ISO / IEC 10081-1).

**quality of protection (QOP)**
A label that implies methods of security protection under a security policy. This normally includes a combination of integrity and confidentiality requirements and is typically implemented in a communications environment by a combination of cryptographic

mechanisms.

**quasi-compromised**
Used to qualify a key which is suspected of being compromised.

**reason for revocation**
The reason given for revoking a key.

**repudiation**
Denial by one of the entities involved in a communication of having participated in all or part of the communication (see ISO / IEC 7498-2).

**seal**
A cryptographic checkvalue that supports integrity but does not protect against forgery by the recipient (that is, it does not support non-repudiation). When a seal is associated with a data element, that data element is *sealed* (see ISO / IEC 10081-1).

**secret key**
In a symmetric cryptographic algorithm the key shared between two entities (see ISO / IEC 10081-1).

**secure association**
An instance of secure communication (using communication in the broad sense of space and / or time) which makes use of a secure context.

**security attribute**
A security attribute is a piece of security information which is associated with an entity.

**security aware**
The caller of an API that is aware of the security functionality and parameters which may be provided by an API.

**security domain**
A set of elements, a security policy, a security authority and a set of security-relevant operations in which the set of elements are subject to the security policy, administered by the security authority, for the specified operations (see ISO / IEC 10081-1).

**security policy**
The set of criteria for the provision of security services (see also identity-based and rule-based security policy).

**security service**
A service which may be invoked directly or indirectly by functions within a system that ensures adequate security of the system or of data transfers between components of the system or with other systems.

**security unaware**
The caller of an API that is unaware of the security functionality and parameters which may be provided by an API.

**separation**
The concept of keeping information of different security classes apart in a system (see CESG Memo).

**Note:** Separation may be implemented by temporal, physical, logical or cryptographic techniques.

**session**
All CSF functions occur within the context of a session established between a caller and the CSF. A session commences with a call to *gcs_initialise_session*( ) to authenticate the caller's

identity and authorisation information and ends with a call to *gcs_terminate_session*( ) which releases the session_context. The *session_context* parameter returned by *gcs_initialise_session*( ) encapsulates the authenticated identity and authorisation information and has to be submitted as an input parameter to the GCS-API functions.

**short block policy**
Identifies the policy to apply if the caller submits a short block to a function call, e.g., X9.23 Padding or Reject.

**signature**
See digital signature (see ISO / IEC 7498-2).

**storage unit class**
distinguishes the device on which the cryptographic context is stored. See *CC reference*.

**storage unit instance**
Differentiates between different instances of the same storage unit class. See *CC reference*.

**strength of mechanism**
An aspect of the assessment of the effectiveness of a security mechanism, namely the ability of the security mechanism to withstand direct attack against deficiencies in its underlying algorithms, principles and properties (see ITSEC).

**SPI**
The system programming interface defined by this specification consists of functions for manipulating clear keys.

**target**
An entity to which access may be attempted (see ISO / IEC 10081-3).

**time of revocation**
The date and time at which the key was revoked. See *key context*.

**threat**
A potential violation of security (see ISO / IEC 7498-2).
An action or event that might prejudice security (see ITSEC).

**trust**
A relationship between two elements, a set of operations and a security policy in which element X trusts element Y if and only if X has confidence that Y behaves in a well defined way (with respect to the operations) that does not violate the given security policy (see ISO / IEC 10081-1).

**trusted functionality**
That which is perceived to be correct with respect to some criteria, for example, as established by a security policy (see ISO / IEC 7498-2).

**trusted third party**
A security authority or its agent, trusted by other entities with respect to security-related operations (see ISO / IEC 10081-1).

**vulnerability**
Weakness in an information system or components (for example, system security procedures, hardware design, internal controls) that could be exploited to produce an information-related misfortune (see Federal Criteria).

# Index