ProtectToolkit C
Programming Guide

SafeNet
THE DATA PROTECTION COMPANY

Part Number 007-008396-007
Version 5.0

Trademarks

Disclaimer

SafeNet makes no representations or warranties with respect to the contents of this document and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Furthermore, SafeNet reserves the right to revise this publication and to make changes from time to time in the content hereof without the obligation upon SafeNet to notify any person or organization of any such revisions or changes.

We have attempted to make these documents complete, accurate, and useful, but we cannot guarantee them to be perfect. When we discover errors or omissions, or they are brought to our attention, we endeavor to correct them in succeeding releases of the product.

SafeNet invites constructive comments on the contents of this document. Send your comments, together with your personal and/or company details to the address below:

SafeNet, Inc.
4690 Millennium Drive
Belcamp, Maryland USA 21017

Technical Support

If you encounter a problem while installing, registering or operating this product, please make sure that you have read the documentation. If you cannot resolve the issue, please contact your supplier or SafeNet support. SafeNet support operates 24 hours a day, 7 days a week. Your level of access to this service is governed by the support plan arrangements made between SafeNet and your organization. Please consult this support plan for further information about your entitlements, including the hours when telephone support is available to you.

Contact method	Contact information	
Address	SafeNet, Inc. 4690 Millennium Drive Belcamp, Maryland 21017 USA	
Phone	United States	(800) 545-6608, (410) 931-7520
	Australia and New Zealand	+1 410-931-7520
	China	(86) 10 8851 9191
	France	0825 341000
	Germany	01803 7246269
	India	+1 410-931-7520
	United Kingdom	0870 7529200, +1 410 931-7520
Web	www.safenet-inc.com	
Support and Downloads	www.safenet-inc.com/Support Provides access to the SafeNet Knowledge Base and quick downloads for various products.	

Technical Support Customer Portal	https://serviceportal.safenet-inc.com Existing customers with a Technical Support Customer Portal account can log in to manage incidents, get the latest software upgrades, and access the SafeNet Knowledge Base.

Revision History

Revision	Date	Reason
A	30 October 2014	Release 5.0
B	07 November 2014	Updates to the list of supported mechanisms.

TABLE OF CONTENTS

C H A P T E R 1
OVERVIEW

This product conforms to the API definition, produced by RSA Labs, named PKCS #11, and otherwise known as CRYPTOKI. ProtectToolkit C currently is compliant with PKCS#11 V 2.10.

The API provides a suite of cryptographic services for general-purpose usage and permanent key storage, which may be hosted by a physical token.

ProtectToolkit C is designed to operate in one of three separate modes:

- as a hardware implementation in conjunction with a compatible SafeNet cryptographic services adapter;
- in a client/server arrangement over a TCP/IP network, or
- in a Software Only mode of operation.

Within the client/server runtime environment, the server performs cryptographic processing at the request of the client. The server itself will only operate in the hardware runtime mode.

The software-only version is available for a variety of platforms including Windows NT and Solaris and is typically used as a development and testing environment for applications that will eventually use the hardware variant of ProtectToolkit C.

Chapter Contents

Runtime Licensing

All of the run-time software, including all applications and the software-only ProtectToolkit C run-time, supplied with this SDK, are licensed for development and testing purposes only. NO RUNTIME LICENSES ARE INCLUDED. Therefore this software, or any component of it, must not be used for production systems. Separate run-time licenses must be purchased for production systems deployed using any ProtectToolkit C support.

Please refer to the “readme.txt” file found in the install directory of the ProtectToolkit C SDK for further details regarding licensing requirements.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2
AN INTRODUCTION TO PKCS#11

The PKCS#11 Cryptographic Token Interface Standard is one of the Public KeyCryptography Standards developed by RSA Security. Also known as Cryptoki, this standard deals with defining the interface between an application and a cryptographic device. This chapter gives a general outline of PKCS#11 and some of its basic concepts. If unfamiliar with PKCS#11, the reader is strongly advised to refer to the PKCS#11 standard. This document can be obtained from the RSA Web site at http://www.rsasecurity.com/rsalabs/pkcs/.

The standard is also placed on the host system in a printable format during the SDK installation. PKCS#11 is used as a low-level interface to perform cryptographic operations without the need for the application to directly interface a device through its driver. PKCS#11 represents cryptographic devices using a common model referred to simply as a token. An application can therefore perform cryptographic operations on any device or token, using the same independent command set.

ProtectToolkit C is an Application Programming Interface (API) that conforms to the PKCS#11 standard.

The PKCS#11 Model

The model for PKCS#11 can be seen illustrated in Figure 1 and demonstrates how an application communicates its requests to a token via the PKCS#11 interface. The term slot represents a physical device interface. For example, a smart card reader would represent a slot and the smart card would represent the token. It is also possible that multiple slots may share the same token.

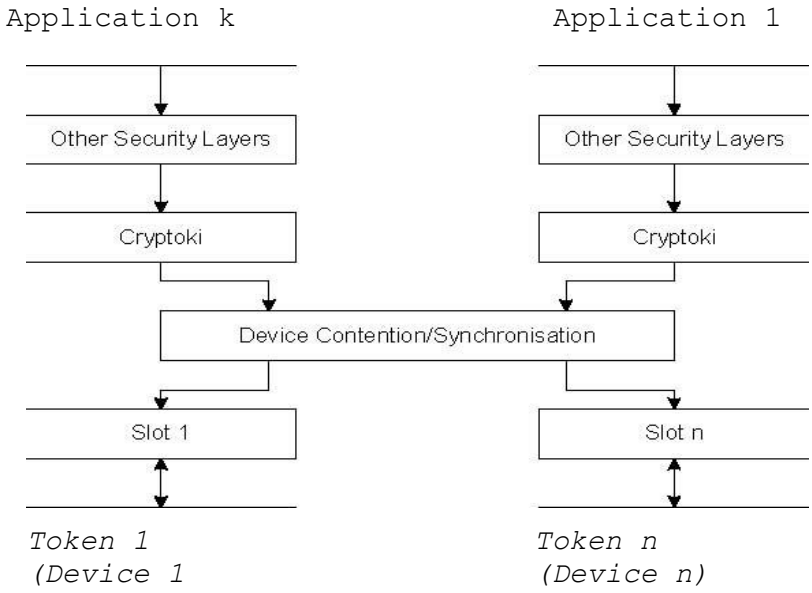

Figure 1 – General PKCS#11 Model

Within PKCS#11, a token is viewed as a device that stores objects and can perform cryptographic functions. Objects are generally defined in one of four classes:

- Data objects, which are defined by an application
- Certificate objects, which are digital certificates such as X.509
- Key objects, which can be public, private or secret cryptographic keys
- Vendor-defined objects

Objects within PKCS#11 are further defined as either a token object or a session object. Token objects are visible by any application which has sufficient access permission and is connected to that token. An important attribute of a token object is that it remains on the token until a specific action is performed to remove it.

A connection between a token and an application is referred to as a session. Session objects are temporary and only remain in existence while the session is open. In addition, session objects are only ever visible to the application that created them.

Access to objects within PKCS#11 is defined by the object type. Public objects are visible to any user or application, whereas private objects require that the user must be logged into that token in order to view them. PKCS#11 recognizes two types of users, namely a security officer (SO) or normal user. The security officer's only role is to initialize a token and set the normal users access PIN.

NOTE: The normal user, which manipulates objects and performs most operations, cannot log in until the security officer has set that user's PIN.

CHAPTER 3
ENVIRONMENTS

Application Environment

Win32™ Environment

ProtectToolkit C is supplied as a WIN32 Dynamic Link Library (`CRYPTOKI.DLL`) built with Microsoft development tools (MSVC). `CRYPTOKI2.LIB` is an import library that should be linked against applications to resolve function calls into "`CRYPTOKI.DLL`".

UNIX Environments

This is supplied as shared libraries. The hardware based ProtectToolkit C library is stored as the shared library `libcthsm.so` (`libcthsm.sl` for HP-UX on PA-RISC, `libcthsm.a` for AIX) and the software-only version as `libctsw.so` (`libctsw.sl` for HP-UX on PA-RISC, libctsw.a for AIX). The symbolic link libcryptoki.so (`libcryptoki.sl` for HP-UX on PA-RISC, `libcryptoki.a` for AIX) is setup in the /opt/safenet/protecttoolkit5/ptk/lib folder and should point to the appropriate library. Additionally these libraries must be included in the `LD_LIBRARY_PATH` (`SHLIB_PATH` for HP-UX on PA-RISC, or `LIBPATH` for AIX).

The `libcthsm shared object requires the library libethsm.`

For systems that support 32-bit and 64-bit, the **32-bit** libraries and executables are the default.

Java™ Environments

A lightweight proprietary Java wrapper for PKCS#11 API, *JCPROV*, is provided to allow access the ProtectToolkit C functionality from Java, without the overhead of the JCA/JCE API. The aim of this API is to be as similar to the PKCS#11 as the Java language allows. This provides a high-level of familiarity with the PKCS#11 environment and allows for faster implementation of Java programs.

The java API is compatible with JDK 1.3.1 or higher. The library is implemented in `jcprov.jar`, under the namespace `safenet_tech.jcprov`. An accompanying shared library "jcprov" (`jcprov.dll` in Win32 environments, and `libjcprov.so` in UNIX environments (`libjcprov.sl for HP-UX on PA-RISC, libjcprov.a` for AIX)) provides the native methods used to access the appropriate PKCS#11 library.

JCPROV Java JNI Support (AIX Only)

The Java VM on AIX does not support mixed mode JNI libraries. Mixed mode libraries are shared libraries that provide both 32-bit and 64-bit interfaces. It is therefore essential that the correct JNI library is selected for use with Java VM being used.

If using a 32-bit Java VM:

- The `/opt/safenet/protecttoolkit5/ptk/lib/libjcprov.a` symbolic link **must** point to a 32-bit version of the library (libjcprov_32.a).

  For example: /opt/safenet/protecttoolkit5/ptk/lib/libjcprov_32.a

- The `/opt/safenet/protecttoolkit5/ptk/lib/libjcryptoki.a` symbolic link **must** point a 32-bit version of the library (`libjcryptoki_32.a`).

  For example: /opt/safenet/protecttoolkit5/ptk/lib/libjcryptoki_32.a

If using a 64-bit Java VM:

- The `/opt/safenet/protecttoolkit5/ptk/lib/libjcprov.a` symbolic link **must** point to a 64-bit version of the library (`libjcprov_64.a`).

  For example: /opt/safenet/protecttoolkit5/ptk/lib/libjcprov_64.a

- The `/opt/safenet/protecttoolkit5/ptk/lib/libjcryptoki.a` symbolic link **must** point a 64-bit version of the library (`libjcryptoki_64.a`).

  For example: /opt/safenet/protecttoolkit5/ptk/lib/libjcryptoki_64.a

NOTE: When installing the ProtectToolkit C Runtime package, the above links are automatically created to use the **32-bit** versions of the JNI libraries.

Development Environment Guidelines

This manual gives a number of application development guidelines that can be of benefit for both novice and advanced developers using the ProtectToolkit C API.

An API tutorial is provided in Chapter 8, which is designed to show step-by-step development of a sample application.

Further sample programs, for which source code has been provided, may be compiled and linked against the supplied libraries. Further details about the sample programs are covered in Chapter 5.

The additional libraries "`ctextra`", "`ctutil`", "hex2bin" and "LMlib" are static libraries that contain additional PKCS#11 support and helper functions that are not a part of the PKCS#11 standard. For full details on the content of these libraries please refer to Chapter 11, 12, 13, and 15.

The library HSMAdmin call services on the HSM that are not part of the PKCS#11 standard – see Chapter 14 for more details.

This development kit may be used to build applications for any variant of the ProtectToolkit C runtimes including the software-only, the ProtectServer based or the remote client version.

NOTE: It is assumed that the Native C/C++ compiler is being used.

Compiling and Linking Applications on AIX

It is important that new applications link against libraries in the `/opt/safenet/protecttoolkit5/ptk/lib` directory **instead** of the libraries in the `/opt/safenet/protecttoolkit5/ptk/lib/legacy` directory. This can be achieved by using the *-L/opt/safenet/protecttoolkit5/ptk/lib* argument to the compiler or linker. Do **not** specify the `/opt/safenet/protecttoolkit5/ptk/lib/legacy` library path since the legacy shared libraries are deprecated, and support is to be removed in a future release.

It may also be desirable to explicitly specify an embedded library path when linking your own applications and libraries so that your applications automatically find the required libraries when they are run **without** requiring the `LIBPATH` environment variable to be set. This can be achieved by using the `-blibpath:/usr/lib:/lib:/opt/safenet/protecttoolkit5/ptk/lib` option to the linker (`ld`), or alternatively (if using the compiler to link.):

```
-Wl,-blibpath:/usr/lib:/lib:/opt/safenet/protecttoolkit5/ptk/lib
```

Compiling and Linking 64-bit Applications on AIX

To compile 64-bit applications for AIX specify the following compiler and linker flags:

```
-q64
```

Compiling and Linking 64-bit Applications for Solaris SPARC

To compile 64 bit applications for Solaris SPARC specify the following compiler flags:

```
-Xarch = v9
```

```
-DBITS64
```

The 64 bit libraries are to be found in the `/opt/safenet/protecttoolkit5/ptk/lib/sparcv9` directory. To link against them instead of the libraries in the directory `/opt/safenet/protecttoolkit5/ptk/lib`, add the following argument to the compiler or linker:

```
-L /opt/safenet/protecttoolkit5/ptk/lib/sparcv9
```

NOTE: It is assumed that the Sun C/C++ compiler is being used.

Compiling and Linking 64-bit Applications for HP-UX

To compile 64 bit applications for HP-UX specify the following compiler flags:

```
+DD64
```

The 64 bit libraries are to be found in the */opt/safenet/protecttoolkit5/ptk/lib/64* directory. To link against them instead of the libraries in the directory */opt/safenet/protecttoolkit5/ptk/lib*, add the following argument to the compiler or linker:

```
-L /opt/safenet/protecttoolkit5/ptk/lib/64
```

MSVC Project Settings

In order to remove link errors when linking to the additional libraries "`ctextra`" and "`ctutil`" etc, you need to set the MSVC project settings to "Multithreaded" under the C/C++ tab of the "Code generation" category, since this is what the libraries were compiled with.

Also add "_WINDOWS" to the "Preprocessor definitions" under the C/C++ tab of the "General" category.

Modes of Operation

To switch the operational mode of ProtectToolkit C from hardware to software, or remote client, you will need to ensure that you are linking to the correct "`CRYPTOKI.DLL`". There are three variants of this library depending on the operational mode. Refer to your installation guide or ask your system administrator as to where the different versions of this library were installed.

Configuration / Setup

For full details regarding setup and configuration of ProtectToolkit C and or ProtectServer hardware security modules (HSMs), please refer to the following manuals:

- *HSM Access Provider Install and Configuration Guide*
- *ProtectToolkit C Installation Guide*
- *ProtectToolkit C Administration Manual*

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4
THE SDK ENVIRONMENT

Objects

Cryptoki recognizes a number of object classes, as defined in the `CK_OBJECT_CLASS` data type. An object consists of a set of attributes, each of which has a given value. Each attribute that an object possesses has precisely one value. The following figure illustrates the high-level hierarchy of the Cryptoki objects and some of the attributes they support:

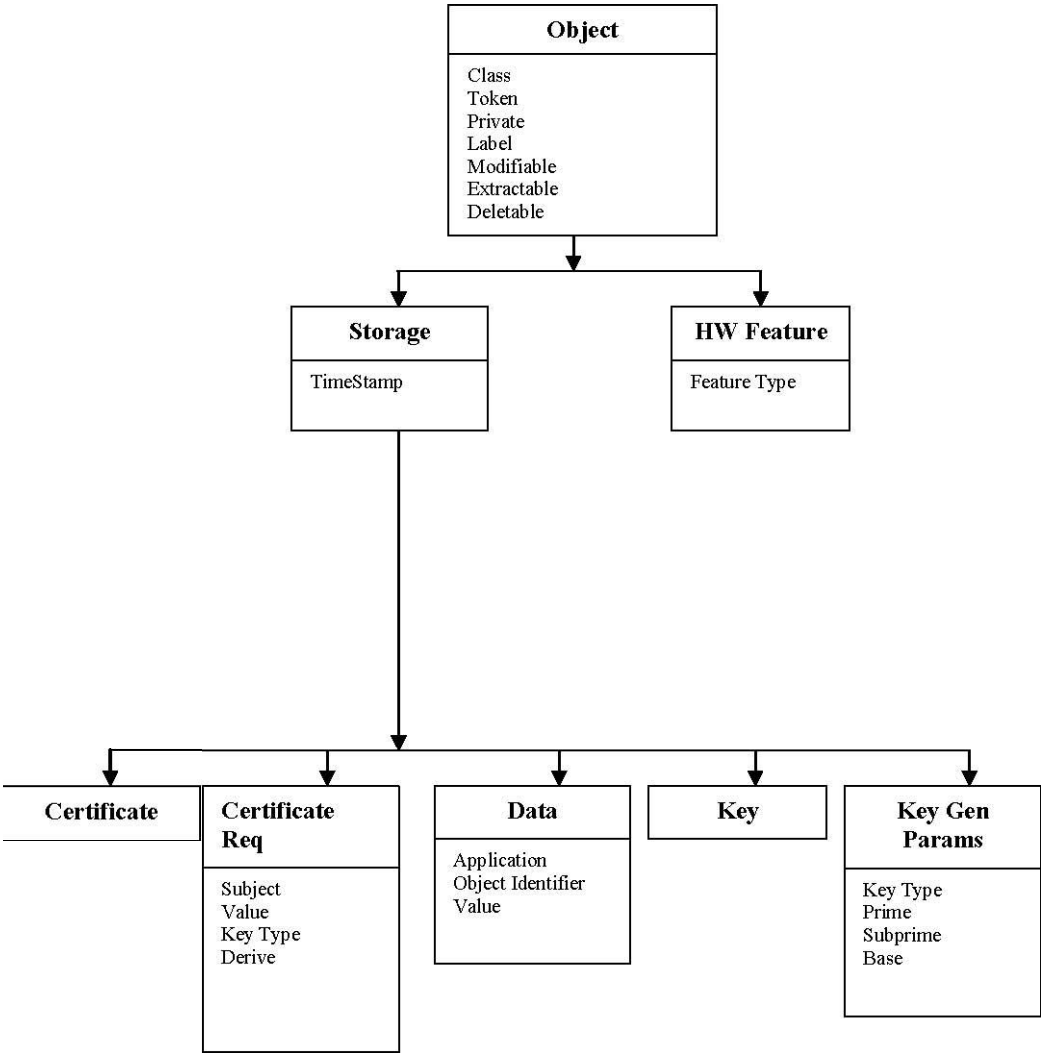

Figure 2 - Object Attribute Hierarchy

Cryptoki provides functions for creating, destroying, and copying objects and for obtaining or modifying their attribute values. Some of the cryptographic functions (for example, `C_GenerateKey`) also create key objects to hold their results.

Objects are always "well-formed" in Cryptoki—that is, an object always contains a minimum set of attributes for its proper operation, and the attributes are always consistent with one another from the time the object is created. However it is possible for an object to have one or more optional attributes missing.

A token can hold several identical objects. That is, it is permissible for two or more objects to have exactly the same values for all of their attributes.

Some object attributes possess default values, and need not be specified when creating an object. Some of these default values may even be the empty string (“”). Nevertheless, the object possesses these attributes. A given object has a single value for each attribute it possesses. Optional attributes are, by default, not created.

In addition to possessing Cryptoki attributes, objects may possess additional vendor-specific attributes. The meanings and values of the attributes not specified by Cryptoki are described below.

Creating, Modifying, Copying, and Deleting Objects

Cryptoki functions that create, modify, or copy objects, take a template as one of their arguments, where the template specifies attribute values. Cryptographic functions that create objects may also contribute some additional attribute values themselves. Which attributes have values contributed by a cryptographic function call depends on which cryptographic mechanism is being performed.

In any case, all the required attributes supported by an object class that do not have default values must be specified when an object is created, either in the template or by the function itself.

Creating Objects

Objects may be created with the Cryptoki functions `C_CreateObject`, `C_GenerateKey`, `C_GenerateKeyPair`, `C_UnwrapKey`, and `C_DeriveKey`. In addition, copying an existing object, with the function `C_CopyObject` or `CT_CopyObject`, also creates a new object.

Attempting to create an object with any of these functions requires an appropriate template to be supplied.

- If the supplied template specifies a value for an unrecognized attribute, then the attribute is stored but ignored.
- If the supplied template specifies an inappropriate value for a valid attribute, then the attribute is stored, except when it is the value attribute for a key in which case the length is checked. Checks are made on the validity of attributes when the object is used in later operations.
- When a token has the `CKF_LOGIN_REQUIRED` flag set in the flags field of the `CK_TOKEN_INFO` structure the token is read-only until the user (or SO) has been authenticated to the token.
- If the attribute values in the supplied template, together with any default attribute values and any attribute values contributed to the object by the object-creation function itself, are insufficient to fully specify the object to create, then the attempt will fail with the error code `CKR_TEMPLATE_INCOMPLETE`.
- If the supplied template specifies the same value for a particular attribute more than once (or the template specifies the same value for a particular attribute that the object-creation function itself contributes to the object), then the duplicate attribute is ignored.

Modifying Objects

If the “Increased Security” flag is set as part of the security policy, then `C_CopyObject` does not allow changing the `CKA_MODIFIABLE` flag from `FALSE` to `TRUE`. (See PTK C Administration Manual for details on setting HSM security policy).

Apart from the above exception, all PKCS#11 Version 2.10 rules applying to object modification are implemented.

Copying Objects

All PKCS#11 Version 2.10 rules applying to copying objects are implemented.

Deleting Objects

In addition to standard object deletion rules there is support for the `CKA_DELETABLE` attribute. This is an optional attribute that may be specified for token objects. For token objects with `CKA_DELETABLE` set to `FALSE` the `C_DestroyObject` function will not delete the object and will instead return the error `CKR_OBJECT_READ_ONLY`.

Additional Attribute Types

There are a number of additional vendor defined attribute types.

CKA_KEY_SIZE

The key size for key type CKK_EC can be any arbitrary bit length. That is, not within the byte boundary (for example, the key size for a P-521 curve).

The CKA_KEY_SIZE attribute has the following additional properties:

- Size is in bits
- Read-only attribute
- Assigned at object creation time
- Applicable to both private and public keys

NOTE: This attribute is applicable only to CKK_EC.

CKA_TIME_STAMP

Every object created is assigned a value for the `CKA_TIME_STAMP` attribute. This value is always read-only and may not be included in a template for a new object. However when an object is duplicated using the `C_CopyObject` function or the object is a key derived using the `C_DeriveKey` the new object will inherit the same creation time as the original object.

The value of this attribute is a text string encoding of the time. The encoding format is "`YYYYMMDDHHMMSS00`".

CKA_TRUSTED

This attribute may be included in a template for the creation of a Certificate object. It is used to indicate whether or not the certificate is **trusted** by the application. Once set the value of this attribute may not be modified.

The following values are defined for this attribute:

CKA_TRUSTED	Description
TRUE (1)	The certificate is trusted.
FALSE (0)	The certificate is not trusted and must be verified.

The value of `CKA_TRUSTED` may be set to `TRUE` only when the Security Officer is currently logged in. That is, the state of the session must be `CKS_RW_SO_FUNCTIONS`. Once a Certificate object has the `CKA_TRUSTED` attribute equal to `TRUE` the Certificate is considered a "trusted root certificate". The certificate validation code will stop once it reaches a trusted root certificate.

The certificate validation algorithm will locate the certificate's issuer by searching for a Certificate object with the `CKA_SUBJECT` attribute equal to the issuer's distinguished name. If located, it will then verify the signature on the certificate. If the signature is invalid it will return false, otherwise it will check the `CKA_TRUSTED` attribute on the issuer's certificate. If not equal to `TRUE` it will search for the issuer of that certificate. The algorithm will continue until a trusted certificate is found, the signature verification fails or the certificate chain is broken. The chain is broken when a certificate for the issuer cannot be found.

Once a certificate is marked as trusted the object's `CKA_VALUE` attribute may no longer be modified.

NOTE: The other attributes of the certificate will remain modifiable unless the `CKA_MODIFIABLE` attribute is set to false.

CKA_USAGE_COUNT

The value of this attribute maintains a count of the number of times a key object is used for a cryptographic operation. It is possible to set the value of this attribute for a key. Afterwards it is automatically incremented each time the key is used in a Cryptoki initialization routine (that is, `C_SignInit`).

Also see description for CKA_USAGE_LIMIT.

When generating Certificate objects with the `CKM_ENCODE_X_509` mechanism the `CKA_SERIAL_NUMBER` attribute for the new certificate object is taken from the certificate signing key's `CKA_USAGE_COUNT` attribute. The usage count from the private key is used only if the serial number is not already included in the template for the new certificate.

CKA_USAGE_LIMIT

This attribute represents the maximum number of times the object can be used. Simply - it is the highest possible CKA_USAGE_COUNT value allowed on this object.

This attribute may be specified when the object is created or added to an object when CKA_MODIFIABLE is true. Once the attribute is added it cannot be changed by the C_SetAttributeValue function.

Only the CKM_SET_ATTRIBUTES ticket mechanism can change this attribute. The Ticket can modify the attribute even if MODIFABLE=False.

CKA_START_DATE, CKA_END_DATE

These attributes control the period in which the object can be used.

These attributes may be specified when the object is created or added to an object when CKA_MODIFIABLE is true. Once the attribute is added it cannot be changed by the C_SetAttributeValue function.

Only the CKM_SET_ATTRIBUTES ticket mechanism can change these attributes. The Ticket can modify the attributes even if MODIFABLE=False.

Attribute validation is performed if these attributes are supplied during a C_CreateObject or C_UnWrapkey or C_DeriveKey operation. One or both of these attributes may be missing or be present but with an empty value. In this case the attribute is interpreted as "No restriction applies". For example if START_DATE is specified but END_DATE is not then the object will be usable from the start date onwards.

If the attribute is specified then it must be valid data structure - i.e. year is between 1900 and 9999, month from 01 to 12 and day from 01 to 31.

CKA_ADMIN_CERT

The CKA_ADMIN_CERT is a new Vendor defined Attribute.

This attribute is used to hold the certificate of an entity that can perform certain Management operations on that Object.

The value of the attribute is the DER encoding of a X509 v3 Public Key Certificate.

Rules for validation of the Certificate are: if it is self signed then it is implicitly trusted, if it signed by another entity then that Entities PKC must be present on the Token and be part of a chain terminating in a Cert marked CKA_TRUSTED=True.

It may be specified in the template when the Object is created, generated or imported. It may be added to an object with the C_SetAttributeValue command only if the CKA_MODIFIABLE is True and the attribute does not already exist i.e. once an object is created and made non-modifiable then the CKA_ADMIN_CERT cannot be later added.

The CKA_ADMIN_CERT is used with the CKM_SET_ATTRIBUTES Ticket Mechanism.

So if an object is not Modifiable and has no CKA_ADMIN_CERT then the CKM_SET_ATTRIBUTES Ticket Mechanism can never be applied to that object and its attributes are forever locked.

CKA_ISSUER_STR, CKA_SUBJECT_STR, CKA_SERIAL_NUMBER_INT

These attributes mirror the standard attributes (without the `_STR` or `_INT` suffix) but present that attribute as a printable value rather than as a DER encoding.

For the distinguished name attributes the string is encoded in the form: C=Country code, O=Organization, CN=Common Name, OU=Organizational Unit, L=Locality name, ST=State name.

These attributes may be supplied by an application in place of the DER encoded form and the other form of the attribute shall be derived from the one supplied in the template.

NOTE: `CKA_SERIAL_NUMBER_INT` is a Cryptoki Big Integer and not an intrinsic integer type. Therefore, its size is not constrained to 4 bytes.

CKA_PKI_ATTRIBUTE_BER_ENCODED

This attribute may be used to supply X.509 certificate extensions or PKCS#10 attribute values when creating these objects using the `CKM_ENCODE_X509` or `CKM_ENCODE_PKCS10` mechanisms respectively. Please refer to the sections #CKM_ENCODE_PKCS_10 and CKM_DECODE_X_509 for more details of these mechanisms.

The value of the `CKA_PKI_ATTRIBUTE_BER_ENCODED` is the BER encoded attribute.

CKA_EXPORT, CKA_EXPORTABLE

These attributes are similar to the standard `CKA_WRAP` and `CKA_EXTRACTABLE` attributes as they determine if a given key can wrap others keys and be extracted from the token in an encrypted form. The important difference between these attributes and their standard counterparts is that there are special controls on who can set the `CKA_EXPORT` flag. This flag may be set to true by the token's Security Officer or by the User if certain conditions are met. Thus the normal user can specify that a key may be exported in an encrypted form (by specifying that the `CKA_EXPORTABLE` attribute is true) but only by keys as determined by the SO (for example, a key that has the `CKA_EXPORT` attribute set to true).

The user may also specify the `CKA_EXPORT` attribute for keys that are generated internally and cannot be extracted other than by another key marked with `CKA_EXPORT`. This class of key may be used for transport keys where a master key encryption key (KEK) exists. In this case the Security Officer would create the KEK however the user could then create transport keys that could be exported only under the master KEK.

All other key usage attributes that might allow such a key, or any key exported by it, to be known outside the adapter must be set to `FALSE`. Specifically the template must specify `FALSE` for `CKA_EXTRACTABLE`, `CKA_DECRYPT`, `CKA_SIGN` and `CKA_MODIFIABLE` as well as `TRUE` for `CKA_SENSITIVE`, the template may also not specify `TRUE` for the `CKA_DERIVE` attribute.

CKA_DELETABLE

This attribute may be set on any token object (that is, where the `CKA_TOKEN` attribute is true) to specify that the object is permanent and may not be deleted. Once created, an object with the `CKA_DELETABLE` attribute set to false may be deleting only by re-initialization of the token (or during a hardware tamper process).

CKA_SIGN_LOCAL_CERT

This attribute must be set to true on any private key that is used with the Proof of origin mechanism (`CKM_ENOCDE_X_509_LOCAL_CERT`). Signing keys that do not have this attribute may not be used with this mechanism. For further information regarding this mechanism please refer to the sections CKM_WRAPKEY_DES3_ECB and CKM_WRAPKEY_DES3_CBC.

Keys with this attribute should have the `CKA_SIGN` and `CKA_ENCRYPT` attributes set to false to ensure that the key cannot be used to sign arbitrary data. Further special precautions should be taken to ensure that the key cannot leave the adapter – generally `CKA_EXTRACTABLE` and `CKA_EXPORTABLE` should be false and `CKA_SENSITIVE` should be true.

CKA_CHECK_VALUE

This attribute is a key check value that is calculated as follows:

- Take a buffer of the cipher block size of binary zeros (`0x00`).
- Encrypt this block in ECB mode.
- Take the first three bytes of cipher text as the check value.

This attribute is calculated on all keys of class `CKO_SECRET`, which means all symmetric key types when they are created or generated. The attribute is generated by default if it is not supplied in the key template. If it is supplied in the template, then the template value is used, even if its value would conflict with the one calculated as shown above. This is applicable when a customer wants to use an alternative method to validate a key.

NOTE: The `CKA_ENCRYPT` attribute is not required to be set to `TRUE` on the key object, in order for the check value attribute to be generated. This attribute cannot be changed once it has been set.

CKA_IMPORT

This attribute is similar to the standard `CKA_UNWRAP` attribute to determine if a given key can be used to unwrap encrypted key material. The important difference between these attributes and their standard counterparts is that if this attribute is set to True and `CKA_UNWRAP` attribute is set to False, then the only unwrap mechanism that can be used is `CKM_WRAPKEY_DES3_CBC`. With this combination, the error code `CKR_MECHANISM_INVALID` is returned for all other mechanisms. The default of `CKA_IMPORT` is set to `FALSE`.

CKA_CERTIFICATE_START_TIME; CKA_CERTIFICATE_END_TIME

These attributes are used to specify a user defined validity period for X.509 certificates. Without these, the certificate validity period is 1 year from the date and time of creation. The format is `YYYYMMDDhhmmss00`, which is identical to that defined for `utcTime` in `CK_TOKEN_INFO`.

CKA_MECHANISM_LIST

These attributes hold an array of CK_MECHANISM_TYPE values.
The CKA_MECHANISM_LIST attribute is used to restrict the operations that can be performed with any object containing it.

The following functions will check the object for the attribute, and if found, then the CK_MECHANISM_TYPE being requested must be present in the attribute else CKR_MECHANISM_INVALID error is returned:

- `C_Wrapkey`
- `C_Unwrapkey`
- `C_EncryptInit`
- `C_DecryptInit`
- `C_SignInit`
- `C_VerifyInit`
- `C_SignRecoverInit`
- `C_VerifyRecoverInit`

CKA_ENUM_ATTRIBUTE

This attribute is used to enumerate all the attributes of an object.

The attribute can only be passed in as part of a pTemplate parameter to the C_GetAttributeValue. It is never stored on an object.

Each PTK C session can hold an index value that is just used to support attribute enumeration.

Each call to C_GetAttributeValue using CKA_ENUM_ATTRIBUTE will return the next object attribute.

The error CKR_ATTRIBUTE_TYPE_INVALID is returned to indicate that the object has no more attributes.

A call to C_GetAttributeValue with the ulCount parameter set to zero will reset the index to zero.

Common Attributes

The following table defines the attributes common to all objects:

Table 1 – Common Object Attributes

Attribute	Data Type	Meaning
CKA_CLASS[1]	CK_OBJECT_CLASS	Object class (type)

[1] This attribute must be specified when the object is created

ProtectToolkit C supports the following Cryptoki Version 2.1 values for `CKA_CLASS` (that is, the following classes (types) of objects):

- CKO_HW_FEATURE
- CKO_DATA, CKO_CERTIFICATE
- CKO_PUBLIC_KEY
- CKO_PRIVATE_KEY
- CKO_SECRET_KEY

The following `CKA_CLASS` values are ProtectToolkit C extensions:

- CKO_CERTIFICATE_REQUEST
- CKO_CRL

Hardware Feature Objects

Hardware feature objects (CKO_HW_FEATURE) represent features of the device. They are created by the firmware on boot-up. The following figure illustrates the hierarchy of hardware feature objects and the attributes they support:

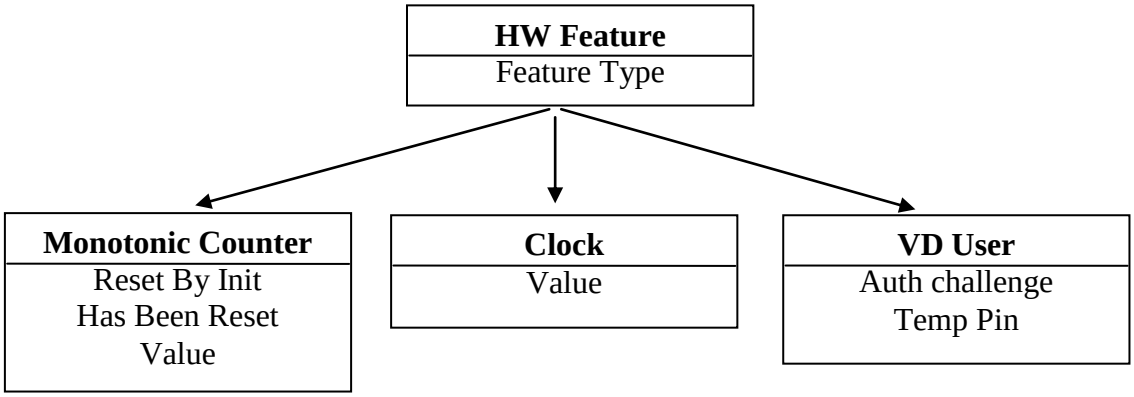

Figure 3 - Hardware Feature Object Attribute Hierarchy

Hardware feature objects act as an interface to a hardware feature and exist independent of the feature being represented. For example creating two clock objects does not imply that there are two clocks, just two interfaces to the one clock. Further, deleting the clock object does not affect the clock device in any way. However hardware feature objects may contain information independent of the feature being represented which may affect the behavior of the object. In addition the slot in which the object is created and the state of the session may affect the behavior of the object.

Table 2 – Hardware Feature Common Attributes

Attribute	Data Type	Meaning
CKA_HW_FEATURE_TYPE	CK_HW_FEATURE	Hardware feature (type)

ProtectToolkit C supports the following values for CKA_HW_FEATURE_TYPE:

- CKH_CLOCK
- CKH_MONOTONIC_COUNTER
- CKH_VD_USER

Clock Objects

Clock objects represent real-time clocks that exist on the device. This represents the same clock source as the utcTime field in the CK_TOKEN_INFO structure.

Table 3 – Clock Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE	CK_CHAR[16]	Current time as a character-string of length 16, represented in the format YYYYMMDDhhmmssxx (4 characters for the year; 2 characters each for the month, day, hour, minute and second; and 2 additional reserved characters set to 0).

The CKA_VALUE attribute may be set using the C_SetAttributeValue function if the object exists in the Admin Token and the session is in RW User Mode.

`C_SetAttributeValue` returns the error `CKR_USER_NOT_LOGGED_IN` to indicate that a different user type is required to set the value.

One object of this type is automatically created in the Admin token.

Monotonic Counter Objects

Monotonic counter objects represent hardware counters that exist on the device. In addition:

- The value of the counter is guaranteed to increase by one each time it is read.
- The monotonic counter is supported only on soft (non-smart card based) tokens and the value of the counter on each different token is the same.
- There is only one monotonic counter per token.
- The monotonic counter is automatically created whenever a token is initialized and exists by default on the Admin Token.
- The value is interpreted as a 160-bit big-endian binary integer (MSB on left).
- The Token SO may change the count value by setting the CKA_VALUE attribute.

Table 4 – Monotonic Counter Attributes

Attribute	Data Type	Meaning
CKA_RESET_ON_INIT[1]	CK_BBOOL	The value of the counter will reset to a previously returned value if the token is initialized using C_InitializeToken.
CKA_HAS_RESET[1]	CK_BBOOL	The value of the counter has been reset at least once at some point in time.
CKA_VALUE	Byte Array	The current version of the monotonic counter. The value is returned in big endian order. This is 20 bytes in size. Any attempt to set a value less than 20 bytes will fail.

[1] Read Only. The **CKA_VALUE** attribute may not be set by the client.

User Objects

User objects provide a means to obtain Authentication values i.e. these objects can be used when logging into a Token.

- The User object is supported only on soft (non-smart card based) tokens.
- The User Object is automatically created whenever a token is initialized.

The attributes of the User Object may be read to obtain an Authentication Challenge or to get a Temporary Pin.

For more details on the use of the User Object, refer to the description of the C_Login function.

Table 5 – User Attributes

Attribute	Data Type	Meaning
CKA_AUTH_CHALLENGE	CK_CHAR[16]	The current challenge value. Each time this attribute is read a new challenge value will be returned.
CKA_TEMP_PIN	CK_CHAR[32]	The current Temporary pin value. Each time this attribute is read a new pin value will be returned. A CKU_USER or CKU_SO must be logged in or else a read of this attribute will return CKR_USER_NOT_LOGGED_IN error. The pin returned can only be used to authenticate the same user that is currently logged in.

Storage Objects

Table 6 – Common Storage Object Attributes

Attribute	Data Type	Meaning
CKA_TOKEN	CK_BBOOL	TRUE if object is a token object. FALSE if object is a session object. Default is FALSE.
CKA_PRIVATE	CK_BBOOL	TRUE if object is a private object. FALSE if object is a public object. Default value is token-specific, and may depend on the values of other attributes of the object.
CKA_MODIFIABLE	CK_BBOOL	TRUE if object can be modified. FALSE if object can not be modified. Default is TRUE.
CKA_LABEL	RFC2279 string	Description of the object. Default is empty.

Only the CKA_LABEL attribute can be modified after the object is created. The CKA_TOKEN, CKA_PRIVATE, and CKA_MODIFIABLE attributes can be changed in the process of copying an object.

The CKA_TOKEN attribute identifies whether the object is a token object or a session object.

When the CKA_PRIVATE attribute is TRUE, a user may not access the object until the user has been authenticated to the token.

The value of the CKA_MODIFIABLE attribute determines whether or not an object is read-only.

ProtectToolkit C unmodifiable objects can be deleted. Objects may however specify CKA_DELETABLE to FALSE, for token objects only, in which case the object may not be deleted using the C_DestroyObject function. Only by re-initializing the token can the object be destroyed.

The CKA_LABEL attribute is intended to assist users in browsing.

Data Objects

Data objects (object class CKO_DATA) hold information defined by an application. Other than providing access to it, Cryptoki does not attach any special meaning to a data object. The following table lists the attributes supported by data objects, in addition to the common attributes listed in Table 1 and Table 6:

Table 7 – Data Object Attributes

Attribute	Data Type	Meaning
CKA_APPLICATION	RFC2279 string	Description of the application that manages the object (default empty)
CKA_OBJECT_ID	Byte Array	DER-encoding of the object identifier indicating the data object type (default empty)
CKA_VALUE	Byte array	Value of the object (default empty)

Each of these attributes may be modified after the object is created.

The CKA_APPLICATION attribute provides a means for applications to indicate ownership of the data objects they manage. However Cryptoki does not provide a means of ensuring that only a particular application has access to a data object.

The CKA_OBJECT_ID attribute provides an application an independent and expandable way to indicate the type of a data object. Cryptoki does not provide a means of insuring that the data object identifier matches the data object type.

Certificate Objects

The following figure illustrates details of certificate objects:

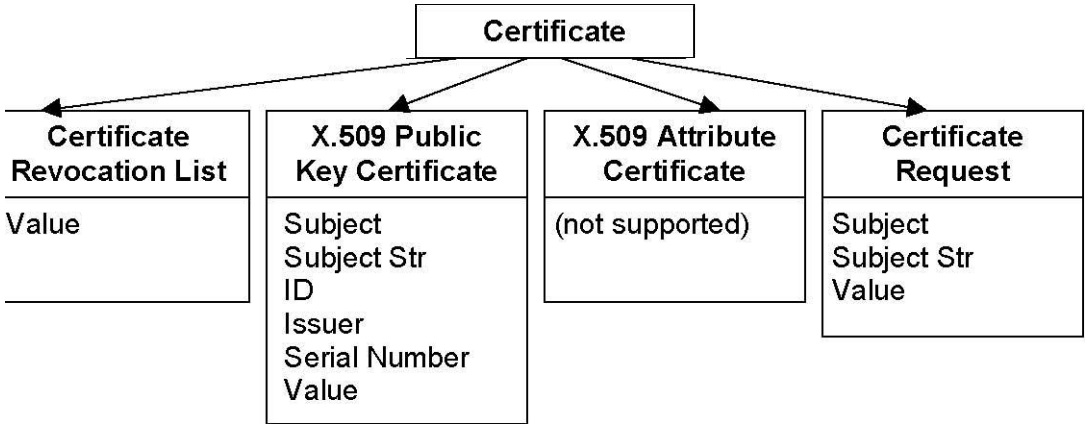

Figure 4 - Certificate Object Attribute Hierarchy

Certificate objects (object class `CKO_CERTIFICATE`) hold public-key or attribute certificates. Other than providing access to certificate objects, Cryptoki does not attach any special meaning to certificates. ProtectToolkit C however does include a number of extensions to Cryptoki that allows for more sophisticated certificate processing.

In addition to a number of extension attributes, it is possible to use a certificate object in place of a public key object. It is also possible to generate certificates (or certification requests) from public keys. Finally it is possible to introduce trusted certificates that allow for certificate path verification.

The following table defines the common certificate object attributes, in addition to the common attributes listed in Table 1 and Table 6:

Table 8 – Common Certificate Object Attributes

Attribute	Data Type	Meaning
CKA_CERTIFICATE_TYPE[1]	CK_CERTIFICATE_TYPE	Type of certificate
CKA_TRUSTED[2,3]	CK_BBOOL	Trust state of the object; see above description
CKA_DERIVE[2]	CK_BBOOL	Indicates if certificate can be used in derive mechanisms

[1] Must be specified when the object is created.
[2] SafeNet Extension
[3] May be specified as `TRUE` only by the Security Officer.
The `CKA_CERTIFICATE_TYPE` attribute may not be modified after an object is created.

X.509 Public Key Certificate Objects

X.509 certificate objects (certificate type `CKC_X_509`) hold X.509 public key certificates. The following table defines the X.509 certificate object attributes, in addition to the common attributes listed in Table 1, Table 6 and Table 8:

Table 9 – X.509 Certificate Object Attributes

Attribute	Data Type	Meaning
CKA_SUBJECT[1]	Byte array	DER-encoding of the certificate subject name
CKA_SUBJECT_STR[2]	Byte array	Printable representation of `CKA_SUBJECT` attribute
CKA_ID	Byte array	Key identifier for public/private key pair (default empty)
CKA_ISSUER	Byte array	DER-encoding of the certificate issuer name (default empty)
CKA_ISSUER_STR[2]	Byte array	Printable representation of `CKA_ISSUER` attribute
CKA_SERIAL_NUMBER	Byte array	DER-encoding of the certificate serial number (default empty)
CKA_SERIAL_NUMBER_INT[2]	Big Integer	Certificate serial number as an integer (default empty)
CKA_VALUE[1]	Byte array	BER-encoding of the certificate

[1] Must be specified when the object is created. [2] SafeNet Extension

Only the `CKA_ID`, `CKA_ISSUER` and `CKA_SERIAL_NUMBER` attributes may be modified after the object is created.

The `CKA_ID` attribute is intended to be a means of distinguishing multiple public/private key pairs held by the same subject (whether stored in the same token or not). Since subject names, as well as identifiers, distinguish keys, it is possible that keys that have different subjects may have the same `CKA_ID` value without introducing any ambiguity.

It is intended, in the interests of interoperability, that the subject name and key identifier for a certificate is to be the same as those for the corresponding public and private keys (though it is not required that all be stored in the same token). Cryptoki does not enforce this association or even the uniqueness of the key identifier for a given subject. In fact an application may leave the key identifier empty.

The `CKA_ISSUER` and `CKA_SERIAL_NUMBER` attributes are for compatibility with PKCS #7 and Privacy Enhanced Mail (RFC1421).

NOTE: With the version 3 extensions to X.509 certificates, the key identifier may be carried in the certificate. It is intended that the CKA_ID value be identical to the key identifier in such a certificate extension, however Cryptoki will not enforce this.

Certificate Request Objects

Certificate request objects (object class `CKO_CERTIFICATE_REQUEST`) hold a PKCS#10 certificate request. This object class is a vendor defined extension class. The following table defines the Certificate request object attributes, in addition to the common attributes listed in Table 1, Table 6 and Table 8:

Table 10 – Certificate Request Object Attributes

Attribute	Data Type	Meaning
CKA_SUBJECT	Byte array	DER-encoding of the certificate subject name
CKA_SUBJECT_STR[2]	Byte array	Printable representation of CKA_SUBJECT attribute
CKA_VALUE[1]	Byte array	BER-encoding of the certificate

[1] Must be specified when the object is created. [2] SafeNet Extension

Certificate Revocation List

Certificate Revocation List (CRL) objects (object class `CKO_CRL`) hold a certificate revocation list. This object class is a vendor defined extension class.

The following table defines the CRL object attributes, in addition to the common attributes listed in Table 1, Table 6 and Table 8:

Table 51 – Certificate Revocation Object Attributes

Attribute	Data Type	Meaning
CKA_SUBJECT	Byte array	DER-encoding of the certificate subject name
CKA_SUBJECT_STR[2]	Byte array	Printable representation of CKA_SUBJECT attribute
CKA_VALUE[1]	Byte array	BER-encoding of the certificate

[1] Must be specified when the object is created. [2] SafeNet Extension

Key Objects

The following figure illustrates details of key objects:

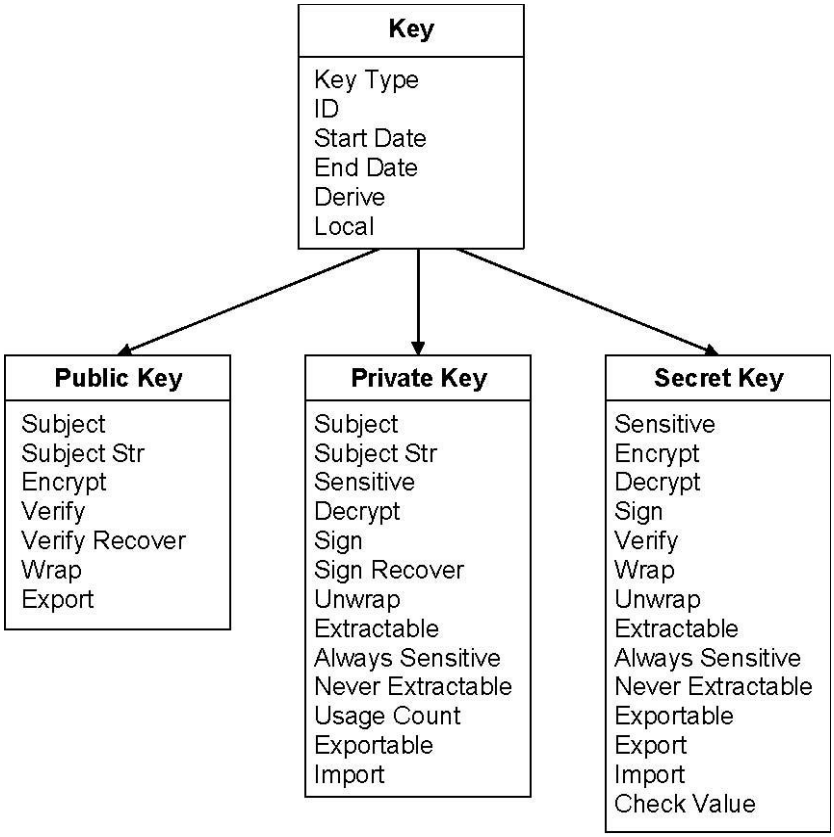

Figure 5 - Key Attribute Detail

Key objects hold encryption or authentication keys, which can be public keys, private keys, or secret keys. The following common footnotes apply to all the tables describing attributes of keys:

Table 6 – Common footnotes for key attribute tables

[1] Must be specified when object is created with `C_CreateObject`.
[2] Must *not* be specified when object is created with `C_CreateObject`.
[3] Must be specified when object is generated with `C_GenerateKey` or `C_GenerateKeyPair`.
[4] Must *not* be specified when object is generated with `C_GenerateKey` or `C_GenerateKeyPair`.
[5] Must be specified when object is unwrapped with `C_UnwrapKey`.
[6] Must *not* be specified when object is unwrapped with `C_Unwrap`.
[7] Cannot be revealed if object has `CKA_SENSITIVE` attribute set to `TRUE` or its `CKA_EXTRACTABLE` attribute set to `FALSE`.
[8] May be modified after object is created with a `C_SetAttributeValue` call, or in the process of copying object with a `C_CopyObject` call. As mentioned previously, however, it is possible that a particular token may not permit modification of the attribute.
[9] Default value is token-specific, and may depend on the values of other attributes.
[10] SafeNet Extension

The following table defines the attributes common to public key, private key and secret key classes, in addition to the common attributes listed in Table 1 and Table 6:

Table 7 – Common Key Attributes

Attribute	Data Type	Meaning
CKA_KEY_TYPE[1,3,5]	CK_KEY_TYPE	Type of key
CKA_ID[8]	Byte array	Key identifier for key (default empty)
CKA_START_DATE[8]	CK_DATE	Start date for the key (default empty). If not empty then the attribute holds starting date for the key.
CKA_END_DATE[8]	CK_DATE	End date for the key (default empty). If not empty then the attribute holds expiry date for the key.
CKA_ADMIN_CERT[10]	Byte array	DER encoded certificate of the key administrator. See more details in the discussion on Key Usage Limits.
CKA_DERIVE[8]	CK_BBOOL	TRUE if key supports key derivation (that is, if other keys can be derived from this one (default `FALSE`)
CKA_LOCAL[2,4,6]	CK_BBOOL	`TRUE` only if key was either • generated locally (that is, on the token) with a `C_GenerateKey`or `C_GenerateKeyPair`call • created with a `C_CopyObject`call as a copy of a key which had its `CKA_LOCAL` attribute set to `TRUE`
CKA_MECHANISM_LIST[10]	CKA_MECHANISM_ TYPE array	List of allowable mechanisms that can be used. For more information see the entry for this attribute in the *Additional Attribute Types* section above.

Public Key Objects

Public key objects (object class `CKO_PUBLIC_KEY`) hold public keys. This version of Cryptoki recognizes four types of public keys: RSA, DSA, Diffie-Hellman and Elliptic Curve. The following table defines the attributes common to all public keys, in addition to the common attributes listed in Table 1, Table 6, and Table 13:

Table 84 – Common Public Key Attributes

Attribute	Data Type	Meaning
CKA_SUBJECT[8]	Byte array	DER-encoding of the key subject name (default empty)
CKA_SUBJECT_STR[10]	Byte array	Printable version of CKA_SUBJECT
CKA_ENCRYPT[8]	CK_BBOOL	TRUE if key supports encryption [9]
CKA_VERIFY[8]	CK_BBOOL	TRUE if key supports verification where the signature is an appendix to the data [9]
CKA_VERIFY_RECOVER[8]	CK_BBOOL	TRUE if key supports verification where the data is recovered from the signature [9]
CKA_WRAP[8]	CK_BBOOL	TRUE if key supports wrapping (that is, can be used to wrap other keys)[9]
CKA_EXPORT[10]	CK_BBOOL	TRUE if the key may be used to export Exportable keys.

It is intended in the interests of interoperability that the subject name and key identifier for a public key is to be the same as those for the corresponding certificate and private key. However, this is not enforced, and it is not required that the certificate and private key be stored on the same token.

To map between ISO/IEC 9594-8 (X.509) key usage flags for public keys and the PKCS #11 attributes for public keys, use the following table. ProtectToolkit C does not enforce these usage flags. When a certificate object is created it may have any of the standard Cryptoki usage attributes, which is enforced.

Table 15 – Mapping of X.509 key usage flags to Cryptoki attributes for public keys

Key Usage Flags for Public Keys in X.509 Public Key Certificates	Corresponding Cryptoki Attributes for Public Keys
dataEncipherment	CKA_ENCRYPT
digitalSignature, keyCertSign, cRLSign	CKA_VERIFY
digitalSignature, keyCertSign, cRLSign	CKA_VERIFY_RECOVER
keyAgreement	CKA_DERIVE
keyEncipherment	CKA_WRAP
nonRepudiation	CKA_VERIFY
nonRepudiation	CKA_VERIFY_RECOVER

RSA Public Key Objects

RSA public key objects (object class `CKO_PUBLIC_KEY`**,** key type `CKK_RSA`) hold RSA public keys. The following table defines the RSA public key object attributes, in addition to the common attributes listed in Table 1, Table 6, Table 13, and Table 14:

Table 9 – RSA Public Key Object Attributes

Attribute	Data Type	Meaning
CKA_MODULUS[1,4,6]	Big integer	Modulus *n*
CKA_MODULUS_BITS[2,3,6]	CK_ULONG	Length in bits of modulus *n*
CKA_PUBLIC_EXPONENT[1,3,6]	Big integer	Public exponent *e*

Depending on the token, there may be limits on the length of key components. See PKCS #1 for more information on RSA keys.

DSA Public Key Objects

DSA public key objects (object class `CKO_PUBLIC_KEY`, key type `CKK_DSA`) hold DSA public keys. The following table defines the DSA public key object attributes, in addition to the common attributes listed in Table 1, Table 6, Table 13, and Table 14:

Table 17 – DSA Public Key Object Attributes

Attribute	Data Type	Meaning
CKA_PRIME[1,3,6]	Big integer	Prime *p* (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME[1,3,6]	Big integer	Subprime *q* (160 bits)
CKA_BASE[1,3,6]	Big integer	Base *g*
CKA_VALUE[1,4,6]	Big integer	Public value *y*

The CKA_PRIME, CKA_SUBPRIME and CKA_BASE attribute values are collectively the "DSA parameters".

Diffie-Hellman Public Key Objects

Diffie-Hellman public key objects (object class CKO_PUBLIC_KEY, key type CKK_DH) hold Diffie-Hellman public keys. The following table defines the Diffie-Hellman public key object attributes, in addition to the common attributes listed in Table 1, Table 6, Table 13, and Table 14:

Table 108 – Diffie-Hellman Public Key Object Attributes

Attribute	Data Type	Meaning
CKA_PRIME[1,3,6]	Big integer	Prime *p*
CKA_BASE[1,3,6]	Big integer	Base *g*
CKA_VALUE[1,4,6]	Big integer	Public value *y*

The CKA_PRIME and CKA_BASE attribute values are collectively the "Diffie-Hellman parameters". Depending on the token, there may be limits on the length of the key components. See PKCS #3 for more information on Diffie-Hellman keys.

Elliptic Curve Public Key Objects

EC (also related to ECDSA) public key objects (object class CKO_PUBLIC_KEY, key type CKK_EC or CKK_ECDSA in PKCS#11 v2.10) hold EC public keys. The following table defines the EC public key object attributes, in addition to the common attributes listed in Table 1, Table 6, Table 13, and Table 14:

Table 119 – Elliptic Curve Public Key Object Attributes

Attribute	Data Type	Meaning
CKA_EC_PARAMS[1,3] (CKA_ECDSA_PARAMS)	Byte Array	DER-encoding of an ANSI X9.62 Parameters value
CKA_POINT[1,4]	Byte Array	DER-encoding of an ANSI X9.62 ECPoint value *Q*

The CKA_EC_PARAMS or CKA_ECDSA_PARAMS attribute value is known as the "EC domain parameters" and is defined in ANSI X9.62 as a choice of three parameter representation methods with the following syntax:

```
Parameters ::= CHOICE {
ecParameters ECParameters,
namedCurve CURVES.&id({CurveNames}),
implicitlyCA NULL
}
```

This allows detailed specification of all required values using choice ecParameters, the use of a namedCurve as an object identifier substitute for a particular set of elliptic curve domain parameters, or implicitlyCA to indicate that the domain parameters are explicitly defined elsewhere. The use of a namedCurve is recommended over the choice ecParameters. The choice implicitlyCA must not be used in Cryptoki.

Both the namedCurve and ecParameters methods are supported in ProtectToolkit C, see CKM_EC_KEY_PAIR_GEN mechanism for details.

Private Key Objects

Private key objects (object class `CKO_PRIVATE_KEY`) hold private keys. This version of ProtectToolkit C recognizes four types of private key: RSA, DSA, Diffie-Hellman and Elliptic Curve. The following defines the attributes common to all private keys, in addition to the common attributes listed in Table 1, Table 6, and Table 13.

Table 20 – Common Private Key Attributes

Attribute	Data Type	Meaning
CKA_SUBJECT[8]	Byte array	DER-encoding of certificate subject name (default empty)
CKA_SUBJECT_STR[10]	Byte array	Printable version of CKA_SUBJECT (default empty)
CKA_SENSITIVE[8] (see below)	CK_BBOOL	TRUE if key is sensitive[9]
CKA_SECONDARY_AUTH	CK_BBOOL	This is not supported.
CKA_AUTH_PIN_FLAGS[2,4,6]	CK_FLAGS	This is not supported.
CKA_DECRYPT[8]	CK_BBOOL	TRUE if key supports decryption[9]
CKA_SIGN[8]	CK_BBOOL	TRUE if key supports signatures where the signature is an appendix to the data[9]
CKA_SIGN_RECOVER[8]	CK_BBOOL	TRUE if key supports signatures where the data can be recovered from the signature[9]
CKA_UNWRAP[8]	CK_BBOOL	TRUE if key supports unwrapping (that is, can be used to unwrap other keys)[9]
CKA_EXTRACTABLE[8] (see below)	CK_BBOOL	TRUE if key is extractable[9]
CKA_ALWAYS_SENSITIVE[2,4,6]	CK_BBOOL	TRUE if key has *always* had the CKA_SENSITIVE attribute set to TRUE
CKA_NEVER_EXTRACTABLE[2,4,6]	CK_BBOOL	TRUE if key has *never* had the CKA_EXTRACTABLE attribute set to TRUE
CKA_USAGE_COUNT[10]	CK_ULONG	This optional field will hold a usage counter. The numeric value is incremented each time the key is used.
CKA_EXPORTABLE[10]	CK_BBOOL	`TRUE` if key may be wrapped with a key that has the CKA_`EXPORT` attribute set.
CKA_IMPORT[10]	CK_BBOOL	If `TRUE` and CKA_UNWRAP is `FALSE` supports unwrapping only using CKM_WRAPKEY_DES3_CBC.

RSA Private Key Objects

RSA private key objects (object class CKO_PRIVATE_KEY, key type CKK_RSA) hold RSA private keys. The following table defines the RSA private key object attributes, in addition to the common attributes listed in Table 1, Table 6, Table 13, and Table 19:

Table 121 – RSA Private Key Object Attributes

Attribute	Data Type	Meaning
CKA_MODULUS[1,4,6]	Big integer	Modulus *n*
CKA_PUBLIC_EXPONENT[4,6]	Big integer	Public exponent *e*
CKA_PRIVATE_EXPONENT[1,4,6,7]	Big integer	Private exponent *d*
CKA_PRIME_1[4,6,7]	Big integer	Prime *p*
CKA_PRIME_2[4,6,7]	Big integer	Prime *q*
CKA_EXPONENT_1[4,6,7]	Big integer	Private exponent *d* modulo *p*-1
CKA_EXPONENT_2[4,6,7]	Big integer	Private exponent *d* modulo *q*-1
CKA_COEFFICIENT[4,6,7]	Big integer	CRT coefficient q^{-1} mod *p*

RSA modulus size may range from 512 to 4096 bits (or 1024 to 4096 bits in FIPS mode). RSA private keys can include all CRT components or just the modulus and exponent. Performance is greatly enhanced by providing all CRT components so this is advised. Any RSA keys generated locally will always include all components.

NOTE: When generating an RSA private key, there is no CKA_MODULUS_BITS attribute specified. This is because RSA private keys are only generated as part of an RSA key *pair*, and the CKA_MODULUS_BITS attribute for the pair is specified in the template for the RSA public key.

DSA Private Key Objects

DSA private key objects (object class CKO_PRIVATE_KEY, key type CKK_DSA) hold DSA private keys. The following table defines the DSA private key object attributes, in addition to the common attributes listed in Table 1, Table 6, Table 13, and Table 19:

Table 132 – DSA Private Key Object Attributes

Attribute	Data Type	Meaning
CKA_PRIME[1,4,6]	Big integer	Prime p (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME[1,4,6]	Big integer	Subprime q (160 bits)
CKA_BASE[1,4,6]	Big integer	Base g
CKA_VALUE[1,4,6,7]	Big integer	Private value x

The CKA_PRIME, CKA_SUBPRIME and CKA_BASE attribute values are collectively the "DSA parameters". See FIPS PUB 186 for more information on DSA keys.

NOTE: When generating a DSA private key, the DSA parameters are **not** specified in the key's template. This is because DSA private keys are only generated as part of a DSA key *pair*, and the DSA parameters for the pair are specified in the template for the DSA public key. If they are present in the private key template they are ignored.

Diffie-Hellman Private Key Objects

Diffie-Hellman private key objects (object class CKO_PRIVATE_KEY, key type CKK_DH) hold Diffie-Hellman private keys. The following table defines the Diffie-Hellman private key object attributes, in addition to the common attributes listed in Table 1, Table 6, Table 13, and Table 19:

Table 23 – Diffie-Hellman Private Key Object Attributes

Attribute	Data Type	Meaning
CKA_PRIME[1,4,6]	Big integer	Prime *p*
CKA_BASE[1,4,6]	Big integer	Base *g*
CKA_VALUE[1,4,6,7]	Big integer	Private value *x*
CKA_VALUE_BITS[2,6]	CK_ULONG	Length in bits of private value *x*

The CKA_PRIME and CKA_BASE attribute values are collectively the "Diffie-Hellman parameters". Depending on the token, there may be limits on the length of the key components. See PKCS #3 for more information on Diffie-Hellman keys.

NOTE: When generating a Diffie-Hellman private key, the Diffie-Hellman parameters are *not* specified in the key's template. This is because Diffie-Hellman private keys are only generated as part of a Diffie-Hellman key *pair*, and the Diffie-Hellman parameters for the pair are specified in the template for the Diffie-Hellman public key. If they are present in the private key template they are ignored.

Elliptic Curve Private Key Objects

EC (also related to ECDSA) private key objects (object class CKO_PRIVATE_KEY, key type CKK_EC or CKK_ECDSA in PKCS#11 v2.10) hold EC private keys. The following table defines the EC private key object attributes, in addition to the common attributes listed in Table 1, Table 6, Table 13, and Table 20:

Table 24 – Elliptic Curve Private Key Object Attributes

Attribute	Data Type	Meaning
CKA_EC_PARAMS[1,4,6] (CKA_ECDSA_PARAMS)	Byte Array	DER-encoding of an ANSI X9.62 Parameters value
CKA_POINT[1,4,6,7]	Byte Array	ANSI X9.62 private value *d*

The CKA_EC_PARAMS or CKA_ECDSA_PARAMS attribute value is known as the "EC domain parameters" and is defined in ANSI X9.62 as a choice of three parameter representation methods with the following syntax:

```
Parameters ::= CHOICE {
ecParameters ECParameters,
namedCurve CURVES.&id({CurveNames}),
implicitlyCA NULL
}
```

This allows detailed specification of all required values using choice ecParameters, the use of a namedCurve as an object identifier substitute for a particular set of elliptic curve domain parameters, or implicitlyCA to indicate that the domain parameters are explicitly defined elsewhere. The use of a namedCurve is recommended over the choice ecParameters. The choice implicitlyCA **must not** be used in Cryptoki.

Both the ecParameters and the namedCurve method are supported in ProtectToolkit C. See CKM_EC_KEY_PAIR_GEN mechanism for details.

NOTE: When generating an EC private key, the EC domain parameters are not specified in the key's template. This is because EC private keys are generated only as part of an EC key pair, and the EC domain parameters for the pair are specified in the template for the EC public key.

Secret Key Objects

Secret key objects (object class `CKO_SECRET_KEY`) hold secret keys. This version of Cryptoki recognizes the following types of secret key: generic, RC2, RC4, DES, DES2, DES3, CAST128 (also known as CAST5), IDEA, and AES. The following table defines the attributes common to all secret keys, in addition to the common attributes listed in Table 1, Table 6, and Table 13:

Table 25 – Common Secret Key Attributes

Attribute	Data Type	Meaning
CKA_SENSITIVE[8] (see below)	CK_BBOOL	TRUE, if object is sensitive (default FALSE)
CKA_ENCRYPT[8]	CK_BBOOL	TRUE, if key supports encryption[9]
CKA_DECRYPT[8]	CK_BBOOL	TRUE, if key supports decryption[9]
CKA_SIGN[8]	CK_BBOOL	TRUE, if key supports signatures (that is, authentication codes) where the signature is an appendix to the data[9]
CKA_VERIFY[8]	CK_BBOOL	TRUE, if key supports verification (that is, of authentication codes) where the signature is an appendix to the data[9]
CKA_WRAP[8]	CK_BBOOL	TRUE, if key supports wrapping (that is, can be used to wrap other keys)[9]
CKA_UNWRAP[8]	CK_BBOOL	TRUE, if key supports unwrapping (that is, can be used to unwrap other keys)[9]
CKA_EXTRACTABLE[8] (see below)	CK_BBOOL	TRUE, if key is extractable[9]
CKA_ALWAYS_SENSITIVE[2,4,6]	CK_BBOOL	TRUE if key has always had the CKA_SENSITIVE attribute set to TRUE
CKA_NEVER_EXTRACTABLE[2,4,6]	CK_BBOOL	TRUE, if key has never had the CKA_EXTRACTABLE attribute set to TRUE
CKA_SUBJECT[8]	Byte array	DER-encoding of certificate subject name (default empty)
CKA_EXPORT[10]	CK_BBOOL	TRUE, if the key may be used to wrap Exportable keys. Restrictions apply on who can set this attribute to TRUE.
CKA_EXPORTABLE[10]	CK_BBOOL	TRUE, if key may be wrapped with a key attribute set with CKA_EXPORT.
CKA_IMPORT[10]	CK_BBOOL	If `TRUE` and CKA_UNWRAP is `FALSE` supports unwrapping only using CKM_WRAPKEY_DES3_CBC.
CKA_CHECK_VALUE	Byte Array	A calculated key check value. Fixed size of 3 bytes.

After an object is created, the `CKA_SENSITIVE` attribute may be changed, but only to the value `TRUE`. Similarly, after an object is created, the `CKA_EXTRACTABLE` attribute may be changed, but only to the value `FALSE`. Attempts to make other changes to the values of these attributes should return the error code `CKR_ATTRIBUTE_READ_ONLY`.

If the CKA_SENSITIVE attribute is TRUE, or if the CKA_EXTRACTABLE attribute is FALSE, then certain attributes of the secret key cannot be revealed in plain text outside the token. The attributes that are affected by the sensitive and extractable attributes are specified by the 7-superscript in the attribute table, in the section describing that type of key.

If the CKA_EXTRACTABLE and CKA_EXPORTABLE attribute is FALSE, then the key cannot be wrapped.

Generic Secret Key Objects

Generic secret key objects (object class CKO_SECRET_KEY**,** key type CKK_GENERIC_SECRET) hold generic secret keys. These keys do not support encryption, decryption, signatures or verification (other than HMAC algorithms); however, other keys can be derived from them. The following table defines attributes of generic secret key objects, in addition to the common attributes listed in Table 1, Table 6, Table 13, and Table 23:

Table 26 – Generic Secret Key Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (arbitrary length)
CKA_VALUE_LEN[2,3,6]	CK_ULONG	Length in bytes of key value

RC2 Secret Key Objects

RC2 secret key objects (object class CKO_SECRET_KEY**,** key type CKK_RC2) hold RC2 keys. The following table defines the RC2 secret key object attributes, in addition to the common attributes listed in Table 1, Table 6, Table 13, and Table 23:

Table 14 – RC2 Secret Key Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (1 to 128 bytes)
CKA_VALUE_LEN[2,3,6]	CK_ULONG	Length in bytes of key value

RC4 Secret Key Objects

RC4 secret key objects (object class CKO_SECRET_KEY**,** key type CKK_RC4) hold RC4 keys. The following table defines the RC4 secret key object attributes, in addition to the common attributes listed in Table 1, Table 6, Table 13, and Table 23:

Table 28 – RC4 Secret Key Object

Attribute	Data Type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (1 to 256 bytes)
CKA_VALUE_LEN[2,3,6]	CK_ULONG	Length in bytes of key value

AES Secret Key Objects

AES secret key objects (object class CKO_SECRET_KEY**,** key type CKK_AES) hold AES keys. The following table defines the AES secret key object attributes, in addition to the common attributes listed in Table 1, Table 13, and Table 23:

Table 15 – AES Secret Key Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (16 to 32 bytes)
CKA_VALUE_LEN[2,3,6]	CK_ULONG	Length in bytes of key value

DES Secret Key Objects

DES secret key objects (object class CKO_SECRET_KEY**,** key type CKK_DES) hold single-length DES keys. The following table defines the DES secret key object attributes, in addition to the common attributes listed in Table 1, Table 6, Table 13, and Table 23:

Table 30 – DES Secret Key Object

Attribute	Data Type	Meaning
CKA_VALUE [1,4,6,7]	Byte array	Key value (always 8 bytes long)

DES keys should always have their parity bits properly set as described in FIPS PUB 46-2. However, attempting to create or unwrap a DES key with incorrect parity will not return an error as the key will still function correctly.

DES2 Secret Key Objects

DES2 secret key objects (object class CKO_SECRET_KEY, key type CKK_DES2) hold double-length DES keys. The following table defines the DES2 secret key object attributes, in addition to the common attributes listed in Table 1, Table 6, Table 13, and Table 23:

Table 31 – DES2 Secret Key Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE [1,4,6,7]	Byte array	Key value (always 16 bytes long)

DES2 keys should have their parity bits properly set as described in FIPS PUB 46-2 (that is, each of the DES keys comprising a DES2 key should have its parity bits properly set). However, attempting to create or unwrap a DES2 key with incorrect parity will not return an error as the key will still function correctly.

DES3 Secret Key Objects

DES3 secret key objects (object class CKO_SECRET_KEY**,** key type CKK_DES3) hold triple-length DES keys. The following table defines the DES3 secret key object attributes, in addition to the common attributes listed in Table 1, Table 6, Table 13, and Table 23:

Table 162 – DES3 Secret Key Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE [1,4,6,7]	Byte array	Key value (always 24 bytes long)

DES3 keys should always have their parity bits properly set as described in FIPS PUB 46-2 (that is, each of the DES keys comprising a DES3 key should have its parity bits properly set). However, attempting to create or unwrap a DES3 key with incorrect parity will not return an error as the key will still function correctly.

CAST128 (CAST5) Secret Key Objects

CAST128 (also known as CAST5) secret key objects (object class `CKO_SECRET_KEY`, key type `CKK_CAST128` or `CKK_CAST5`) hold CAST128 keys. The following table defines the CAST128 secret key object attributes, in addition to the common attributes listed in Table 1, Table 6, Table 13, and Table 23:

Table 173 – CAST128 (CAST5) Secret Key Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (1 to 16 bytes)
CKA_VALUE_LEN[2,3,6]	CK_ULONG	Length in bytes of key value

IDEA Secret Key Objects

IDEA secret key objects (object class `CKO_SECRET_KEY`, key type `CKK_IDEA`) hold IDEA keys. The following table defines the IDEA secret key object attributes, in addition to the common attributes listed in Table 1, Table 6, Table 13, and Table 23:

Table 184 – IDEA Secret Key Object

Attribute	Data Type	Meaning
CKA_VALUE [1,4,6,7]	Byte array	Key value (always 16 bytes long)

SEED Secret Key Objects

SEED secret key objects (object class `CKO_SECRET_KEY`, key type `CKK_SEED`) hold SEED keys. The following table defines the SEED secret key object attributes, in addition to the common attributes listed in Table 1, Table 6, Table 13, and Table 23:

Table 35 – SEED Secret Key Object

Attribute	Data type	Meaning
CKA_VALUE [1,4,6,7,10]	Byte array	Key value (always 16 bytes long)

Key Parameter Objects

ProtectToolkit C includes support for key parameter objects (as specified in PKCS#11 2.11 draft 3). These objects are used to store parameters associated with DSA or DH keys. It is possible to generate new objects of this type using the `C_GenerateKey` function.

Key parameter objects (object class `CKO_DOMAIN_PARAMETERS`) hold public key generation parameters. This version of Cryptoki recognizes the following types of key parameters: DSA and Diffie-Hellman. The following table defines the footnotes that apply to each of the following attribute tables:

Table 36 – Common footnotes for key parameter attribute tables

[1] Must be specified when object is created with `C_CreateObject`.
[2] Must *not* be specified when object is created with `C_CreateObject`.
[3] Must be specified when object is generated with `C_GenerateKey` or `C_GenerateKeyPair`.
[4] Must *not* be specified when object is generated with `C_GenerateKey` or `C_GenerateKeyPair`.

The following table defines the attributes common to key attribute objects in addition to the common attributes listed in Table 1 and Table 6:

Table 197 – Common Key Parameter Attributes

Attribute	Data Type	Meaning
CKA_KEY_TYPE[1]	CK_KEY_TYPE	Type of key the parameters can be used to generate.
CKA_LOCAL[2,4]	CK_BBOOL	TRUE only if key parameters were either: • generated locally (that is, on the token) with a C_GenerateKey • created with a C_CopyObjectcall as a copy of key parameters which had its CKA_LOCAL attribute set to TRUE

The rules applying to the CKA_LOCAL mean that this attribute has the value TRUE if and only if the key was originally generated on the token by a C_GenerateKey call.

DSA Public Key Parameter Objects

DSA public key parameter objects (object class CKO_DOMAIN_PARAMETERS**,** key type CKK_DSA) hold DSA public key parameters. The following table defines the DSA public key parameter object attributes, in addition to the common attributes listed in Table 1, Table 6, and Table 35:

Table 38 – DSA Public Key Parameter Object Attributes

Attribute	Data Type	Meaning
CKA_PRIME[1,4]	Big integer	Prime *p* (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME[1,4]	Big integer	Subprime *q* (160 bits)
CKA_BASE[1,4]	Big integer	Base *g*
CKA_PRIME_BITS[2,3]	CK_ULONG	Length of the prime value

The CKA_PRIME**,** CKA_SUBPRIME and CKA_BASE attribute values are collectively the "DSA parameters". See FIPS PUB 186 for more information on DSA key parameters.

Objects of this type may be generated by using the C_GenerateKey with the CKM_DSA_PARAMETER_GEN mechanism.

Diffie-Hellman Public Key Parameter Objects

Diffie-Hellman public key parameter objects (object class CKO_DOMAIN_PARAMETERS**,** key type CKK_DH) hold Diffie-Hellman public key parameters. The following table defines the Diffie-Hellman public key parameter object attributes, in addition to the common attributes listed in Table 1, Table 6 and Table 35:

Table 39 – Diffie-Hellman Public Key Parameter Object Attributes

Attribute	Data Type	Meaning
CKA_PRIME[1,4]	Big integer	Prime *p*
CKA_BASE[1,4]	Big integer	Base *g*
CKA_PRIME_BITS[2,3]	CK_ULONG	Length of the prime value

The **CKA_PRIME** and **CKA_BASE** attribute values are collectively the "Diffie-Hellman parameters". Depending on the token, there may be limits on the length of the key components. See PKCS #3 for more information on Diffie-Hellman key parameters.

Objects of this type may be generated by using the C_GenerateKey with the CKM_DH_PKCS_PARAMETER_GEN mechanism.

Elliptic Curve Public Key Parameter Objects

Elliptic Curve public key parameter objects (object class CKO_DOMAIN_PARAMETERS, key type CKK_EC) hold Elliptic Curve public key parameters.

The following table defines the Elliptic Curve public key parameter object attributes, in addition to the common attributes listed in Tables 7.2, 7.14 and 7.41:

Table 40 – Elliptic Curve Public Key Parameter Object Attributes

Attribute	Data Type	Meaning
CKA_EC_PARAMS[1,3,6]	Byte Array	DER encoding of ANSI X9.62 Parameters value

The CKA_EC_PARAMS attribute values is the "Elliptic Curve parameters". Depending on the token, there may be limits on the length of the key components.

PTK C does not support generation of this type of object.

When objects of this type are stored using the C_CreateObject then the domain parameters are verified. See description of CKM_EC_KEY_PAIR_GEN mechanism for more details on the Parameter value.

Mechanisms

Characteristics of all ProtectToolkit C mechanisms are summarized in the tables that follow. Both PKCS #11 standard mechanisms and SafeNet proprietary mechanisms are included.

Table 41 lists the operations supported by each mechanism.
Table 42 lists the key size range and any parameters defined for each mechanism.

NOTE: Functions in bold in the tables are SafeNet proprietary.

After the tables are notes corresponding to the superscript numbers and, in alphabetical order, a detailed description of each mechanism.

Table 41 – Mechanisms - Operations Supported

Mechanism	Encrypt & Decrypt	Sign & Verify	SR & VR1	Digest	Gen. Key / Key-Pair	Wrap & Un-wrap	Derive	FIPS
CKM_ARIA_CBC	y					y		
CKM_ ARIA _CBC_PAD	y					y		
CKM_ ARIA _ECB	y					y		
CKM_ ARIA _KEY_GEN					y			
CKM_ ARIA _MAC		y						
CKM_ ARIA _MAC_GENERAL		y						
CKM_AES_CBC	y					y		y
CKM_AES_CBC_PAD	y					y		y
CKM_AES_ECB	y					y		y

Mechanism	**Encrypt & Decrypt**	**Sign & Verify**	**SR & VR1**	**Digest**	**Gen. Key / Key-Pair**	**Wrap & Un-wrap**	**Derive**	**FIPS**
CKM_AES_KEY_GEN					y			y
CKM_AES_KEY_WRAP						y		y
CKM_AES_KEY_WRAP_PAD						y		y
CKM_AES_MAC		y						
CKM_AES_MAC_GENERAL		y						
CKM_ARDFP[15]				y				
CKM_CAST128_CBC (CKM_CAST5_CBC)	y					y		
CKM_CAST128_CBC_PAD (CKM_CAST5_CBC_PAD)	y					y		
CKM_CAST128_ECB (CKM_CAST5_ECB)	y					y		
CKM_CAST128_ECB_PAD[4]	y					y		
CKM_CAST128_KEY_GEN (CKM_CAST5_KEY_GEN)					y			
CKM_CAST128_MAC (CKM_CAST5_MAC)		y						
CKM_CAST128_MAC_GENERAL (CKM_CAST5_MAC_GENERAL)		y						
CKM_CONCATENATE_BASE_AND_DATA							y	y
CKM_CONCATENATE_BASE_AND_KEY							y	y
CKM_CONCATENATE_DATA_AND_BASE							Y	y
CKM_DECODE_PKCS_7[4]							Y	y
CKM_DECODE_ X_509[4]							Y	y
CKM_DES_BCF [4, 15]	y					y		
CKM_DES_CBC	y					y		
CKM_DES_CBC_PAD	y					y		
CKM_DES_DERIVE_CBC[4]							y	
CKM_DES_DERIVE_ECB[4]							y	
CKM_DES_ECB	y					y		
CKM_DES_ECB_PAD[4]	y					y		
CKM_DES_KEY_GEN					y			
CKM_DES_MAC		y						
CKM_DES_MAC_GENERAL		y						
CKM_DES_MDC_2_PAD1[4]				y				
CKM_DES_OFB64[4]	Y							
CKM_DES2_KEY_GEN					y			yes
CKM_DES3_BCF [4, 15]	Y					y		y[15]
CKM_DES3_CBC	y					y		y
CKM_DES3_CBC_PAD	y					y		y

Mechanism	**Encrypt & Decrypt**	**Sign & Verify**	**SR & VR1**	**Digest**	**Gen. Key / Key-Pair**	**Wrap & Un-wrap**	**Derive**	**FIPS**
CKM_DES3_DDD_CBC[4]	Y					y		
CKM_DES3_DERIVE_CBC[4]							y	
CKM_DES3_DERIVE_ECB[4]							y	
CKM_DES3_ECB	Y					y		y
CKM_DES3_ECB_PAD[4]	Y					y		y
CKM_DES3_KEY_GEN					y			y
CKM_DES3_MAC		y						y
CKM_DES3_MAC_GENERAL		y						y
CKM_DES3_OFB64[4]	**y**							y
CKM_DES3_RETAIL_CFB_MAC[4]		Y						y
CKM_DES3_X919_MAC_GENERAL[4]		Y						y
CKM_DES3_X919_MAC[4]		Y						y
CKM_DH_PKCS_DERIVE							y	yes
CKM_DH_PKCS_KEY_PAIR_GEN					y			y
CKM_DH_PKCS_PARAMETER_GEN					y			y
CKM_DSA		y[2]						y
CKM_DSA_KEY_PAIR_GEN					y			y
CKM_DSA_PARAMETER_GEN					y			y
CKM_DSA_SHA1		y						no[6]
CKM_DSA_SHA1_PKCS[4]		y						no[6]
CKM_DSA_SHA224_PKCS[4]		y						yes
CKM_DSA_SHA256_PKCS[4]		y						yes
CKM_EC_KEY_PAIR_GEN					Y			y
CKM_ECDH1_DERIVE							Y	Yes
CKM_ECDSA		y						Y
CKM_ECDSA_SHA1		y						no[6]
CKM_ECDSA_SHA224		y						Y
CKM_ECDSA_SHA256		y						Y
CKM_ECDSA_SHA386		y						Y
CKM_ECDSA_SHA512		y						Y
CKM_ECIES[4]	y[2]							
CKM_ENCODE_ATTRIBUTES[4]						y		Y
CKM_ENCODE_PKCS_10[4]							y	y
CKM_ENCODE_PUBLIC_KEY[4]						y		y
CKM_ENCODE_X_509_LOCAL_CERT[4]							y	y
CKM_ENCODE_X_509[4]							y	y

Mechanism	**Encrypt & Decrypt**	**Sign & Verify**	**SR & VR1**	**Digest**	**Gen. Key / Key-Pair**	**Wrap & Un-wrap**	**Derive**	**FIPS**
CKM_EXTRACT_KEY_FROM_KEY							y	no
CKM_FM_DOWNLOAD[4, 8, 9]		y[6]						y
CKM_FM_DOWNLOAD_2[4, 8, 9]		y[6]						y
CKM_GENERIC_SECRET_KEY_GEN					y			y
CKM_IDEA_CBC	y					y		
CKM_IDEA_CBC_PAD	y					y		
CKM_IDEA_ECB	y					y		
CKM_IDEA_ECB_PAD[4]	y					y		
CKM_IDEA_KEY_GEN					y			
CKM_IDEA_MAC		y						
CKM_IDEA_MAC_GENERAL		y						
CKM_KEY_TRANSLATION[4, 7]						y		
CKM_KE.Y_WRAP_SET_OAEP						y		y
CKM_MD2				y				
CKM_MD2_HMAC		y						
CKM_MD2_HMAC_GENERAL		y						
CKM_MD2_KEY_DERIVATION							y	
CKM_MD2_RSA_PKCS		y						
CKM_MD5				y				
CKM_MD5_HMAC		y						
CKM_MD5_HMAC_GENERAL		y						
CKM_MD5_KEY_DERIVATION							y	
CKM_MD5_RSA_PKCS		y						
CKM_NVB[15]				y				
CKM_OS_UPGRADE[4, 8]		y[6]						y
CKM_OS_UPGRADE_2[4, 8]		y[6]						y
CKM_PBA_SHA1_WITH_SHA1_HMAC					Y			
CKM_PBE_MD2_DES_CBC					y			
CKM_PBE_MD5_CAST128_CBC (CKM_PBE_MD5_CAST5_CBC)					y			
CKM_PBE_MD5_DES_CBC					y			
CKM_PBE_SHA1_CAST128_CBC (CKM_PBE_SHA1_CAST5_CBC)					y			
CKM_PBE_SHA1_DES2_EDE_CBC					y			
CKM_PBE_SHA1_DES3_EDE_CBC					y			
CKM_PBE_SHA1_RC2_128_CBC					Y			
CKM_PBE_SHA1_RC2_40_CBC					Y			
CKM_PBE_SHA1_RC4_128					Y			

Mechanism	**Encrypt & Decrypt**	**Sign & Verify**	**SR & VR1**	**Digest**	**Gen. Key / Key-Pair**	**Wrap & Un-wrap**	**Derive**	**FIPS**
CKM_PBE_SHA1_RC4_40					Y			
CKM_PKCS12_PBE_EXPORT						Y		
CKM_PKCS12_PBE_IMPORT						Y		
CKM_PP_LOAD_SECRET[4]					y			y
CKM_RC2_CBC	y					y		
CKM_RC2_CBC_PAD	y					y		
CKM_RC2_ECB	y					y		
CKM_RC2_ECB_PAD[4]	y					Y		
CKM_RC2_KEY_GEN					y			
CKM_RC2_MAC		y						
CKM_RC2_MAC_GENERAL		y						
CKM_RC4	y							
CKM_RC4_KEY_GEN					y			
CKM_REPLICATE_TOKEN_RSA_AES						y		y
CKM_RIPEMD128				y				
CKM_RIPEMD128_HMAC		y						
CKM_RIPEMD128_HMAC_GENERAL		y						
CKM_RIPEMD128_RSA_PKCS		y						
CKM_RIPEMD160				y				
CKM_RIPEMD160_HMAC		y						
CKM_RIPEMD160_HMAC_GENERAL		y						
CKM_RIPEMD160_RSA_PKCS		y						
CKM_RSA_9796		y[2]	y					
CKM_RSA_FIPS_186_4_PRIME_KEY_PAIR_GEN					y			y
CKM_RSA_PKCS	y[2]	y[2]	y			y		y
CKM_RSA_PKCS_KEY_PAIR_GEN					y			Yes*
CKM_RSA_PKCS_OAEP	y[2]					y		y
CKM_RSA_X_509	y[2]	y[2]	y			y		
CKM_RSA_X9_31_KEY_PAIR_GEN					y			y
CKM_SECRET_RECOVER_WITH_ATTRIBUTES[4]							y	y
CKM_SECRET_SHARE_WITH_ATTRIBUTES[4]							y	y
CKM_SEED_CBC[4]	y					Y		
CKM_SEED_CBC_PAD[4]	y					Y		
CKM_SEED_ECB[4]	y					Y		
CKM_SEED_ECB_PAD[4]	y					Y		

Mechanism	**Encrypt & Decrypt**	**Sign & Verify**	**SR & VR1**	**Digest**	**Gen. Key / Key-Pair**	**Wrap & Un-wrap**	**Derive**	**FIPS**
CKM_SEED_KEY_GEN[4]					y			
CKM_SEED_MAC[4]		y						
CKM_SEED_MAC_GENERAL[4]		y						
CKM_SET_ATTRIBUTES [4]								
CKM_SHA_1				y				Yes
CKM_SHA_1_HMAC		y						Y
CKM_SHA_1_HMAC_GENERAL		y						Y
CKM_SHA1_KEY_DERIVATION							y	
CKM_SHA1_RSA_PKCS		y						no[6]
CKM_SHA1_RSA_PKCS_TIMESTAMP[4]		y [10]						
CKM_SHA224				y				Y
CKM_SHA224_HMAC		y						Y
CKM_SHA224_HMAC_GENERAL		y						Y
CKM_SHA224_KEY_DERIVATION							y	
CKM_SHA224_RSA_PKCS		y						Y
CKM_SHA256				y				Y
CKM_SHA256_HMAC		y						Y
CKM_SHA256_HMAC_GENERAL		y						Y
CKM_SHA256_KEY_DERIVATION							y	
CKM_SHA256_RSA_PKCS		y						Y
CKM_SHA384				y				Y
CKM_SHA384_HMAC		y						Y
CKM_SHA384_HMAC_GENERAL		y						Y
CKM_SHA384_KEY_DERIVATION							y	
CKM_SHA384_RSA_PKCS		y						Y
CKM_SHA512				y				Y
CKM_SHA512_HMAC		y						Y
CKM_SHA512_HMAC_GENERAL		y						Y
CKM_SHA512_KEY_DERIVATION							y	No
CKM_SHA512_RSA_PKCS		y						Y
CKM_SSL3_KEY_AND_MAC_DERIVE							y	No
CKM_SSL3_MASTER_KEY_DERIVE							y	
CKM_SSL3_MD5_MAC		y						
CKM_SSL3_PRE_MASTER_KEY_GEN					y			Y
CKM_SSL3_SHA1_MAC		y						
CKM_VISA_CVV[4]		y						

Mechanism	Encrypt & Decrypt	Sign & Verify	SR & VR1	Digest	Gen. Key / Key-Pair	Wrap & Un-wrap	Derive	FIPS
CKM_WRAPKEY_DES3_CBC[4]						y		Y
CKM_WRAPKEY_DES3_ECB[4]						y		Y
CKM_WRAPKEY_AES_CBC[4]						y		Y
CKM_WRAPKEYBLOB_DES3_CBC[4]						y		Y
CKM_WRAPKEYBLOB_AES_CBC[4]						y		Y
CKM_X9_42_DH_KEY_PAIR_GEN[4]					y			Y
CKM_X9_42_DH_PARAMETER_GEN[4]					y			Y
CKM_X9_42_DH_DERIVE[4]							Y	y
CKM_XOR_BASE_AND_DATA							y	y
CKM_XOR_BASE_AND_KEY[4]							y	**Y**
CKM_ZKA_MDC_2_KEY_DERIVATION[4]							y	

Note: In the above table, “y” means “yes” a condition is true for the given mechanism, while an empty cell means that the condition is not true.

Some mechanisms are explicitly marked ”no” in the FIPS column – this is to indicate mechanisms that formerly were acceptable for FIPS, but which are no longer acceptable in FIPS mode because the standard (or its interpretation by the compliance-testing/validation community) has evolved. Some of the mechanisms marked “no” can still be used for verify operations, in which case they are marked with the 6 subscript note (ie, no[6]) . See the following table for a complete list of the subscripts and their meanings.

Table 42 – Mechanisms - Key Size Range and Parameters

Mechanism	Min	Max -1 == infinite	Parameter
CKM_ ARIA _CBC	16	32	byte[16]
CKM_ ARIA _CBC_PAD	16	32	byte[16]
CKM_ ARIA _ECB	16	32	Null
CKM_ ARIA _KEY_GEN	16	32	Null
CKM_ ARIA _MAC	16	32	Null
CKM_ ARIA _MAC_GENERAL	16	32	CK_MAC_GENERAL_PARAMS
CKM_AES_KEY_WRAP	16	32	Byte[8] (optional)
CKM_AES_KEY_WRAP_PAD	16	32	Byte[8] (optional)
CKM_AES_CBC	16	32	byte[16]
CKM_AES_CBC_PAD	16	32	byte[16]
CKM_AES_ECB	16	32	Null
CKM_AES_KEY_GEN	16	32	Null
CKM_AES_MAC	16	32	Null
CKM_AES_MAC_GENERAL	16	32	CK_MAC_GENERAL_PARAMS
CKM_CAST128_CBC (CKM_CAST5_CBC)	1	16	byte[8]
CKM_CAST128_CBC_PAD (CKM_CAST5_CBC_PAD)	1	16	byte[8]
CKM_CAST128_ECB (CKM_CAST5_ECB)	1	16	Null
CKM_CAST128_ECB_PAD[4]	**1**	**16**	**Null**
CKM_CAST128_KEY_GEN (CKM_CAST5_KEY_GEN)	1	16	Null
CKM_CAST128_MAC (CKM_CAST5_MAC)	1	16	Null
CKM_CAST128_MAC_GENERAL (CKM_CAST5_MAC_GENERAL)	1	16	CK_MAC_GENERAL_PARAMS
CKM_CONCATENATE_BASE_AND_DATA	0	-1	CK_KEY_DERIVATION_STRING_DATA
CKM_CONCATENATE_BASE_AND_KEY	0	-1	CK_OBJECT_HANDLE
CKM_CONCATENATE_DATA_AND_BASE	0	-1	CK_KEY_DERIVATION_STRING_DATA
CKM_DECODE_PKCS_7[4]	**0**	**0**	**Null**
CKM_DECODE_ X_509[4]	**0**	**0**	**Null**
CKM_DES_BCF [4, 15]	8	8	byte[8]
CKM_DES_CBC	8	8	byte[8]
CKM_DES_CBC_PAD	8	8	byte[8]
CKM_DES_DERIVE_CBC[4]	**8**	**8**	**CK_DES_CBC_PARAMS**
CKM_DES_DERIVE_ECB[4]	**8**	**8**	**byte[n*8]**
CKM_DES_ECB	8	8	Null
CKM_DES_ECB_PAD[4]	**8**	**8**	**Null**
CKM_DES_KEY_GEN	8	8	Null

Mechanism	Min	Max -1 == infinite	Parameter
CKM_DES_MAC	8	8	CK_MAC_GENERAL_PARAMS
CKM_DES_MAC_GENERAL	8	8	CK_MAC_GENERAL_PARAMS
CKM_DES_MDC_2_PAD1[4]	**0**	**0**	**Null**
CKM_DES_OFB64[4]	**8**	**8**	**byte[8]**
CKM_DES2_KEY_GEN	16	16	Null
CKM_DES3_BCF [4, 15]	16	24	byte[8]
CKM_DES3_CBC	16	24	byte[8]
CKM_DES3_CBC_PAD	16	24	byte[8]
CKM_DES3_DDD_CBC[4]	**16**	**24**	**byte[8]**
CKM_DES3_DERIVE_CBC[4]	**16**	**24**	**CK_DES2_CBC_PARAMS CK_DES3_CBC_PARAMS**
CKM_DES3_DERIVE_ECB[4]	**0**	**0**	**byte[n*8]**
CKM_DES3_ECB	16	24	Null
CKM_DES3_ECB_PAD[4]	**16**	**24**	**Null**
CKM_DES3_KEY_GEN	24	24	Null
CKM_DES3_MAC	16	24	Null
CKM_DES3_MAC_GENERAL	16	24	CK_MAC_GENERAL_PARAMS
CKM_DES3_OFB64[4]	**16**	**24**	**byte[8]**
CKM_DES3_RETAIL_CFB_MAC[4]	**16**	**24**	**byte[8] (IV)**
CKM_DES3_X919_MAC_GENERAL[4]	**16**	**24**	**byte[8]**
CKM_DES3_X919_MAC[4]	**16**	**24**	**CK_MAC_GENERAL_PARAMS**
CKM_DH_PKCS_DERIVE [12]	512/ 1024	4096	byte[] (Big Integer)
CKM_DH_PKCS_KEY_PAIR_GEN [12]	512/ 1024	4096	Null
CKM_DH_PKCS_PARAMETER_GEN [12]	512/ 1024	4096	Null
CKM_DSA [12]	512/ 2048	4096	Null
CKM_DSA_KEY_PAIR_GEN [12]	512/ 2048	4096	Null
CKM_DSA_PARAMETER_GEN [12]	512/ 2048	4096	Null
CKM_DSA_SHA1	512/ 2048	4096	Null
CKM_DSA_SHA1_PKCS[4]	**512/ 2048**	**4096**	**Null**
CKM_DSA_SHA224_PKCS[4]	**1024 /2048**	**4096**	**Null**
CKM_DSA_SHA256_PKCS[4]	**1024 /204**	**4096**	**Null**

Mechanism	Min	Max -1 == infinite	Parameter
	8		
CKM_EC_KEY_PAIR_GEN	160/ 224	571	Null
CKM_ECDH1_DERIVE	160	571	CK_ECDH1_DERIVE_PARAMS
CKM_ECDSA	160/ 224	571	Null
CKM_ECDSA_SHA1	160/ 224	571	Null
CKM_ECDSA_SHA224	160/ 224	571	Null
CKM_ECDSA_SHA256	160/ 224	571	Null
CKM_ECDSA_SHA384	160/ 224	571	Null
CKM_ECDSA_SHA512	160/ 224	571	Null
CKM_ECIES [4]	160	571	CK_ECIES_PARAMS
CKM_ENCODE_ATTRIBUTES[4]	**0**	**0**	**Null**
CKM_ENCODE_PKCS_10[4]	**0**	**0**	**Null**
CKM_ENCODE_PUBLIC_KEY[4]	**0**	**0**	**Null**
CKM_ENCODE_X_509_LOCAL_CERT[4]	**0**	**0**	**Null**
CKM_ENCODE_X_509[4]	**0**	**0**	**CK_MECH_TYPE_AND_OBJECT**
CKM_EXTRACT_KEY_FROM_KEY	0	0	CK_EXTRACT_PARAMS
CKM_FM_DOWNLOAD[4, 8, 9 , 11]	512/ 1024	4096	**Null**
CKM_FM_DOWNLOAD_2[4, 8, 9 , 11]	2048 /102 4	4096	**Null**
CKM_GENERIC_SECRET_KEY_GEN	0	-1	Null
CKM_IDEA_CBC	16	16	byte[8]
CKM_IDEA_CBC_PAD	16	16	byte[8]
CKM_IDEA_ECB	16	16	Null
CKM_IDEA_ECB_PAD[4]	**16**	**16**	**Null**
CKM_IDEA_KEY_GEN	16	16	Null
CKM_IDEA_MAC	16	16	Null
CKM_IDEA_MAC_GENERAL	16	16	CK_MAC_GENERAL_PARAMS
CKM_KEY_TRANSLATION[4, 7]	512	4096	Null
CKM_KEY_WRAP_SET_OAEP [11]	512/ 1024	4096	CK_KEY_WRAP_SET_OAEP_PARAMS
CKM_MD2	0	0	Null
CKM_MD2_HMAC	0	0	Null

Mechanism	Min	Max -1 == infinite	Parameter
CKM_MD2_HMAC_GENERAL	0	0	CK_MAC_GENERAL_PARAMS
CKM_MD2_KEY_DERIVATION	0	0	Null
CKM_MD2_RSA_PKCS	512	4096	Null
CKM_MD5	0	0	Null
CKM_MD5_HMAC	0	0	Null
CKM_MD5_HMAC_GENERAL	0	0	CK_MAC_GENERAL_PARAMS
CKM_MD5_KEY_DERIVATION	0	0	Null
CKM_MD5_RSA_PKCS	512	4096	Null
CKM_NVB[15]	0	0	Null
CKM_OS_UPGRADE[4, 8, 9]	1024	4096	Null
CKM_OS_UPGRADE_2[4, 8, 9]	1024	4096	Null
CKM_PBA_SHA1_WITH_SHA1_HMAC	20	20	CK_PBE_PARAMS
CKM_PBE_MD2_DES_CBC	8	8	CK_PBE_PARAMS
CKM_PBE_MD5_CAST128_CBC (CKM_PBE_MD5_CAST5_CBC)	16	16	CK_PBE_PARAMS
CKM_PBE_MD5_DES_CBC	8	8	CK_PBE_PARAMS
CKM_PBE_SHA1_CAST128_CBC (CKM_PBE_SHA1_CAST5_CBC)	16	16	CK_PBE_PARAMS
CKM_PBE_SHA1_DES2_EDE_CBC	16	16	CK_PBE_PARAMS
CKM_PBE_SHA1_DES3_EDE_CBC	24	24	CK_PBE_PARAMS
CKM_PBE_SHA1_RC2_128_CBC	16	16	CK_PBE_PARAMS
CKM_PBE_SHA1_RC2_40_CBC	5	5	CK_PBE_PARAMS
CKM_PBE_SHA1_RC4_128	16	16	CK_PBE_PARAMS
CKM_PBE_SHA1_RC4_40	5	5	CK_PBE_PARAMS
CKM_PKCS12_PBE_EXPORT [13]	1	-1	CKM_PKCS12_PBE_EXPORT_PARAMS
CKM_PKCS12_PBE_IMPORT [14]	1	-1	CKM_PKCS12_PBE_IMPORT_PARAMS
CKM_PP_LOAD_SECRET[4]	1	-1	CK_PP_LOAD_SECRET_PARAMS
CKM_RC2_CBC	1	128	CK_RC2_CBC_PARAMS
CKM_RC2_CBC_PAD	1	128	CK_RC2_CBC_PARAMS
CKM_RC2_ECB	1	128	CK_RC2_PARAMS
CKM_RC2_ECB_PAD[4]	1	128	CK_RC2_PARAMS
CKM_RC2_KEY_GEN	1	128	Null
CKM_RC2_MAC	1	128	CK_RC2_PARAMS
CKM_RC2_MAC_GENERAL	1	128	CK_RC2_MAC_GENERAL_PARAMS
CKM_RC4	0	256	Null
CKM_RC4_KEY_GEN	0	256	Null
CKM_REPLICATE_TOKEN_RSA_AES	2048	2048	CK_REPLICATE_TOKEN_PARAMS
CKM_RIPEMD128	0	0	Null
CKM_RIPEMD128_HMAC	0	0	Null

Mechanism	Min	Max -1 == infinite	Parameter
CKM_RIPEMD128_HMAC_GENERAL	0	0	CK_MAC_GENERAL_PARAMS
CKM_RIPEMD128_RSA_PKCS	512	4096	Null
CKM_RIPEMD160	0	0	Null
CKM_RIPEMD160_HMAC	0	0	Null
CKM_RIPEMD160_HMAC_GENERAL	0	0	CK_MAC_GENARAL_PARAMS
CKM_RIPEMD160_RSA_PKCS	512	4096	Null
CKM_RSA_9796	512	4096	Null
CKM_RSA_PKCS [11]	512/ 1024	4096	Null
CKM_RSA_PKCS_KEY_PAIR_GEN [11]	512/ 1024	4096	Null
CKM_RSA_PKCS_OAEP [11]	512/ 1024	4096	CK_RSA_PKCS_OAEP_PARAMS
CKM_RSA_FIPS_186_4_PRIME_KEY_PAIR_GEN [11]	2048	3072	CK_ULONG (optional)
CKM_RSA_X_509 [11]	512/ 1024	4096	Null
CKM_RSA_X9_31_KEY_PAIR_GEN [11]	1024	4096	Null
CKM_SECRET_RECOVER_WITH_ATTRIBUTES[4]	0	-1	CK_SECRET_SHARE_PARAMS
CKM_SECRET_SHARE_WITH_ATTRIBUTES[4]	0	-1	Null
CKM_SEED_CBC[4]	16	16	byte[16]
CKM_SEED_CBC_PAD[4]	16	16	byte[16]
CKM_SEED_ECB[4]	16	16	Null
CKM_SEED_ECB_PAD[4]	16	16	Null
CKM_SEED_KEY_GEN[4]	16	16	Null
CKM_SEED_MAC[4]	16	16	Null
CKM_SEED_MAC_GENERAL[4]	16	16	CK_MAC_GENERAL_PARAMS
CKM_SET_ATTRIBUTES [4]	0	0	Null
CKM_SHA_1	0	0	Null
CKM_SHA_1_HMAC	0	-1	Null
CKM_SHA_1_HMAC_GENERAL	0	-1	CK_MAC_GENERAL_PARAMS
CKM_SHA1_KEY_DERIVATION	0	0	Null
CKM_SHA1_RSA_PKCS [11]	512	4096	Null
CKM_SHA1_RSA_PKCS_TIMESTAMP [4, 11]	512	4096	CK_TIMESTAMP_PARAMS
CKM_SHA2246	0	0	Null
CKM_SHA225_HMAC	0	-1	Null
CKM_SHA224_HMAC_GENERAL	0	-1	CK_MAC_GENERAL_PARAMS
CKM_SHA224_KEY_DERIVATION	0	0	Null
CKM_SHA224_RSA_PKCS [11]	512/ 1024	4096	Null

Mechanism	Min	Max -1 == infinite	Parameter
CKM_SHA256	0	0	Null
CKM_SHA256_HMAC	0	-1	Null
CKM_SHA256_HMAC_GENERAL	0	-1	CK_MAC_GENERAL_PARAMS
CKM_SHA256_KEY_DERIVATION	0	0	Null
CKM_SHA256_RSA_PKCS [11]	512/ 1024	4096	Null
CKM_SHA384	0	0	Null
CKM_SHA384_HMAC	0	-1	Null
CKM_SHA384_HMAC_GENERAL	0	-1	CK_MAC_GENERAL_PARAMS
CKM_SHA384_KEY_DERIVATION	0	0	Null
CKM_SHA384_RSA_PKCS [11]	512/ 1024	4096	Null
CKM_SHA512	0	0	Null
CKM_SHA512_HMAC	0	-1	Null
CKM_SHA512_HMAC_GENERAL	0	-1	CK_MAC_GENERAL_PARAMS
CKM_SHA512_KEY_DERIVATION	0	0	Null
CKM_SHA512_RSA_PKCS [11]	512/ 1024	4096	Null
CKM_SSL3_KEY_AND_MAC_DERIVE	48	48	CK_SSL3_KEY_MAT_PARAMS
CKM_SSL3_MASTER_KEY_DERIVE	48	48	CK_SSL3_MASTER_KEY_DERIVE_PARAMS
CKM_SSL3_MD5_MAC	0	-1	CK_MAC_GENERAL_PARAMS
CKM_SSL3_PRE_MASTER_KEY_GEN	48	48	CK_VERSION
CKM_SSL3_SHA1_MAC	0	-1	CK_MAC_GENERAL_PARAMS
CKM_VISA_CVV [4]	16	16	Null
CKM_WRAPKEY_DES3_CBC [4]	16	24	Null
CKM_WRAPKEY_DES3_ECB [4]	16	24	Null
CKM_WRAPKEY_AES_CBC [4]	16	32	Null
CKM_WRAPKEYBLOB_DES3_CBC [4]	16	32	Null
CKM_WRAPKEYBLOB_AES_CBC [4]	16	32	Null
CKM_X9_42_DH_KEY_PAIR_GEN [4]	1024	4096	null
CKM_X9_42_DH_PARAMETER_GEN [4]	1024	4096	null
CKM_X9_42_DH_DERIVE [4]	1024	4096	CK_X9_42_DH1_DERIVE_PARAMS
CKM_XOR_BASE_AND_DATA	0	-1	CK_KEY_DERIVATION_STRING_DATA
CKM_XOR_BASE_AND_KEY [4]	0	-1	CK_OBJECT_HANDLE
CKM_ZKA_MDC_2_KEY_DERIVATION [4]	0	0	byte[arbitrary]

Note – key size limitations specified above may also be further limited depending on the specific operation being performed e.g. CKM_DES3_CBC mechanism specified 16 byte key as a lower limit but in FIPS mode such keys are only allowed for legacy decryption operations and not new encryptions. For more details see the relevant section.

[1] SR = SignRecover, VR = VerifyRecover
[2] Single part operation only
[3] Mechanism can be used only for wrapping, not unwrapping
[4] Mechanism SafeNet proprietary
[5] Sign Only
[6] Verify Only
[7] Only available when CKF_ENTRUST_READY set in Security Policy Register.
[8] Only available on Administration token
[9] Only available on FM Enabled FW
[10] Sign only
[11] All RSA operations performed under FIPS mode are carried out only if the specified key has a modulus of 2048 bits or greater. Any attempt to create an RSA key smaller than 2048 bits while running in FIPS mode results in a CKR_KEY_SIZE_RANGE or CKA_TEMPLATE_INCONSISTENT error.
[12] All DSA and DH operations performed under FIPS mode are carried out only if the specified key has a modulus of 2048 bits or greater. Any attempt to create a DSA or DH key smaller than 2048 bits while running in FIPS mode results in a CKR_KEY_SIZE_RANGE or CKA_TEMPLATE_INCONSISTENT error.
[13] Wrap
[14] Unwrap
[15] Available in SW Emulation only

CKM_AES_CBC

AES-CBC, denoted CKM_AES_CBC, is a mechanism for single and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on NIST's Advanced Encryption Standard and cipher-block chaining mode. It has a parameter, a 16-byte initialization vector.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the CKA_VALUE attribute of the key that is wrapped; padded on the trailing end with up to block size, minus one, null bytes so that the resulting length is a multiple of the block size. The output data is the same length as the padded input data. It does not wrap the key type, key length, or any other information about the key. The application must convey these separately.

For unwrapping, the mechanism decrypts the wrapped key and truncates the result according to the CKA_KEY_TYPE attribute of the template and, if it has one and the key type supports it, the CKA_VALUE_LEN attribute of the template. The mechanism contributes the result as the CKA_VALUE attribute of the new key. Other attributes required by the key type must be specified in the template.

Constraints on key types and the length of data are summarized in the following table:

Table 43 – AES-CBC: Key and Data Length

Function	Key Type	Input length	Output length	Comments
C_Encrypt	AES	Multiple of block size	same as input length	no final part
C_Decrypt	AES	Multiple of block size	same as input length	no final part
C_WrapKey	AES	Any	input length rounded up to multiple of the block size	
C_UnwrapKey	AES	Multiple of block size	determined by type of key being unwrapped or CKA_VALUE_LEN	

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of AES key sizes, in bytes.

CKM_AES_CBC_PAD

AES-CBC with PKCS padding, denoted CKM_AES_CBC_PAD, is a mechanism for single and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on NIST's Advanced Encryption Standard; cipher-block chaining mode; and the block cipher padding method detailed in PKCS #7. It has a parameter, a 16-byte initialization vector.

The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the cipher text value. No value should be specified for the CKA_VALUE_LEN attribute when unwrapping keys with this mechanism.

In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, and DSA private keys. The entries in Table 40 for data length constraints when wrapping and unwrapping keys do not apply to wrapping and unwrapping private keys.

Constraints on key types and the length of data are summarized in the following table: For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of AES key sizes, in bytes.

Table 44 – AES-CBC with PKCS Padding: Key and Data Length

Function	Key Type	Input Length	Output Length
C_Encrypt	AES	Any	Input length rounded up to multiple of the block size
C_Decrypt	AES	Multiple of block size	Between 1 and block size bytes shorter than input length
C_WrapKey	AES	Any	Input length rounded up to multiple of the block size
C_UnwrapKey	AES	Multiple of block size	Between 1 and block length bytes shorter than input length

CKM_AES_ECB

AES-ECB, denoted `CKM_AES_ECB`, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on NIST Advanced Encryption Standard and electronic codebook mode. It does not have a parameter.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the `CKA_VALUE` attribute of the key that is wrapped; padded on the trailing end with up to block size, minus one, null bytes so that the resulting length is a multiple of the block size. The output data is the same length as the padded input data. It does not wrap the key type, key length, or any other information about the key. The application must convey these separately.

For unwrapping, the mechanism decrypts the wrapped key and truncates the result according to the `CKA_KEY_TYPE` attribute of the template and, if it has one and the key type supports it, the `CKA_VALUE_LEN` attribute of the template. The mechanism contributes the result as the `CKA_VALUE`attribute of the new key. Other attributes required by the key type must be specified in the template.

Constraints on key types and the length of data are summarized in the following table:

Table 45 – AES-ECB: Key and Data Length

Function	Key Type	Input Length	Output Length	Comments
C_Encrypt	AES	Multiple of block size	Same as input length	No final part
C_Decrypt	AES	Multiple of block size	Same as input length	No final part
C_WrapKey	AES	Any	Input length rounded up to multiple of block size	
C_UnwrapKey	AES	Multiple of block size	Determined by type of key being unwrapped or CKA_VALUE_LEN	

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of AES key sizes, in bytes.

CKM_AES_KEY_GEN

The AES key generation mechanism, denoted CKM_AES_KEY_GEN, is a key generation mechanism for NIST's Advanced Encryption Standard. It does not have a parameter.

The mechanism generates AES keys with a particular length in bytes, as specified in the CKA_VALUE_LEN attribute of the template for the key.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE and CKA_VALUE attributes to the new key. Other attributes supported by the AES key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key or else are assigned default initial values.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of AES key sizes, in bytes. Key sizes from 8 to 256 bytes are supported. The algorithm block size is 16 bytes.

CKM_AES_MAC

AES-MAC, denoted by CKM_AES_MAC, is a special case of the general-length AES-MAC mechanism (see section above). AES-MAC always produces and verifies MACs that are half the block size in length. It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 20 – AES-MAC: Key and Data Length

Function	Key Type	Data Length	Signature Length
C_Sign	AES	Any	½ block size (8 bytes)
C_Verify	AES	Any	½ block size (8 bytes)

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of AES key sizes, in bytes.

CKM_AES_MAC_GENERAL

General-length AES-MAC, denoted CKM_AES_MAC_GENERAL, is a mechanism for single- and multiple-part signatures and verification, based on NIST Advanced Encryption Standard.

It has a parameter, a CK_MAC_GENERAL_PARAMS structure, which specifies the output length desired from the mechanism.

The output bytes from this mechanism are taken from the start of the final AES cipher block produced in the MACing process.

Constraints on key types and the length of data are summarized in the following table:

Table 21 – General-length AES-MAC: Key and Data Length

Function	Key Type	Data Length	Signature Length
C_Sign	AES	Any	0-block size, as specified in parameters
C_Verify	AES	Any	0-block size, as specified in parameters

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of AES key sizes, in bytes.

CKM_CAST128_ECB_PAD

This is a padding mechanism. Other padding mechanisms implemented are: `CKM_RC2_ECB_PAD`, `CKM_DES_ECB_PAD`, `CKM_DES3_ECB_PAD` and `CKM_IDEA_ECB_PAD`.

These block cipher mechanisms are all based on the corresponding Electronic Code Book (ECB) algorithms, implied by their name, but with the addition of the block-cipher padding method detailed in PKCS#7.

These mechanisms are supplied for compatibility only and their use in new applications is not recommended.

PKCS#11 Version 2.1 specifies mechanisms for Chain Block Cipher algorithms with and without padding and ECB algorithms without padding, but not ECB with padding. These mechanisms fill this gap. The mechanisms may be used for general data encryption and decryption and also for key wrapping and unwrapping (provided all the access conditions of the relevant keys are satisfied).

CKM_DECODE_PKCS_7

This mechanism is used with the `C_DeriveKey` function to derive a set of X.509 Certificate objects and X.509 CRL objects from a PKCS#7 object. The base key object handle is a `CKO_DATA` object (the PKCS#7 encoding) which has a `CKA_OBJECT_ID` attribute indicating the type of the object as being a PKCS#7 encoding. This mechanism does not take any parameters.

One of the functions of PKCS7 is a mechanism for distributing certificates and CRLs in a single encoded package. In this case the PKCS7 message content is usually empty. This mechanism is provided to split certificates and CRLs from such a PKCS7 encoding so that those certificates and CRLs may be further processed.

This mechanism will decode a PKCS7 encoding and create PKCS#11 objects for all certificates (object class `CKO_CERTIFICATE`) and CRLs (object class `CKO_CRL`) that it finds in the encoding. The signature on the PKCS7 content is not verified. The parameter containing the newly derived key is the last Certificate or CRL that is extracted from the PKCS7 encoding. The attribute template is applied to all objects extracted from the encoding.

CKM_DECODE_X_509

This mechanism is used with the `C_DeriveKey` function to derive a public key object from an X.509 certificate or a PKCS#10 certification request. This mechanism does not perform a certificate validation.

The base key object handle should refer to the X.509 certificate or PKCS#10 certificate request. This mechanism has no parameter.

CKM_DES_DERIVE_CBC

The CKM_DES_DERIVE_CBC and CKM_DES3_DERIVE_CBC mechanisms are used with the `C_DeriveKey` function to derive a secret key by performing a CBC (no padding) encryption. They create a new secret key whose value is generated by encrypting the provided data with the provided Single, Double or Triple length DES key.

Three new mechanism Parameter structures are created, CK_DES_CBC_PARAMS, CK_DES2_CBC_PARAMS and CK_DES3_CBC_PARAMS, for use by these mechanisms. These structures consists of 2-byte arrays, the first array contains the IV (must be 8 bytes) and the second array contains the data to be encrypted, being 8, 16 or 24 bytes in length, for each PARAMS structure respectively.

These mechanisms require the pParameter in the CK_MECHANISM structure to be a pointer to one of the above new Parameter structures and the parameterLen to be the size of the provided Parameter structure.

If the length of data to be encrypted by the CBC mechanism does not fit into one of the above PARAMS structures, the developer must produce their own byte array with the following layout. The first 8 bytes must be the IV, then the data to be encrypted. To use this array, the pParameter in the CK_MECHANISM structure must be a pointer to this array and the parameterLen is the length of the IV (must be 8 bytes) plus the length of the provided data, which must be a multiple of 8 bytes.

The following rules apply to the provided attribute template:

- If no length or key type is provided in the template, then the key produced by these mechanisms is a generic secret key. Its length is equal to the length of the provided data.
- If no key type is provided in the template, but a length is, then the key produced by these mechanisms is a generic secret key of the specified length, extracted from the left bytes of the cipher text.
- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by these mechanisms is of the type specified in the template. If it doesn't, an error is returned.
- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by these mechanisms is of the specified type and length, extracted from the left bytes of the cipher text.

If a DES key is derived with these mechanisms, the parity bits of the key are set properly. If the requested type of key requires more bytes than the length of the provided data, an error is generated.

These mechanisms have the following rules about key sensitivity and extractability:

- If the base key has its CKA_SENSITIVE attribute set to TRUE, so does the derived key. If not, then the derived key's CKA_SENSITIVE attribute is set either from the supplied template or else it defaults to TRUE.
- Similarly, the derived key's CKA_EXTRACTABLE attribute is set either from the supplied template or else it defaults to the value of the CKA_EXTRACTABLE of the base key.
- The derived key's CKA_ALWAYS_SENSITIVE attribute is set to TRUE if and only if the base key has its CKA_ALWAYS_SENSITIVE attribute set to TRUE.
- Similarly, the derived key's CKA_NEVER_EXTRACTABLE attribute is set to TRUE if and only if the base key has its CKA_NEVER_EXTRACTABLE attribute set to TRUE.

CKM_DES_DERIVE_ECB

The CKM_DES_DERIVE_ECB and CKM_DES3_DERIVE_ECB mechanisms are used with the *C_DeriveKey* function to derive a secret key by performing an ECB (no padding) encryption. They create a new secret key whose value is generated by encrypting the provided data with the provided single, double or triple length DES key.

The CKM_DES_DERIVE_ECB and CKM_DES3_DERIVE_ECB mechanisms require the pParameter in the CK_MECHANISM structure to be the pointer to the data that is to be encrypted. The parameterLen is the length of the provided data, which must be a multiple of 8 bytes.

The following rules apply to the provided attribute template:

- If no length or key type is provided in the template, then the key produced by these mechanisms is a generic secret key. Its length is equal to the length of the provided data.
- If no key type is provided in the template, but a length is, then the key produced by these mechanisms is a generic secret key of the specified length, extracted from the left bytes of the cipher text.
- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by these mechanisms is of the type specified in the template. If it doesn't, an error is returned.
- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by these mechanisms is of the specified type and length, extracted from the left bytes of the cipher text.

If a DES key is derived with these mechanisms, the parity bits of the key are set properly. If the requested type of key requires more bytes than the length of the provided data, an error is generated.

The mechanisms have the following rules about key sensitivity and extractability:

- If the base key has its CKA_SENSITIVE attribute set to TRUE, so does the derived key. If not, then the derived key's CKA_SENSITIVE attribute is set either from the supplied template or else it defaults to TRUE.
- Similarly, the derived key's CKA_EXTRACTABLE attribute is set either from the supplied template or else it defaults to the value of the CKA_EXTRACTABLE of the base key.
- The derived key's CKA_ALWAYS_SENSITIVE attribute is set to TRUE if and only if the base key has its CKA_ALWAYS_SENSITIVE attribute set to TRUE.
- Similarly, the derived key's CKA_NEVER_EXTRACTABLE attribute is set to TRUE if and only if the base key has its CKA_NEVER_EXTRACTABLE attribute set to TRUE.

CKM_DES_ECB_PAD

See the entry for CKM_CAST128_ECB_PAD.

CKM_DES_MDC_2_PAD1

This mechanism is a hash function as defined in ISO/IEC DIS 10118-2 using DES as block algorithm. This mechanism implements padding in accordance with ISO 10118-1 Method 1. Basically, zeros are used to pad the input data to a multiple of eight if required. If the input data is already a multiple of eight, then no padding is added.

CKM_DES_OFB64

Single DES-OFB64 denoted `CKM_DES_OFB64` is a mechanism for single and multiple part encryption and decryption; based on DES Output Feedback Mode.

It has a parameter, an 8-byte initialization vector.

This mechanism does not require either clear text or cipher text to be presented in multiple block lengths. There is no padding required. The mechanism will always return a reply equal in length to the request.

CKM_DES3_DDD_CBC

CKM_DES3_DDD_CBC is a mechanism for single- and multiple-part encryption and decryption, key wrapping and key unwrapping, based on the DES block cipher and cipher-block chaining mode as defined in FIPS PUB 81.

The DES3-DDD cipher encrypts an 8 byte block by `D(KL, D(KR, D(KL, data)))` and decrypts with `E(KL, E(KR, E(KL, cipher)))`; where `Key = KL || KR`, and `E(KL, data)` is a single DES encryption using key `KL` and `D(KL, cipher)` is a single DES decryption.

It has a parameter, an initialization vector for cipher block chaining mode. The initialization vector has the same length as the block size, which is 8 bytes.

Constraints on key types and the length of data are summarized in the following table:

Table 22 – DES3-DDD Block Cipher CBC: Key and Data Length

Function	Key Type	Input Length	Output Length	Comments
C_Encrypt	CKK_DES2	Any	input length rounded up to multiple of block size	no final part
C_Decrypt	CKK_DES2	Multiple of block size	same as input length	no final part
C_WrapKey	CKK_DES2	Any	input length rounded up to multiple of block size	
C_UnwrapKey	CKK_DES2	Any	Determined by type of key being unwrapped or CKA_VALUE_LEN	

For the encrypt and wrap operations, the mechanism performs zero-padding when the input data or wrapped key's length is not a multiple of 8. That is, the value `0x00` is appended to the last block until its length is 8 (for example, plaintext `0x01` would be padded to become `0x010x000x000x000x000x000x000x00`).

With the exception of the algorithm specified in this section, the use of this mechanism is identical to the use of other secret key mechanisms. Therefore, for further details on aspects not covered here (for example, access control, or error codes) refer to the PKCS#11 standard.

CKM_DES3_DERIVE_CBC

See the entry for CKM_DES_DERIVE_CBC.

CKM_DES3_DERIVE_ECB

See the entry for CKM_DES_DERIVE_ECB.

CKM_DES3_ECB_PAD

See the entry for CKM_CAST128_ECB_PAD.

CKM_DES3_OFB64

Triple DES-OFB64 denoted `CKM_DES3_OFB64`is a mechanism for single and multiple part encryption and decryption; based on DES Output Feedback Mode.

It has a parameter, an 8-byte initialization vector.

This mechanism does not require either clear text or cipher text to be presented in multiple block lengths. There is no padding required. The mechanism will always return a reply equal in length to the request.

CKM_DES3_RETAIL_CFB_MAC

This is a signature generation and verification mechanism. The produced MAC is 8 bytes in length. It is an extension of the single length key MAC mechanisms. It takes an 8 byte IV as a parameter, which is encrypted (ECB mode) with the left most key value before the first data block is MAC'ed.

The data, which must be a multiple of 8 bytes, is MAC'ed with the left most key value in the normal manner, but the final cipher block is then decrypted (ECB mode) with the middle key value and encrypted (ECB mode) with the Right most key part.

For double length DES keys, the Right key component is the same as the Left key component.

CKM_DES3_X919_MAC

See the entry for CKM_DES3_X919_MAC_GENERAL.

CKM_DES3_X919_MAC_GENERAL

CKM_DES3_X919_MAC and CKM_DES3_X919_MAC_GENERAL are signature generation and verification mechanisms, as defined by ANSI X9.19. They are an extension of the single length key MAC mechanisms. The data is MAC'ed with the left most key value in the normal manner, but the final cipher block is then decrypted (ECB mode) with the middle key value and encrypted (ECB mode) with the Right most key part.

For double length keys, the Right key component is the same as the Left key component.

CKM_DH_PKCS_PARAMETER_GEN

The PKCS #3 Diffie-Hellman key parameter generation mechanism, denoted `CKM_DH_PKCS_PARAMETER_GEN`, is a key parameter generation mechanism based on Diffie-Hellman key agreement, as defined in PKCS #3. It does not have a parameter.

The mechanism generates Diffie-Hellman key parameters with a particular prime length in bits, as specified in the CKA_PRIME_BITS attribute of the template for the key parameters. The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, CKA_PRIME, CKA_BASE, and CKA_PRIME_BITS attributes to the new object. Other attributes supported by the Diffie-Hellman key parameter types may also be specified in the template for the key parameters, or else are assigned default initial values.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of Diffie-Hellman prime sizes, in bits.

CKM_DSA_PARAMETER_GEN

The DSA key parameter generation mechanism, denoted CKM_DSA_PARAMETER_GEN, is a key parameter generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186. This mechanism does not have a parameter.

The mechanism generates DSA key parameters with a particular prime length in bits, as specified in the CKA_PRIME_BITS attribute of the template for the key parameters. The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, CKA_PRIME, CKA_BASE, CKA_SUBPRIME, and CKA_PRIME_BITS attributes to the new object. Other attributes supported by the DSA key parameter types may also be specified in the template for the key parameters, or else are assigned default initial values.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of DSA prime sizes, in bits.

CKM_DSA_SHA1_PKCS

The PKCS #1 DSA signature with SHA-1 mechanism, denoted CKM_DSA_SHA1_PKCS, performs single and multiple-part digital signature and verification operations without message recovery. The operations performed are as described in PKCS #1 with the object identifier sha1WithDSAEncryption.

It is similar to the PKCS#11 mechanism CKM_RSA_SHA1_PKCSexcept DSA is used instead of RSA. This mechanism has no parameter.

CKM_EC_KEY_PAIR_GEN

The elliptic curve key pair generation mechanism, denoted CKM_EC_KEY_PAIR_GEN, is a key pair generation mechanism for EC Operation.

This mechanism operates as specified in PKCS#11, with the following adjustments.

The CKA_EC_PARAMS or CKA_ECDSA_PARAMS attribute value must be supplied in the Public Key Template. This attribute is known as the “EC domain parameters” and is defined in ANSI X9.62 as a choice of three parameter representation methods with the following syntax:

```
Parameters ::= CHOICE {
ecParameters ECParameters,
namedCurve CURVES.&id({CurveNames}),
implicitlyCA NULL
}
```

If the CKA_EC_PARAMS attribute contains a namedCurve then it must be the of DER OID-encoding of one of the following supported curves:

- { iso(1) member-body(2) US(840) x9-62(10045) curves(3) characteristicTwo(0) c2tnb191v1(5) }
- { iso(1) member-body(2) US(840) x9-62(10045) curves(3) prime(1) prime192v1(1) }
- { iso(1) identified-organization(3) Certicom(132) certicom_ellipticCurve(0) secp224r1(33) }
- { iso(1) member-body(2) US(840) x9-62(10045) curves(3) prime(1) prime256v1(7) }
- { iso(1) identified-organization(3) Certicom(132) certicom_ellipticCurve(0) secp384r1(34) }

- { iso(1) identified-organization(3) Certicom(132) certicom_ellipticCurve(0) secp521r1(35) }

Plus the custom curve with unofficial OID:

- { iso(1) member-body(2) US(840) x9-62(10045) curves(3) characteristicTwo(0) c2tnb191v1e (15) }

Refer to the CT_DerEncodeNamedCurve function in the CTUTIL library for a convenient way to obtain the encodings of supported namedCurve OIDs.

If the CKA_EC_PARAMS attribute is in the form of the ECParameters sequence then the domain parameters may be described explicitly. In this way the developer is able to specify the curve parameters for curves that the firmware has no prior knowledge of.

Support for ECParameters sequence is disabled unless the Security Configuration "User Specified ECC Domain Parameters Allowed"is enabled (see ctconf –fE).

Refer to the CT_GetECCDomainParameters function in the CTUTILS library and the KM_EncodeECParamsP and KM_EncodeECParams2M functions from the KMLIB library for convenient methods to obtain ECParameters encodings.

CKM_ECDH1_DERIVE

The elliptic curve Diffie-Hellman (ECDH) key derivation mechanism, denoted CKM_ECDH1_DERIVE, is a mechanism for key derivation based on the Diffie-Hellman version of the elliptic curve key agreement scheme, as defined in ANSI X9.63, where each party contributes one key pair all using the same EC domain parameters.

This mechanism has a parameter, a CK_ECDH1_DERIVE_PARAMS structure.

```
typedef struct CK_ECDH1_DERIVE_PARAMS {
    CK_EC_KDF_TYPE kdf;       /* key derivation function */
    CK_ULONG  ulSharedDataLen;      /* optional extra shared data */
    CK_BYTE_PTR pSharedData;
    CK_ULONG   ulPublicDataLen;     /* other party public key value */
    CK_BYTE_PTR  pPublicData;
} CK_ECDH1_DERIVE_PARAMS;
typedef struct CK_ECDH1_DERIVE_PARAMS * CK_ECDH1_DERIVE_PARAMS_PTR;
```

The fields of the structure have the following meanings:

kdf	This is the Key Derive Function (see below for the description of the possible values of this field).
ulSharedDataLen	This is the length of the optional shared data used by some of the key derive functions. This may be zero if there is no shared data.
pSharedData	This is the address of the optional shared data or NULL if there is no shared data.
ulPublicDataLen	This is the length of the other party public key.
pPublicData	This is the pointer to the other party public key. Only uncompressed format is accepted.

The mechanism calculates an agreed value using the EC Private key referenced by the base object handle and the EC Public key passed to the mechanism through the pPublicData field of the mechanism parameter.

The length of the agreed value is equal to the 'q' value of the underlying EC curve.

The agreed value is then processed by the Key Derive Function (kdf) to produce the CKA_VALUE of the new Secret Key object.

Four main types of KDFs are supported:

- The NULL KDF performs no additional processing and can be used to obtain the raw agreed value. Basically: Key = Z

- The CKF_<hash>_KDF algorithms are based on the algorithm described in section 5.6.3 of ANSI X9.63 2001. Basically: Key = H(Z || *counter* || *OtherInfo*)

- The CKF_<hash>_SES_KDF algorithms are based on the variant of the x9.63 algorithm specified in *Technical Guideline TR-03111 - Elliptic Curve Cryptography (ECC) based on ISO 15946 Version 1.0,* Bundesamt Fur Sicherheit in der Informationstechnik (BSI)

  Basically: Key = H(Z || *counter*) where *counter* is a user specified parameter

- The CKF_<hash>_NIST_KDF algorithms are based on the algorithm described in NIST 800-56A Concatenisation Algorithm

  Basically: Key = H(*counter* || Z || *OtherInfo*)

The CKF_SES_<hash>_KDF algorithms require the value of the counter to be specified. This is done by arithmetically adding the counter value to the CKF value.

The following Counter values are defined in TR-03111:

Counter Name	Value	Description
CKD_SES_ENC_CTR	0x00000001	Default encryption Key
CKD_SES_AUTH_CTR	0x00000002	Default authentication Key
CKD_SES_ALT_ENC_CTR	0x00000003	Alternate encryption Key
CKD_SES_ALT_AUTH_CTR	0x00000004	alternate Authentication Key
CKD_SES_MAX_CTR	0x0000FFFF	Maximum counter value

For example:

To derive a session key to be used as an Alternate key for Encryption the counter must equal 0x00000003. If the SHA-1 hash algorithm is required then the kdf value would be set like this:

```
CK_ECDH1_DERIVE_PARAMS Params;
Params.kdf = CKD_SHA1_SES_KDF + CKD_SES_ALT_ENC_CTR;
```

The table below describes the supported KDFs.

KDF Type	Description
CKD_NULL	The null transformation. The derived key value is produced by taking bytes from the left of the agreed value. The new key size is limited to the size of the agreed value. The Shared Data is not used by this KDF and pSharedData should be NULL.
CKD_SHA1_KDF	This KDF generates secret keys of virtually any length using the algorithm described in X9.63 with the SHA-1 hash algorithm. Shared data may be provided.
CKD_SHA224_KDF	This KDF generates secret keys of virtually any length using the algorithm described in X9.63 with the SHA-224 hash algorithm. Shared data may be provided.
CKD_SHA256_KDF	This KDF generates secret keys of virtually any length using the algorithm described in X9.63 with the SHA-256 hash algorithm. Shared data may be provided.

KDF Type	Description
CKD_SHA384_KDF	This KDF generates secret keys of virtually any length using the algorithm described in X9.63 with the SHA-384 hash algorithm. Shared data may be provided.
CKD_SHA512_KDF	This KDF generates secret keys of virtually any length using the algorithm described in X9.63 with the SHA-512 hash algorithm. Shared data may be provided.
CKD_RIPEMD160_KDF	This KDF generates secret keys of virtually any length using the algorithm described in X9.63 with the RIPE MD 160 hash algorithm. Shared data may be provided. This KDF is not available if the HSM is configured for "Only allow Fips Approved Algorithms".
CKD_SHA1_SES_KDF	This KDF generates session keys. It uses the algorithm described in TR-03111 with the SHA-1 hash algorithm. Shared data may be provided but typically it is not used. The counter value that is a parameter to this KDF must be added to this constant.
CKD_SHA224_SES_KDF	This KDF generates single, double and triple length DES keys that are intended for Encryption operations. It uses the algorithm described in TR-03111 with the SHA-224 hash algorithm. Shared data may be provided but typically it is not used. The counter value that is a parameter to this KDF must be added to this constant.
CKD_SHA256_SES_KDF	This KDF generates single, double and triple length DES keys that are intended for Encryption operations. It uses the algorithm described in TR-03111 with the SHA-256 hash algorithm. Shared data may be provided but typically it is not used. The counter value that is a parameter to this KDF must be added to this constant.
CKD_SHA384_SES_KDF	This KDF generates single, double and triple length DES keys that are intended for Encryption operations. It uses the algorithm described in TR-03111 with the SHA-384 hash algorithm. Shared data may be provided but typically it is not used. The counter value that is a parameter to this KDF must be added to this constant.
CKD_SHA512_SES_KDF	This KDF generates single, double and triple length DES keys that are intended for Encryption operations. It uses the algorithm described in TR-03111 with the SHA-512 hash algorithm. Shared data may be provided but typically it is not used. The counter value that is a parameter to this KDF must be added to this constant.
CKD_RIPEMD160_SES_KDF	This KDF generates single, double and triple length DES keys that are intended for Encryption operations. It uses the algorithm described in TR-03111 with the Ripe MD 160 hash algorithm. Shared data may be provided but typically it is not used. The counter value that is a parameter to this KDF must be added to this constant. This KDF is not available if the HSM is configured for "Only allow Fips Approved Algorithms".
CKD_SHA1_NIST_KDF	This KDF generates secret keys of virtually any length using the algorithm described in NIST 800-56A with the SHA-1 hash algorithm. Shared data should be formatted according to the standard.
CKD_SHA224_NIST_KDF	This KDF generates secret keys of virtually any length using the algorithm described in NIST 800-56A with the SHA-224 hash algorithm. Shared data should be formatted according to the standard.

KDF Type	Description
CKD_SHA256_NIST_KDF	This KDF generates secret keys of virtually any length using the algorithm described in NIST 800-56A with the SHA-256 hash algorithm. Shared data should be formatted according to the standard.
CKD_SHA384_NIST_KDF	This KDF generates secret keys of virtually any length using the algorithm described in NIST 800-56A with the SHA-384 hash algorithm. Shared data should be formatted according to the standard.
CKD_SHA512_NIST_KDF	This KDF generates secret keys of virtually any length using the algorithm described in NIST 800-56A with the SHA-512 hash algorithm. Shared data should be formatted according to the standard.
CKD_RIPEMD160_NIST_KDF	This KDF generates secret keys of virtually any length using the algorithm described in NIST 800-56A with the RIPE MD 160 hash algorithm. Shared data should be formatted according to the standard. This KDF is not available if the HSM is configured for "Only allow Fips Approved Algorithms".

This mechanism derives a secret value, and truncates the result according to the CKA_KEY_TYPE attribute of the template and, if it has one and the key type supports it, the CKA_VALUE_LEN attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the CKA_VALUE attribute of the new key; other attributes required by the key type must be specified in the template.

The following rules apply to the provided attribute template:

- A key type must be provided in the template or else a Template Error is returned.
- If no length is provided in the template then that key type must have a well-defined length. If it doesn't, an error is returned.
- If both a key type and a length are provided in the template, the length must be compatible with that key type.
- If a DES key is derived with these mechanisms, the parity bits of the key are set properly.
- If the requested type of key requires more bytes than the Key Derive Function can provide, an error is generated.

The mechanisms have the following rules about key sensitivity and extractability:

- The CKA_SENSITIVE, CKA_EXTRACTABLE and CKA_EXPORTABLE attributes in the template for the new key can both be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes all take on the default value TRUE.
- If the base key has its CKA_ALWAYS_SENSITIVE attribute set to CK_FALSE, then the derived key will as well. If the base key has its CKA_ALWAYS_SENSITIVE attribute set to CK_TRUE, then the derived key has its CKA_ALWAYS_SENSITIVE attribute set to the same value as its CKA_SENSITIVE attribute.
- Similarly, if the base key has its CKA_NEVER_EXTRACTABLE attribute set to CK_FALSE, then the derived key will, too. If the base key has its CKA_NEVER_EXTRACTABLE attribute set to CK_TRUE, then the derived key has its CKA_NEVER_EXTRACTABLE attribute set to the opposite value from its CKA_EXTRACTABLE attribute.

CKM_ECIES

The Elliptic Curve Integrated Encryption Scheme (ECIES) mechanism, denoted CKM_ECIES, performs single-part encryption and decryption operations. The operations performed are as described in ANSI X9.63-2001.

This mechanism has a parameter, a CK_ECIES_PARAMS structure. This structure is defined as follows:

```
typedef struct CK_ECIES_PARAMS
{
     CK_EC_DH_PRIMITIVE dhPrimitive;
    CK_EC_KDF_TYPE kdf;
    CK_ULONG ulSharedDataLen1;
    CK_BYTE_PTR pSharedData1;
    CK_EC_ENC_SCHEME encScheme;
    CK_ULONG ulEncKeyLenInBits;
    CK_EC_MAC_SCHEME macScheme;
    CK_ULONG ulMacKeyLenInBits;
    CK_ULONG ulMacLenInBits;
    CK_ULONG ulSharedDataLen2;
    CK_BYTE_PTR pSharedData2;
} CK_ECIES_PARAMS;
```

The fields of this structure have the following meanings:

dhPrimitive	This is the Diffie-Hellman primitive used to derive the shared secret value. Valid value: **CKDHP_STANDARD**
kdf	This is the key derivation function used on the shared secret value. Valid value: **CKD_SHA1_KDF**
ulSharedDataLen1	This is the length in bytes of the key derivation shared data.
pSharedData1	This is the key derivation padding data shared between the two parties.
encScheme	This is the encryption scheme used to transform the input data. Valid value: **CKES_XOR**
ulEncKeyLenInBits	This is the bit length of the key to use for the encryption scheme.
macScheme	This is the MAC scheme used for MAC generation or validation. Valid values: **CKMS_HMAC_SHA1** **CKMS_SHA1** **NB:** The MAC scheme **CKMS_SHA1**, should **only** be used for compatability with RSA BSAFE® Crypto-C, which uses a NON-STANDARD MAC scheme, which was defined in the 10/97 X9.63 Draft, but was removed from the released ANSI X9.63-2001 specification.

ulMacKeyLenInBits	This is the bit length of the key to use for the MAC scheme.
ulMacLenInBits	This is the bit length of the MAC scheme output.
ulSharedDataLen2	This is the length in bytes of the MAC shared data.
pSharedData2	This is the MAC padding data shared between the two parties.

The *pSharedData1* and *pSharedData2* parameters are optional, and if not supplied then they must be NULL and the *ulSharedDataLen1* and *ulSharedDataLen2* parameters must be zero. With the MAC scheme CKMS_SHA1, any supplied shared data is ignored.

With the encryption scheme CKES_XOR, the *ulEncKeyLenInBits* parameter MUST be zero. With any other encryption scheme, the *ulEncKeyLenInBits* parameter must be set to the applicable key length in bits.

With the MAC scheme CKMS_SHA1, the *ulMacKeyLenInBits* parameter must be 0. With any other MAC scheme, the *ulMacKeyLenInBit*s parameter must be a minimum of 80 bits, and a multiple of 8 bits.

The *ulMacLenInBits* parameter must be a minimum of 80 bits, a multiple of 8 bits, and not greater than the maximum output length for the specified Hash.

Constraints on key types and the length of the data are summarized in the following table.

Table 239 – ECIES: Key and Data Length

Function	Key Type	Input Length	Output Length
C_Encrypt	EC public key	any	1 + 2modLen + any + macLen
C_Decrypt	EC private key	1 + 2modLen + any + macLen	any

Where:

- modLen is the curve modulus length
- macLen is the length of the produced MAC

The encrypted data is in the format QE||EncData||MAC, where:

- QE is the uncompressed bit string of the ephemeral EC public key
- EncData is the encrypted data
- MAC is the generated MAC

CKM_ENCODE_ATTRIBUTES

This wrapping mechanism takes the attributes of an object and encodes them. The encoding is not encrypted therefore the wrapping key object handle parameter is ignored.

If the object is sensitive then only non-sensitive attributes of the object are encoded. The encoding format is a simple proprietary encoding with the attribute type, length, a value presence indicator (Boolean) and the attribute value. This simple encoding format is used wherever BER or DER is not required.

CKM_ENCODE_PKCS_10

This mechanism is used with the C_DeriveKey function to create a PKCS#10 certification request from a public key. Either an RSA or DSA public key may be used with this function. The PKCS#10 certificate request could then be sent to a Certificate authority for signing.

From PKCS#10

A certification request consists of a distinguished name, a public key and optionally a set of attributes that are collectively signed by the entity requesting certification. Certification requests are sent to a certification authority, which will transform the request to an X.509 public-key certificate.

Usage

- Use CKM_RSA_PKCS_KEY_PAIR_GEN to generate a key.
- Add a CKA_SUBJECT attribute to the public key, containing the subject's distinguished name.
- Initialize the signature mechanism to sign the request. Note that a digest/sign mechanism must be chosen. For example, CKM_SHA1_RSA_PKCS
- Call C_DeriveKey with the CKM_ENCODE_PKCS_10 mechanism to perform the generation.
- On success, an object handle for the certificate request is returned.
- The object's CKA_VALUE attribute contains the PKCS#10 request.

CKM_ENCODE_PUBLIC_KEY

This wrapping mechanism performs a DER encoding of a Public Key object. The encoding is not encrypted therefore the wrapping key object handle parameter is ignored.

Public keys of type CKK_RSA, CKK_DSA and CKK_DH may be encoded with this mechanism. The encoding format is defined in PKCS#1. This mechanism has no parameter.

CKM_ENCODE_X_509

This mechanism is used with the C_DeriveKey function to derive an X.509 certificate from a public key or a PKCS#10 certification request. This mechanism creates a new X.509 certificate based on the provided public key or certification request signed with a CA key. This mechanism takes no parameter.

The new certificate validity period is based on the CKA_START_DATE and CKA_END_DATE attributes on the base object. If the start date is missing the current time is used. If the end date is missing the certificate is valid for one year. These dates may be specified as relative values by adding the + character at the start of the date value. The start date is relative to 'now' and the end date is relative to the start date if relative times are specified. Negative relative times are not allowed. If the start or end date is invalid then the error CKR_TEMPLATE_INCONSISTENT is returned.

The certificate's serial number is taken from the template's CKA_SERIAL_NUMBER, CKA_SERIAL_NUMBER_INT or the signing key's CKA_USAGE_COUNT in that order. If none of these values is available CKR_WRAPPING_KEY_HANDLE_INVALID error is returned.

To determine the Subject distinguished name for the new certificate if the base object is a public key the algorithm will use the CKA_SUBJECT_STR, CKA_SUBJECT from the template or the base key (in that order). If none of these values is available CKR_KEY_HANDLE_INVALID is returned.

It is also possible to include arbitrary X.509 extensions in the certificate. These are not verified for validity nor parsed for correctness. Rather they are included verbatim in the newly generated certificate. In order to specify an extension use the CKA_PKI_ATTRIBUTE_BER_ENCODED attribute with the value specified as a BER encoding of the attribute. If the base object is a Certification request or a self-signed certificate the subject is taken from the objects encoded subject name.

Currently this mechanism supports generation of RSA or DSA certificates. On success, a handle to a new CKO_CERTIFICATE object is returned. The certificate will include the CKA_ISSUER, CKA_SERIAL_NUMBER and CKA_SUBJECT attributes as well as a CKA_VALUE attribute which will contain the DER encoded certificate.

To create a X.509 certificate that uses EC keys, either provide a PKCS#10 certificate request that was created with EC keys, or provide an EC public key for the hBaseKey parameter to the function. To sign the certificate as a CA using EC keys, use the CKM_ECDSA_SHA1 mechanism to initialise the sign operation before calling C_DeriveKey().

Usage

- Create a key-pair using the `CKM_RSA_PKCS` mechanism (this is the key-pair for the new certificate), or
- Create a `CKO_CERTIFICATE_REQUEST` object (with the object's `CKA_VALUE` attribute set to the PKCS#10 data)
- This object is the "base-key" used in the C_DeriveKey function
- Initialize the signature mechanism to sign the request using `C_SignInit`. Note that a digest / sign mechanism must be chosen. For example, CKM_SHA1_RSA_PKCS
- Call C_DeriveKey with CKM_ENCODE_X_509 to perform the generation

The new certificate's template may contain:

CKA_ISSUER_STR CKA_ISSUER	The distinguished name of the issuer of the new certificate. If this attribute is not included the issuer is taken from the signing key's CKA_SUBJECT attribute. CKA_ISSUER is the encoded version of this attribute.
CKA_SERIAL_NUMBER_INT CKA_SERIAL_NUMBER	The serial number of the new certificate. If this attribute is not included the serial number is set to the value of the CKA_USAGE_COUNT attribute of the signing key. CKA_SERIAL_NUMBER is the encoded version of this attribute.
CKA_SUBJECT_STR CKA_SUBJECT	If the base key (i.e. the input object) is a public key then either the template must contain this attribute or the public key must have a CKA_SUBJECT attribute. This attribute contains the distinguished name of the subject. When the base key is a PKCS#10 certification request the CKA_SUBJECT information is taken from there. CKA_SUBJECT is the encoded version of this attribute.
CKA_START_DATE CKA_END_DATE	These attributes are used to determine the new certificate's validity period. If the start date is missing the current date is used. If the end date is missing the date is set to one year from the start date. Relative values may be specified (see above).
CKA_PKI_ATTRIBUTE_BER _ENCODED	These attributes are used to determine the new certificate's extended attributes.

CKM_ENCODE_X_509_LOCAL_CERT

This mechanism is similar to the `CKM_ENCODE_X_509` mechanism in that it is used to create an X 509 public key certificate. The basic difference is that this mechanism has additional usage controls.

This mechanism will only create certificates for public keys locally generated on the adapter. That is, the base key must have a `CKA_CLASS` attribute of `CKO_PUBLIC_KEY`and have the `CKA_LOCAL` attribute set to `TRUE`.

In addition, the signing key specified in the mechanism parameter (see below) must have the `CKA_SIGN_LOCAL_CERT` attribute set to `TRUE`. It is used with the `C_KeyDerive` function only, (that is, it is a derive mechanism).

It takes a parameter that is a pointer to a `CK_MECH_TYPE_AND_OBJECT` structure.

```
typedef struct CK_MECH_TYPE_AND_OBJECT {
CK_MECHANISM_TYPE mechanism;
CK_OBJECT_HANDLE obj;
} CK_MECH_TYPE_AND_OBJECT;
```

The above mechanism field specifies the actual signature mechanism to use in generation of the certificate signature. This must be one of the multipart digest RSA or DSA algorithms. The `obj` field above specifies the signature generation key. That is, it should specify a RSA or DSA private key as appropriate for the chosen signature mechanism.

To create a X.509 local certificate that uses EC keys, either provide a PKCS#10 certificate request that was created with EC keys, or provide an EC public key for the `hBaseKey` parameter to the function. To sign the certificate as a CA using EC keys, use the `CKM_ECDSA_SHA1` mechanism to initialize the sign operation before calling `C_DeriveKey()`. The `CKM_ECDSA_SHA1` mechanism and EC key must also be specified in the mechanism parameter.

CKM_IDEA_ECB_PAD

See the entry for CKM_CAST128_ECB_PAD.

CKM_NVB

This is a message digest mechanism. It is an implementation of the NVB (Nederlandse Vereiniging van Banken) Dutch hash standard. This hash algorithm is also known as the BGC hash, version 7.1. This mechanism is only available in the software Emulation version of the PTK C.

CKM_KEY_TRANSLATION

This is a key wrapping mechanisms as used by Entrust compliant applications. This mechanism is only visible when the `CKF_ENTRUST_READY` flag is set in the SecurityMode attribute of the Adapter Configuration object in the Admin Token of the adapter.

CKM_PBA_SHA1_WITH_HMAC_SHA1

This is a mechanism used for generating a 160-bit generic secret key from a password and a salt value by using the SHA-1 digest algorithm and an iteration count.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process. The parameter also has a field to hold the location of an application-supplied buffer which will receive an IV; for this mechanism, the contents of this field are ignored, since authentication with SHA-1-HMAC does not require an IV.

The key generated by this mechanism will typically be used for computing a SHA-1 HMAC to perform password-based authentication (not *password-based encryption*). At the time of this writing, this is primarily done to ensure the integrity of a PKCS #12 PDU.

CKM_PBE_SHA1_RC2_128_CBC

This is a mechanism used for generating a 128-bit RC2 secret key and IV from a password and a salt value by using the SHA-1 digest algorithm and an iteration count.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer that will receive the 8-byte IV generated by the mechanism.

When the key and IV generated by this mechanism are used to encrypt or decrypt, the effective number of bits in the RC2 search space should be set to 128. This ensures compatibility with the ASN.1 Object Identifier `pbeWithSHA1And128BitRC2-CBC`.

The key and IV produced by this mechanism will typically be used for performing password-based encryption.

CKM_PBE_SHA1_RC2_40_CBC

This is a mechanism used for generating a 40-bit RC2 secret key and IV from a password and a salt value by using the SHA-1 digest algorithm and an iteration count.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer that will receive the 8-byte IV generated by the mechanism.

When the key and IV generated by this mechanism are used to encrypt or decrypt, the effective number of bits in the RC2 search space should be set to 40. This ensures compatibility with the ASN.1 Object Identifier `pbeWithSHA1And40BitRC2-CBC`.

The key and IV produced by this mechanism will typically be used for performing password-based encryption.

CKM_PBE_SHA1_RC4_128

This is a mechanism used for generating a 128-bit RC4 secret key from a password and a salt value by using the SHA-1 digest algorithm and an iteration count.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process. The parameter also has a field to hold the location of an application-supplied buffer that will receive an IV; for this mechanism, the contents of this field are ignored, since RC4 does not require an IV. The key produced by this mechanism will typically be used for performing password-based encryption.

CKM_PBE_SHA1_RC4_40

This is a mechanism used for generating a 40-bit RC4 secret key from a password and a salt value by using the SHA-1 digest algorithm and an iteration count.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process. The parameter also has a field to hold the location of an application-supplied buffer which will receive an IV; for this mechanism, the contents of this field are ignored, since RC4 does not require an IV.

The key produced by this mechanism will typically be used for performing password-based encryption.

CKM_PKCS12_PBE_EXPORT

The PKCS#12 export mechanism, denoted `CKM_PKCS12_PBE_EXPORT` is a mechanism for wrapping a private key and a certificate. The outcome of the wrapping operation is a PKCS#12 byte buffer.

This mechanism has a parameter, a **CK_PKCS12_PBE_EXPORT_PARAMS** structure.

This mechanism will enforce a password length based on the token. If the PIN is too short, then **CKR_PIN_LEN_RANGE** is returned.

This mechanism does **not** require a wrapping key and it only support RSA, ECDSA and DSA private keys and certificates.

During the wrapping operation, this mechanism performs a sign and verify test on the supplied key/certificate pair. Should this test fail, the wrapping operation will abort.

If the exported key is marked CKA_EXPORTABLE=TRUE and CKA_EXTRACTABLE=FALSE this mechanism forces the export to be performed under the Security Officer session. In this case, the user must ensure that the private key is either visible to the Security Ofiicer or made available to the Security Officer by performing a copy.

Note that the user performing the private key export is asked to supply two (2) passwords. These passwords must be identical if MS Windows is to be used to later extract the created PKCS#12 file. For other 3rd party tools such as OpenSSL these two passwords do not have to be the same.

CK_PKCS12_PBE_EXPORT_PARAMS is a structure that provides parameter to the CKM_PKCS12_PBE_EXPORT mechanism. This structure is defined as follows:

```
typedef struct CK_PKCS12_PBE_EXPORT_PARAMS
{
CK_OBJECT_HANDLE keyCert;
CK_CHAR_PTR passwordAuthSafe;
CK_SIZE passwordAuthSafeLen;
CK_CHAR_PTR passwordHMAC;
CK_SIZE passwordHMACLen;
CK_MECHANISM_TYPE safeBagKgMech;
CK_MECHANISM_TYPE safeContentKgMech;
CK_MECHANISM_TYPE hmacKgMech;
}
```

The fields of the structure have the following meanings:

keyCert	This is the certificate handle for the associated private key.
passwordAuthSafe	This is the password for the PBE keys.
passwordAuthSafeLen	This is the length of the password.
passwordHMAC	This is the password for the PBA keys.
passwordHMACLen	This is the length of the password.
safeBagKgMech	This is the key generation mechanism for SafeBag encryption. It is only applicable to pkcs8ShroudedKeyBag. Valid options are: **CKM_PBE_SHA1_RC4_128** **CKM_PBE_SHA1_RC4_40** **CKM_PBE_SHA1_DES3_EDE_CBC** **CKM_PBE_SHA1_DES2_EDE_CBC** **CKM_PBE_SHA1_RC2_128_CBC** **CKM_PBE_SHA1_RC2_40_CBC**
safeContentKgMech	This is the key generation mechanism for SafeContent encryption. It is only applicable to EncryptedData. Valid options are: **CKM_PBE_SHA1_RC4_128** **CKM_PBE_SHA1_RC4_40** **CKM_PBE_SHA1_DES3_EDE_CBC** **CKM_PBE_SHA1_DES2_EDE_CBC** **CKM_PBE_SHA1_RC2_128_CBC** **CKM_PBE_SHA1_RC2_40_CBC**
hmacKgMech	This is the key generation mechanism for generating PFX MAC. Valid option is: **CKM_PBA_SHA1_WITH_SHA1_HMAC**

CKM_PKCS12_PBE_IMPORT

The PKCS#12 import mechanism, denoted CKM_PKCS12_PBE_IMPORT is a mechanism for unwrapping a private key and certificate(s). This mechanism shall return the user a handle to a private key and handle(s) to certificate(s). Note that multiple certificate handles could be returned depending on the contents of the PKCS#12 file.

NOTE: This mechanism does **not** import optional PKCS#12 bag attributes and PKCS#8 private-key attributes. These components are discarded during import.

The mechanism has a parameter, a CK_PKCS12_PBE_IMPORT_PARAMS structure. This mechanism does **not** require an unwrapping key and supports RSA, DH, DSA and EC Private Keys and certificates.

CK_PKCS12_PBE_IMPORT_PARAMS is a structure that provides parameters to the CKM_PKCS12_PBE_IMPORT mechanism. This structure is defined as follows:

```
typedef struct CK_PKCS12_PBE_IMPORT_PARAMS
{
/** AuthenticatedSafe password */
CK_CHAR_PTR passwordAuthSafe;

/** Size of AuthenticatedSafe password */
CK_SIZE passwordAuthSafeLen;
/** HMAC password */
CK_CHAR_PTR passwordHMAC;
/** Size of HMAC password */
CK_SIZE passwordHMACLen;
/** Certificate attributes */
CK_ATTRIBUTE_PTR certAttr;
/** Number of certificate attributes */
CK_COUNT certAttrCount;
/** Handle to returned certificate(s) */
CK_OBJECT_HANDLE_PTR hCert;
/** Number of returned certificate handle(s) */
CK_COUNT_PTR hCertCount;
}CK_PKCS12_PBE_IMPORT_PARAMS;
```

The fields of the structure have the following meanings:

passwordAuthSafe	This is the password to the authenticated safe container.
passwordAuthSafeLen	This is the length of password.
passwordHMAC	This is the password to HMAC.
certAttr	These are the attributes assigned to certificate.
certAttrCount	This is the number of entries in certAttr.
hCert	This is the returned certificate handle(s).
hCertCount	This is the number of handles allocated for hCert or the number of certificates found in PKCS#12 file. See below.

Length Prediction

The PKCS#12 file may contain more than one certificate, as such, the user would need to allocate sufficient buffer to hold the returned handles. The user needs to specify NULL as a parameter to the returned certificate handle (`hCert`), the import mechanism shall then return a count (`hCertCount`) of the certificate found the in the PKCS#12 file. Using the value of `hCertCount`, the user then allocates the required buffer to hold the returned certificate handles for the next `C_UnwrapKey` function call.

Returning Multiple Ceritificates

Assuming the user has allocated sufficient buffer to hold the certificate handles and there is multiple certificate in the PKCS#12 files, the import mechanism shall populate buffer hCert with the allocated certificate handles. The returned hCertCount shall match the specified value.

Reporting Remaining Certificates

In the event of the user not reserving sufficient buffer in hCert and there are more certificates to be unwrapped, the import mechanism shall unwrap up to a maximum of cerficate handles allocated by the user and return the total count of the certificates found in the PKCS#12 file. For example, if the user initially allocated one handle (`hCertCount=1`) and the PKCS#12 contains 2 certificates, the import mechanism shall extract the first certificate it encounters and return `hCertCount=2`. In this case, the returned `hCertCount` shall always be **larger** than the specified value.

PKCS#12 Import Return Code

The following vendor specific return code may be returned in the event of errors:

CKR_PKCS12_DECODE	This error code is returned when there is an error decoding the PKCS#12 file.
CKR_PKCS12_UNSUPPORTED_SAFEBAG_TYPE	This error code is returned when unsupported SafeBag is found. The import mechanism for this release only supports keyBag, pkcs8ShroudedKeyBag, and certBag.
CKR_PKCS12_UNSUPPORTED_PRIVACY_MODE	This error code is returned when a PKCS#12 file with unsupported privacy mode is encountered. The import mechanism for this release only supports password privacy mode.
CKR_PKCS12_UNSUPPORTED_INTEGRITY_MODE	This error code is returned when a PKCS#12 file with unsupported integrity mode is encountered. The import mechanism for this release only supports password integrity mode.

CKM_PP_LOAD_SECRET

This is a key generate mechanism to provide the capability to load a clear key component from a directly attached pin pad device.

It has a parameter, a CK_PP_LOAD_SECRET_PARAMS, which holds the operational details for the mechanism.

```
struct CK_PP_LOAD_SECRET_PARAMS
{
/** Entered characters should be masked with '*' or similar to hide the

* value being entered. An error is returned if this is TRUE
* and the device does not support this feature. */
 CK_BBOOL bMaskInput;


/** Entered characters should be converted from the ASCII representation
* to binary before being stored, according to the conversion type
* supplied. If the device does not support the specified type of input
* (e.g. hex input on a decimal keyboard), an error is returned.
* The octal and decimal representations will expect 3 digits per byte,
```

```
* whereas the hexadecimal representations will expect 2 digits per byte.
* An error is returned if the data contains invalid encoding (such
* as 351 for decimal conversion).

*/
 CK_PP_CONVERT_TYPE cConvert;

/** The time to wait for operator response - in seconds. An error is
* returned if the operation does not complete in the specified time.
* This field may be ignored if the device does not support a configurable
* timeout. */
 CK_CHAR cTimeout;


/** Reserved for future extensions. Must be set to zero. */
 CK_CHAR reserved;
/** The prompt to be displayed on the device. If the prompt cannot fit on
* the device display, the output is clipped. If the device does not
* have any display, the operation will continue without any prompt, or
* error.
*

* The following special characters are recognized on the display:
* - Newline (0x0a): Continue the display on the next line.
*/


 CK_CHAR_PTR prompt;
};
```

The template supplied with the call to the C_GenerateKey function determines the type of object generated by the operation. CKA_CLASS may be CKO_SECRETKEY only, and the only key type supported is CKK_GENERIC_SECRET. (This restriction applies because only key components are to be entered by this mechanism).

The normal rules for template consistencies apply. In particular the CKA_ALWAYS_SENSITIVE must be set FALSE and the CKA_NEVER_EXTRACTABLE must be FALSE.

The expected size of the object value created by this operation is supplied in the CKA_VALUE_LEN parameter in the template.

CKM_RC2_ECB_PAD

See the entry for CKM_CAST128_ECB_PAD.

CKM_REPLICATE_TOKEN_RSA_AES

This mechanism is a SafeNet vendor defined mechanism for wrapping and unwrapping tokens.

Wrapping Tokens

The mechanism wraps the token associated with the *hSession* parameter to C_WrapKey() into a protected format. When the mechanism is used to wrap a token it has a required parameter, a CK_REPLICATE_TOKEN_PARAMS_PTR.

The CK_REPLICATE_TOKEN_PARAMS structure is defined as follows:

```
typedef struct CK_REPLICATE_TOKEN_PARAMS {
CK_CHAR peerId[CK_SERIAL_NUMBER_SIZE];
} CK_REPLICATE_TOKEN_PARAMS;
```

The *peerId* field identifies the peer public key on the administrative token. The public key is used to wrap the token encryption key and therefore must identify the public key of the destination HSM.

`CK_REPLICATE_TOKEN_PARAMS_`PTR is a pointer to a `CK_REPLICATE_TOKEN_PARAMS`.

The following conditions must be satisfied:

- The token being wrapped which is associated with the *hSession* parameter to the C_WrapKey() must be a regular user token (i.e. NOT the administrative token or a smart-card token).
- The session state for *hSession* must be one of CKS_RO_USER_FUNCTIONS or CKS_RW_USER_FUNCTIONS.
- The *hWrappingKey* parameter to *C_WrapKey()* must specify CK_INVALID_HANDLE.
- The *hKey* parameter to *C_WrapKey()* must specify CK_INVALID_HANDLE.

Unwrapping Tokens

This mechanism unwraps the protected token information, replacing the entire token contents of the token associated with the *hSession* parameter to C_UnwrapKey().When the mechanism is used for unwrapping a token, a mechanism parameter must not be specified.

The following conditions must be satisfied:

- The token being unwrapped which is associated with the *hSession* parameter to *C_UnwrapKey()* must be a regular user token. That is, NOT the administrative token or a smart card token.
- The session state for *hSession* must be CKS_RW_USER_FUNCTIONS.
- The *hUnwrappingKey* parameter to *C_UnwrapKey()* must specify CK_INVALID_HANDLE.
- The *pTemplate* parameter to *C_UnwrapKey()* must specify NULL.
- The *ulAttributeCount* parameter to *C_UnwrapKey()* must specify zero.
- The *phKey* parameter to *C_UnwrapKey()* must specify NULL.
- Any new sessions must be deferred until the operation has finished.
- The current session must be the only session in existence for the token.
- The application should call C_Finalize() upon completion.

CKM_RSA_PKCS_KEY_PAIR_GEN

The mechanism denoted CKM_RSA_PKCS_KEY_PAIR_GEN is a Key Pair Generation mechanism to create a new RSA key pair of objects using the method described in PKCS#1

This PTK C mechanism has an optional parameter of type CK_ULONG which, if provided, will spcify the size in bits of the random public exponent.

CKM_SECRET_RECOVER_WITH_ATTRIBUTES

The Secret Recovery Mechanism denoted `CKM_SECRET_RECOVER_WITH_ATTRIBUTES` is a derive mechanism to create a new key object by combining two or more shares.

The mechanism has no parameter.

The *C_DeriveKey* parameter *hBaseKey* is the handle of one of the share objects. The mechanism will obtain the CKA_LABEL value from *hBaseKey* and then treat all data objects with the same label as shares.

A template is not required as all the attributes of the object are also recovered from the secret.

Usage Note

To avoid shares getting mixed up between different uses of this mechanism the developer should ensure that data objects with the same label are all from the same secret share batch.

For further information about secure key backup and restoration see the *ProtectToolkit C Administration Manual*.

CKM_SECRET_SHARE_WITH_ATTRIBUTES

The Secret Share Mechanism denoted CKM_SECRET_SHARE_WITH_ATTRIBUTES is a derive mechanism to create M shares of a key such that N shares are required to recover the secret, where N is less than or equal to M.

The mechanism creates a secret value by combining all the attributes of the base key and then shares that secret into M shares.

The algorithm used is according to *A. Shamir - How to Share a Secret, Communications of the ACM vol. 22, no. 11, November 1979, pp. 612-613*

It has a parameter, a CK_SECRET_SHARE_PARAMS, which specifies the number of shares M and the recovery threshold N. See below for the definition.

The mechanism will create M data objects and return the object handle of one of them. It is expected that the data objects would be copied to a smart card token for storage.

The template supplied is used to specify the CKA_LABEL attribute of each new data object. If the CKA_LABEL attribute is not provided in the template then a CKR_TEMPLATE_INCOMPLETE error is returned.

The mechanism contributes the CKA_VALUE attribute of each data object. Any attempt to specify a CKA_VALUE attribute in the template will cause the mechanism to return the error: CKR_TEMPLATE_INCONSISTENT.

The default value of the CKA_TOKEN, CKA_PRIVATE attribute of the new objects is false. The new data objects will have a CKA_SENSITIVE attribute. If the CKA_SENSITIVE attribute of the base key is true then the data objects is sensitive. If the base key is not sensitive then the data objects take the value of CKA_SENSITIVE from the template or it is defaulted to false.

Usage Note

To avoid shares getting mixed up between different uses of this mechanism the developer should ensure that there are no data objects with the same label already on the token before attempting to use this mechanism. If objects are found then these objects should be deleted or a different label chosen.

Security Note

The key to be exported with this mechanism requires the CKA_DERIVE attribute to be true. This has the effect of enabling other key derive mechanisms to be performed with the key. If this is not desired then the CKA_MECHANISM_LIST attribute may be used with the key to restrict its derive operations to this mechanism.

For further information about secure key backup and restoration see the *ProtectToolkit C Administration Manual*.

Secret Share Mechanism Parameter

CK_SECRET_SHARE_PARAMS is used to specify the number of shares M and the recovery threshold N for secret sharing mechanisms. It is defined as follows:

```
typedef struct CK_SECRET_SHARE_PARAMS {

CK_ULONG n;

CK_ULONG m;} CK_SECRET_SHARE_PARAMS;
```

The fields of the structure have the following meanings:
n Number of shares required to recover the secret. Must be at least two and not greater than the number of shares *m* Total number of shares. Must be at least two and not greater than sixty four.

CK_SECRET_SHARE_PARAMS_PTR is a pointer to a CK_SECRET_SHARE_PARAMS.

CKM_SEED_CBC

SEED-CBC, denoted CKM_SEED_CBC, is a mechanism for single and multiple part encryption and decryption, key wrapping and key unwrapping, based on the KISA (Korean Information Security Agency) SEED specification and cipher-block chaining mode.

It has a single parameter; a 16-byte initialization vector.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the CKA_VALUE attribute of the key that is wrapped, padded on the trailing end with up to block size minus one null bytes so that the resulting length is a multiple of the block size. The output data is the same length as the padded input data. It does not wrap the key type, key length, or any other information about the key; the application must convey these separately.

For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the CKA_KEY_TYPE attribute of the template and, if it has one and the key type supports it, the CKA_VALUE_LEN attribute of the template. The mechanism contributes the result as the CKA_VALUE attribute of the new key. Other attributes required by the key type must be specified in the template.

Constraints on key types and the length of data are summarized in the following table.

Table 50 – SEED-CBC: Key and Data Length

Function	Key Type	Input Length	Output Length	Comments
C_Encrypt	CKK_SEED	Multiple of block size	Same as input length	No final part
C_Decrypt	CKK_SEED	Multiple of block size	Same as input length	No final part
C_WrapKey	CKK_SEED	Any	Input length rounded up to multiple of the block size	
C_UnwrapKey	CKK_SEED	Multiple of block size	Determined by type of key being unwrapped or CKA_VALUE_LEN	

CKM_SEED_CBC_PAD

SEED-CBC with PKCS padding, denoted CKM_SEED_CBC_PAD, is a mechanism for single and multiple part encryption and decryption; key wrapping; and key unwrapping, based on the KISA (Korean Information Security Agency) SEED specification, cipher-block chaining mode and the block cipher padding method detailed in PKCS #7.

It has a single parameter; a 16-byte initialization vector.

The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for the CKA_VALUE_LEN attribute.

In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, and DSA private keys.

Constraints on key types and the length of data are summarized in the following table. The data length constraints do not apply to the wrapping and unwrapping of private keys.

Table 241 – SEED-CBC with PKCS Padding: Key and Data Length

Function	Key Tpe	Input Length	Output Length
C_Encrypt	CKK_SEED	Any	This is the input length plus one, rounded up to a multiple of the block size.
C_Decrypt	CKK_SEED	Multiple of block size	Between 1 and block size bytes shorter than input length.
C_WrapKey	CKK_SEED	Any	This is the input length plus one, rounded up to a multiple of the block size.
C_UnwrapKey	CKK_SEED	Multiple of block size	Between 1 and block length bytes shorter than input length.

CKM_SEED_ECB

SEED-ECB, denoted `CKM_SEED_ECB`, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on the KISA (Korean Information Security Agency) SEED specification and electronic codebook mode. It does not have a parameter

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the `CKA_VALUE` attribute of the key that is wrapped, padded on the trailing end with up to block size, minus one null bytes so that the resulting length is a multiple of the block size. The output data is the same length as the padded input data. It does not wrap the key type, key length, or any other information about the key; the application must convey these separately.

For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the `CKA_KEY_TYPE` attribute of the template and, if it has one and the key type supports it, the `CKA_VALUE_LEN` attribute of the template. The mechanism contributes the result as the `CKA_VALUE`attribute of the new key. Other attributes required by the key type must be specified in the template.

Constraints on key types and the length of data are summarized in the following table.

Table 52 – SEED-ECB: Key and Data Length

Function	Key Type	Input Length	Output Length	Comments
C_Encrypt	CKK_SEED	Multiple of block size	Same as input length	No final part
C_Decrypt	CKK_SEED	Multiple of block size	Same as input length	No final part
C_WrapKey	CKK_SEED	Any	Input length rounded up to multiple of block size	
C_UnwrapKey	CKK_SEED	Multiple of block size	Determined by type of key being unwrapped or CKA_VALUE_LEN	

CKM_SEED_ECB_PAD

SEED-ECB with PKCS padding, denoted CKM_SEED_ECB_PAD, is a mechanism for single- and multiple-part encryption and decryption, key wrapping and key unwrapping, based on the KISA (Korean Information Security Agency) SEED specification, electronic code book mode and the block cipher padding method detailed in PKCS #7. It does not have a parameter.

The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for the CKA_VALUE_LEN attribute.

In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, and DSA private keys. The entries in ***Table 53 – SEED-ECB with PKCS Padding: Key and Data Length*** for data length constraints when wrapping and unwrapping keys do not apply to wrapping and unwrapping private keys. Constraints on key types and the length of data are summarized in the following table.

Table 53 – SEED-ECB with PKCS Padding: Key and Data Length

Function	Key Type	Input Length	Output Length
C_Encrypt	CKK_SEED	Any	This is the input length plus one, rounded up to a multiple of the block size.
C_Decrypt	CKK_SEED	Multiple of block size	Between 1 and block size bytes shorter than input length.
C_WrapKey	CKK_SEED	Any	This is the input length plus one, rounded up to a multiple of the block size.
C_UnwrapKey	CKK_SEED	Multiple of block size	Between 1 and block length bytes shorter than input length.

CKM_SEED_KEY_GEN

The SEED key generation mechanism, denoted CKM_SEED_KEY_GEN, is a key generation mechanism for the Korean Information Security Agency's SEED algorithm.

The mechanism does not have a parameter, and it generates SEED keys 16 bytes in length.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, CKA_VALUE_LEN, and CKA_VALUE attributes to the new key. Other attributes supported by the SEED key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or they may be assigned default initial values.

For this mechanism, the ulMinKeySizeand ulMaxKeySizefields of the CK_MECHANISM_INFO structure specify the supported range of SEED key sizes, in bytes, which is 16.

The algorithm block size is 16 bytes.

CKM_SEED_MAC

SEED-MAC, denoted by CKM_SEED_MAC, is a special case of the general-length SEEDMAC mechanism. SEED-MAC always produces and verifies MACs that are eight bytes in length. It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table.

Table 54 – SEED-MAC: Key and Data Length

Function	Key Type	Data Length	Signature Length
C_Sign	CKK_SEED	any	½ block size (8 bytes)
C_Verify	CKK_SEED	any	½ block size (8 bytes)

CKM_SEED_MAC_GENERAL

General-length SEED-MAC, denoted `CKM_SEED_MAC_GENERAL`, is a mechanism for single and multiple part signatures and verification, based on the KISA (Korean Information Security Agency) SEED specification.

It has a single parameter, a `CK_MAC_GENERAL_PARAMS` structure, which specifies the output length desired from the mechanism.

The output bytes from this mechanism are taken from the start of the final SEED cipher block produced in the MACing process.

Constraints on key types and the length of data are summarized in the following table.

Table 55 – General-length SEED-MAC: Key and Data Length

Function	Key Type	Data Length	Signature Length
C_Sign	CKK_SEED	Any	0-block size, as specified in parameters
C_Verify	CKK_SEED	Any	0-block size, as specified in parameters

CKM_SET_ATTRIBUTES

The Set Object Attribute Mechanism denoted CKM_SET_ATTRIBUTES is a TICKET mechanism used to modify the attributes of a key. It does not take a parameter.

The ticket specifies the Digest of the key/object to modify and the new attribute values. The ticket is digitally signed and the certificate used to verify the signature must be contained in the CKA_ADMIN_CERT attribute of the key object being modified.

This mechanism is only used with the CT_PresentTicket command.

CKM_SHA1_RSA PKCS_TIMESTAMP

The PKCS#11 mechanism CKM_SHA1_RSA_PKCS_TIMESTAMP provides time stamping functionality. The supported signing functions are C_Sign_Init and C_Sign. This mechanism supports single and multiple-part digital signatures and verification with message recovery. The mechanism uses the SHA1 hash function to generate the message digest. The mechanism only supports one second granularity in the timestamp although the timestamp format will provide for future sub-second granularity.

A monotonic counter object is used to generate the unique serial number that forms part of the timestamp. The monotonic counter object is automatically created when a token is initialized and exists by default in the Admin Token.

The following structure is used to provide the optional mechanism parameters in the CK_MECHANISM structure. The CK_MECHANISM structure is defined in the *PKCS #11 v2.10: Cryptographic Token Interface Standard, RSA Laboratories December 1999.*

```
typedef struct CK_TIMESTAMP_PARAMS {
CK_BBOOL useMilliseconds;
  CK_TIMESTAMP_FORMAT timestampFormat;
} CK_TIMESTAMP_PARAMS;
```

The "useMilleseconds" parameter specifies whether the timestamp should include millisecond granularity. The default value for this parameter is FALSE. If the mechanism parameters are specified then the useMilliseconds parameter must be set to FALSE as only one-second granularity is provided in the first release of the mechanism's implementation.

The "timeStampFormat" parameter specifies the input/output format of the data to be timestamped. This provides the ability to introduce future support for timestamping protocols such as those defined in RFC3161. The default value for this parameter is CK_TIMESTAMP_FORMAT_PTKC. If the mechanism parameters are specified then the timeStampType parameter must be set to CK_TIMESTAMP_FORMAT_PTKC as only this format is supported in the first release.

For CK_TIMESTAMP_FORMAT_PTKC the mechanism expects the input data to be a stream of bytes for which a message digest must be computed and a timestamp generated according to the format defined below. If mechanism parameters are passed and the two parameters are not set as defined above, the C_SignInit function returns CKR_MECHANISM_PARAM_INVALID.

C_Sign is defined in the PKCS #11 standard as:

```
CK_DEFINE_FUNCTION(CK_RV, C_Sign)(
CK_SESSION_HANDLE hSession,
CK_BYTE_PTR pData,
CK_ULONG ulDataLen,
CK_BYTE_PTR pSignature,
CK_ULONG_PTR pulSignatureLen );
```

The parameter formats are defined in the following tables.

Table 56 – Input format (=pData in C_Sign)

C-Definition	Description
unsigned char Data	Transaction data (variable length), maximum of 3k

Table 57 – Output format (=pSignature in C_Sign)

C-Definition	Contents on Output
Unsigned char serialnumber[20]	This is a unique number for each timestamp, padded with zeroes in a Big Endian 20 byte array. The number is read from the CKH_MONOTONIC_COUNTER hardware feature object on the same token as the signing key. By this read action the value contained by the object is automatically increased by 1.
Unsigned char timestamp[15]	This is the timestamp in the format of GeneralizedTime specified in RFC3161. The syntax is: YYYYMMDDhhmmss[.s...]Z The sub-second component is optional and not supported in the intial release but still defined to ensure backward compatibility in the future.
Unsigned char sign[128]	RSA Signature

NOTE 1: Please see the *PKCS #11 v2.10: Cryptographic Token Interface Standard*, RSA Laboratories December 1999 for a definition of types.

NOTE 2: It is highly recommended that the RFC3161 format timestamp provided by the HSM be stored on the host to allow future independent third party timestamp verification.

The mechanism will perform the following:

- Input data that is provided by the calling host.
- Obtain the time from within the ProtectHost.
- Calculate a signature across the merged input data and time data using PKCS#1 type 01 padding as follows:

  Signature = Sign(SHA1(Data || serialnumber || timestamp)
- Output part of the input data, the time data and the signature.

Verification of the signature can be performed using the CKM_SHA1_RSA_PKCS_TIMESTAMP mechanism with `C_Verify` or `C_VerifyRecover`. The difference between the two functions is that C_Verify calculates the hash but does not return it to the caller where as `C_VerifyRecover()` returns the hash. The following is passed as input data: `<data><serialnumber><timestamp>`

CKM_VISA_CVV

This is a signature generation and verification method. The Card Verification Value signature is generated as specified by VISA.
The mechanism does not have a parameter.
Constraints on key types and the length of data are summarized in the following table:

Table 58 – VISA CVV: Key and Data Length

Function	Key Type	Input Length	Output Length
C_Sign	CKK_DES2	16	2
C_Verify	CKK_DES2	16, 2[2]	N/A

[2] Data length, signature length.

CKM_WRAPKEY_DES3_CBC

The CKM_WRAPKEY_DES3_CBC and CKM_WRAPKEY_DES3_ECB mechanisms are used to wrap a key value plus all of its attributes so that the entire key can be reconstructed without a template at the destination. The key value is encoded and encrypted using CKM_DES3_CBC_PAD and then combined with all other object attributes. The result are then MACed. The wrapping key is supplied as normal to the `C_Wrap` and `C_Unwrap` Cryptoki functions.

The `C_Unwrap` operation will fail with `CKR_SIGNATURE_INVALID` if any of the key's attributes have been tampered with while the key was in transit.

Encoding Format

The encoding is a proprietary encoding where fields are identified by their position (no tags). All fields are preceded by an encoding of the length of the content. The length may be zero indicating an empty field but must always be present. Where the length is zero the content is not present (zero bytes). Where the length is non zero the content has the number of bytes equal to the value of the encoded length. The length is encoded as a 32-bit big-endian binary value and can thus take values from 0 to (2^{32} -1) i.e. around 4 gigabytes.

Definitions

wK	This is the wrapping key under which the subject key is to be wrapped. This key must be valid for the operation Ex.
mK	This is a randomly generated MAC key using CKM_DES2_KEY_GEN. This key is used with Mx.
cK	This is clear encoding of the subject key. For single part symmetric keys, this is just the key value. For compound (e.g., RSA) keys, it is a BER encoding as per PKCS#1.
a	This is the encoded non-sensitive subject key attributes. The attributes are encoded with an attribute header, which is the number of attributes (4 byte), followed by a list of sub encodings which contain the attribute type (4 byte), content length (4 byte), a content presence indicator (1 byte), and the content bytes. The presence indicator allows the content length value to be non-zero, but, where presence indicator = 0, no content bytes are included. If the presence indicator is 1 then the content length must be the number of bytes indicated by the content length field. All numeric values are encoded as big-endian. Note that the sensitive attributes are contained in cK.
E x	This is encryption using CKM_DES3_(ECB/CBC)_PAD with key 'x'.
M x	This is MAC generation using CKM_DES3_MAC_GENERAL (8 byte MAC result) with key 'x'.

A wrapped key using CKM_WRAPKEY_DES3_ECB or CKM_WRAPKEY_DES3_CBC is made up of the following fields:

- ecK the encrypted key value, ecK = E_{wK}(cK).
- a the encoded non-sensitive subject key attributes.
- m a MAC of the key value and attributes, m = M_{mK}(cK + a).
- emK the encrypted MAC key value, emK = E_{wK}(mK).

These fields are then encoded as described above.

E.g. Using CKM_WRAPKEY_DES3_CBC on a Single length DES key, with a Triple DES Wrapping key, produces the encoding:

```
|length | ecK – encrypted key value

00000010 2B847CF929FA2148A0A59BB6D44BBD74

|length | a – encoded non-sensitive attributes

00000120

      0000001900000001000000010101000000020000000101000000000300000
      0501746573740000000106000000010100800001280000000101000000107
      0000000101010000016200000001010180000129000000010101000001700
      0000010101000001040000000101010000010500000001010100000108000
      000101010000010A0000000101010000010300000001010000000163000000
      0101010000000000000040100000004000001000000000401000000130000
      0161000000040100000008800001020000001001323030313131303131323 4
      3530333030000010C0000000101000000010200000000000000110000000
      0000000011100000000000000016500000001010000000164000000010100
      000000000000000000

|length | m – MAC of key value and attributes

00000008 6256751248BFA515

|length | emK – encrypted MAC key value

00000018 2B847CF929FA214837ACF80D3AA9D1470082249D71E053DA
```

CKM_WRAPKEY_DES3_ECB

See the entry for CKM_WRAPKEY_DES3_CBC.

CKM_WRAPKEY_AES_CBC

The CKM_WRAPKEY_AES_CBC mechanism is used to wrap a key value plus all of its attributes so that the entire key can be reconstructed without a template at the destination.

This mechanism is the same as the CKM_WRAPKEY_DES3_CBC mechanism described above but uses only NIST approved cryptographic algorithms and key sizes.

The following fields in the encoding are computed differently to those in CKM_WRAPKEY_DES3_CBC mechanism described above.

mK	This is a randomly generated 256-bit MAC key using CKM_GENERIC_SECRET_KEY_GEN. This key is used with Mx.
E x	This is encryption using CKM_AES_CBC_PAD with key 'x'.
M x	This is MAC generation using CKM_SHA512_HMAC_GENERAL (16 byte MAC result) with key 'x'.

CKM_WRAPKEYBLOB_AES_CBC, CKM_WRAPKEY_DES3_ECB

The CKM_WRAPKEYBLOB_AES_CBC and CKM_WRAPKEYBLOB_DES3_CBC mechanism is used to wrap a private key value using the Microsoft PRIVATEKEYBLOB format.

http://msdn.microsoft.com/en-us/library/cc250013(PROT.13).aspx

The RSA private key is formatted as shown below and then the result is encrypted by CKM_AES_CBC_PAD or CKM_DES3_CBC_PAD:

Header 12 bytes long = 07 02 00 00 00 A4 00 00 52 53 41 32
Bit Length (32 bit LE)
PubExp (32 bit LE)
Modukus (BitLength/8 bytes long LE)
P (BitLength/8 bytes long LE)
Q (BitLength/8 bytes long LE)
Dp (BitLength/8 bytes long LE)
Dq (BitLength/8 bytes long LE)
Iq (BitLength/8 bytes long LE)
D (BitLength/8 bytes long LE)

CKM_XOR_BASE_AND_KEY

XORing key derivation, denoted **CKM_XOR_BASE_AND_KEY**, is a mechanism which provides the capability of deriving a secret key by performing a bit XORing of two existing secret keys. The two keys are specified by handles; the values of the keys specified are XORed together in a buffer to create the value of the new key.

This mechanism takes a parameter, a **CK_OBJECT_HANDLE**. This handle produces the key value information that is XORed with the base key's value information (the base key is the key whose handle is supplied as an argument to **C_DeriveKey**).

For example, if the value of the base key is `0x01234567`, and the value of the other key is `0x89ABCDEF`, then the value of the derived key is taken from a buffer containing the string `0x88888888`.

- If no length or key type is provided in the template, then the key produced by this mechanism is a generic secret key. Its length is equal to the minimum of the lengths of the data and the value of the original key.
- If no key type is provided in the template, but a length is, then the key produced by this mechanism is a generic secret key of the specified length.
- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism is of the type specified in the template. If it doesn't, an error is returned.
- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism is of the specified type and length.
- If a key type is provided in the template the behavior depends on whether the type is identical to the type of the base key. If the base key is of type CKK_GENERIC_SECRET then you can change the type of the new key. Otherwise you can change the type only if the "Pure PKCS11" configuration flag has been set.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key are set properly.

If the requested type of key requires more bytes than are available by taking the shorter of the two key's value, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

- If the base key has its CKA_SENSITIVE attribute set to TRUE, so does the derived key. If not, then the derived key's CKA_SENSITIVE attribute is set either from the supplied template or from a default value.
- Similarly, the derived key's CKA_EXTRACTABLE attribute is set either from the supplied template or else it defaults to the value of the CKA_EXTRACTABLE of the base key.
- The derived key's CKA_ALWAYS_SENSITIVE attribute is set to TRUE if and only if the base key has its CKA_ALWAYS_SENSITIVE attribute set to TRUE.
- Similarly, the derived key's CKA_NEVER_EXTRACTABLE attribute is set to TRUE if and only if the base key has its CKA_NEVER_EXTRACTABLE attribute set to TRUE.

CKM_ZKA_MDC_2_KEY_DERIVATION

This is the ZKA MDC-2 and DES based key derivation mechanism. The algorithm implemented by this mechanism is defined in the ZKA technical appendix, "Technischer Anhang zum Vertrag über die Zulassung als Netzbetreiber im electronic-cash-System der deutschen Kreditwirtschaft" V5.2, section 1.9.2.3, "Generierung kartenindividueller Schlüssel".

It has a parameter, the *derivation data*, which is an arbitrary-length byte array.

This mechanism only operates with the `C_DeriveKey( )` function.

The *derivation data* is digested using the CKM_DES_MDC_2_PAD1 mechanism, and the result is ECB decrypted with the base key. The result is used to make the value of a derived secret key. Only keys of type CKK_DES, CKK_DES2 and CKK_DES3 can be used as the base key for this mechanism. The derived key can have any *key type* with *key length* less than or equal to 16 bytes.

- **If no *key type* and no *length* is provided in the template**, then the key produced by this mechanism is a generic secret key. Its length is 16 bytes (the output size of MDC2).
- **If no *key type* is provided in the template, but a *length* is provided**, then the key produced by this mechanism is a generic secret key of the specified length – created by discarding one or more bytes from the right hand side of the decryption result.
- **If a *key type* is provided in the template, but no *length* is provided**, then that key type must have a well-defined length. If it does, then the key produced by this mechanism is of the type specified in the template. If it doesn't, an error is returned.
- **If both a *key type* and a *length* are provided in the template**, the length must be compatible with that key type. The key produced by this mechanism is of the specified type and length. If the length isn't compatible with the key type, an error is returned.
- If the *derived key type* is CKK_DES, or CKK_DES2, the parity bits of the key are set properly.
- If the *derived key value length* requested is more than 16 bytes, an error is returned.

The following *key sensitivity* and *extractability* rules apply for this mechanism:

- The CKA_SENSITIVE, CKA_EXTRACTABLE and CKA_EXPORTABLE attributes in the template for the new key can be specified to be either TRUE or FALSE. If omitted, these attributes each take on the value of the corresponding attribute of the base key. The default value for the CKA_EXTRACTABLE and CKA_EXPORTABLE attributes is TRUE. The default value of the CKA_SENSITIVE attribute depends on the security flags. If the *No clear Pins* security flag is set, the default value is TRUE; otherwise, it is false.
- If the base key has its CKA_ALWAYS_SENSITIVE attribute set to FALSE, then the derived key will as well. If the base key has its CKA_ALWAYS_SENSITIVE attribute set to TRUE, then the derived key has its CKA_ALWAYS_SENSITIVE attribute set to the same value as its CKA_SENSITIVE attribute.
- If the base key has its CKA_NEVER_EXTRACTABLE attribute set to FALSE, then the derived key will too. If the base key has its CKA_NEVER_EXTRACTABLE attribute set to TRUE, then the derived key has its CKA_NEVER_EXTRACTABLE attribute set to TRUE only if both CKA_EXTRACTABLE and CKA_EXPORTABLE attributes are FALSE. Otherwise, it is set to FALSE.

Vendor-Defined Error Codes

The table below lists the error codes that may be returned from ProtectToolkit C which are Vendor extensions to the PKCS#11 standard.

Table 59 – SafeNet defined Error Codes

Name	Value	Description
CKR_TIME_STAMP	0x80000101	Not used
CKR_ACCESS_DENIED	0x80000102	Attempting to call C_InitToken when HSM configured for "No Clear Pins" Use CT_InitToken instead.
CKR_CRYPTOKI_UNUSABLE	0x80000103	Not used
CKR_ENCODE_ERROR	0x80000104	Template encode/decode error. Usually internal error but may be caused by badly formed function request parameters.
CKR_V_CONFIG	0x80000105	Not used
CKR_SO_NOT_LOGGED_IN	0x80000106	Operation requires session to be in SO RW mode.
CKR_CERT_NOT_VALIDATED	0x80000107	Public key certificate chain not terminated by a TRUSTED certificate.
CKR_PIN_ALREADY_INITIALIZED	0x80000108	Calling C_InitPIN when pin is already initialised. Use C_SetPIN instead.
CKR_REMOTE_SERVER_ERROR	0x8000010A	Not used
CKR_CSA_HW_ERROR	0x8000010B	Not used
CKR_NO_CHALLENGE	0x80000110	Not used
CKR_RESPONSE_INVALID	0x80000111	Failure to disable an FM
CKR_EVENT_LOG_NOT_FULL	0x80000113	Attempting to erase Event log when it is not full.
CKR_OBJECT_READ_ONLY	0x80000114	Attempting to C_DestroyObject with CKA_DELETABLE=TRUE
CKR_TOKEN_READ_ONLY	0x80000115	Not used
CKR_TOKEN_NOT_INITIALIZED	0x80000116	Attempting to Reset a Token that is not initialised
CKR_NOT_ADMIN_TOKEN	0x80000117	Attempting to create an object or write an attribute of an object on a normal token that should only be on an Admin token
CKR_AUTHENTICATION_REQUIRED	0x80000130	Not used
CKR_OPERATION_NOT_PERMITTED	0x80000131	Attempting to generate a timestamp when the RTC is not working or trusted. PKCS#12 import package has more than one private key.
CKR_PKCS12_DECODE	0x80000132	PKCS#12 package corrupt
CKR_PKCS12_UNSUPPORTED_SAFEBAG_TYPE	0x80000133	PKCS#12 package contains unrecognised SAFEBAG
CKR_PKCS12_UNSUPPORTED_PRIVACY_MODE	0x80000134	PKCS#12 package contains unrecognised privacy (public key mode not psupported)
CKR_PKCS12_UNSUPPORTED_INTEGRITY_MODE	0x80000135	PKCS#12 package contains unrecognised integrity (should be MAC)

Name	Value	Description
CKR_KEY_NOT_ACTIVE	0x80000136	Key has exceeded its usage limit or dates.
CKR_ET_NOT_ODD_PARITY	0x80000140	DES key being loaded into HSM has bad parity (should be odd) – fix key or enable "Des Keys Even Parity Allowed" mode (ctconf –fd)
CKR_CANNOT_DERIVE_KEYS	0x80000381	Internal error when establishing a secure messaging connection.
CKR_BAD_REQ_SIGNATURE	0x80000382	Corrupt request to HSM when using secure messaging (network or device driver error)
CKR_BAD_REPLY_SIGNATURE	0x80000383	Corrupt reply from HSM when using secure messaging (network or device driver error)
CKR_SMS_ERROR	0x80000384	General error from secure messaging system – probably caused by HSM failure or network failure.
CKR_BAD_PROTECTION	0x80000385	Cryptoki library has failed to apply proper secure message protection – internal error.
CKR_DEVICE_RESET	0x80000386	HSM has unexpectantly shutdown. Check the event log for errors (ctconf –e)
CKR_NO_SESSION_KEYS	0x80000387	Cryptoki library has failed to establish keys for secure message protection – internal error.
CKR_BAD_REPLY	0x80000388	Reply message from HSM is badly formatted (network or device driver error).
CKR_KEY_ROLLOVER	0x80000389	Secure messaging system has not implemented key rollover protocol properly
CKR_NEED_IV_UPDATE	0x80000310	Secure messaging system has not implemented key rollover protocol properly
CKR_DUPLICATE_IV_FOUND	0x80000311	Not used
CKR_BAD_REQUEST	0x80001001	Badly formed request message (network or device driver error)
CKR_BAD_ATTRIBUTE_PACKING	0x80001002	Cryptoki client has failed to encode attribute list correctly.
CKR_BAD_ATTRIBUTE_COUNT	0x80001003	Cryptoki client has failed to encode attribute list correctly.
CKR_BAD_PARAM_PACKING	0x80001004	Cryptoki client has failed to encode function parameters correctly.
CKR_EXTERN_DCP_ERROR	0x80001386	Not used
CKR_WLD_CONFIG_NOT_FOUND	0x80002001	ET_PTKC_WLD configuration data not consistent
CKR_WLD_CONFIG_ITEM_READ_FAILED	0x80002002	ET_PTKC_WLD configuration data not available
CKR_WLD_CONFIG_NO_TOKEN_LABEL	0x80002003	ET_PTKC_WLD configuration data not formatted correctly
CKR_WLD_CONFIG_TOKEN_LABEL_LEN	0x80002004	ET_PTKC_WLD configuration data not

Name	Value	Description
		formatted correctly
CKR_WLD_CONFIG_TOKEN_SERIAL_NUM_LEN	0x80002005	ET_PTKC_WLD configuration data not formatted correctly
CKR_WLD_CONFIG_SLOT_DESCRIPTION_LEN	0x80002006	ET_PTKC_WLD configuration data not formatted correctly
CKR_WLD_CONFIG_ITEM_FORMAT_INVALID	0x80002007	ET_PTKC_WLD configuration data not formatted correctly
CKR_WLD_LOGIN_CACHE_INCONSISTENT	0x80002010	Internal error in cryptoki library where WLD values are inconsistent.
CKR_HA_MAX_SLOTS_INVALID_LEN	0x80003001	Too many virtual WLD slots are defined
CKR_HA_SESSION_HANDLE_INVALID	0x80003002	Unknown session handle passed to Cryptoki library.
CKR_HA_CANNOT_RECOVER_KEY	0x80003005	HA recovery process needs to create a key but is unable to
CKR_HA_NO_HSM	0x80003006	HA has tried to recover a lost session but no ore working HSMs are available.
CKR_HA_OUT_OF_OBJS	0x80003007	The HA feature has reached its caopacity to manage session objects – too many objects created.

CHAPTER 5
SAMPLE PROGRAMS

Sample programs include a variety of PKCS#11 applications. Unless specifically stated, the source code provided with the ProtectToolkit C SDK product may be modified or incorporated into other programs.

C Samples

Compiling the Sample Programs

The sample programs mentioned above will need to be compiled prior to use.

NOTE: A third-party C software compiler, such as Microsoft Visual C++, must be installed before performing these steps.

To compile under Windows:

1. Set the CPROVDIR environment variable to point to your installation.
   ```
   C:\> set CPROVDIR=C:\program files\safenet\cprov sdk
   ```

2. Use the nmake program to compile the examples.
   ```
   C:\Program files\safenet\Cprov SDK\samples\demo> nmake
   ```

To compile under UNIX:

1. Create a temporary compile directory.
   ```
   % mkdir SafeNet
   ```

2. Copy the sample program and Makefile into that directory.
   ```
   % cp /opt/safenet/protecttoolkit5/ptk/src/demo/* SafeNet
   ```

3. Modify the Makefile to point to your installation directory.
   ```
   CFLAGS=-I/opt/safenet/protecttoolkit5/ptk/include -
   I/opt/safenet/protecttoolkit5/ptk/src/include
   LDFLAGS=-L/opt/safenet/protecttoolkit5/ptk/lib
   ```

4. Use the make program to build the demo.
   ```
   % make
   ```

CTDEMO

This program sets up a 4-token key profile that may be used for an electronic commerce trading application. The token profiles include a sample customer, merchant, bank and certifying authority. The application exchanges public keys between each of the tokens and, where CA mechanism extensions are supported, ProtectToolkit C generates certificates for the public keys.

ProtectToolkit C must be configured to have at least 4 slots/tokens for this demonstration program to operate correctly.

CTDEMO is a console application that takes the following arguments:

ctdemo -s<slotID> -m<modulus size> -q -f –x
where:

-q	Quick. Does not prompt for values but uses defaults.
-f	Force. Does not warn about overwriting token contents.
-m	Specify modulus size.
-s	First slot number to use.
-x	Extended. Creates more keys.

Defaults:

Security Officer (SO) PIN = 9999

Slot	Token Label	PIN
0	Alice	0000
1	NAB	1111
2	Meyer	2222
3	SAFENET	3333

NOTE: This will overwrite the contents of all of the above tokens.

FCRYPT

FCRYPT is a file encryption program that takes a recipient's public key and sender's private key and uses these to encrypt and sign the contents of a file. Random transport keys for triple DES are generated for the bulk file content encryption. Alternately the Password Based Encryption (PBE) variant can be used so that only the password needs to be shared and no public keys/certificates need to be exchanged.

FCRYPT is a console application that takes the following arguments:

Usage

```
fcrypt [-d] [-t] [-o<outfile>] -p<password> infile
fcrypt [-d] [-t] [-o<outfile>] -s<key> -r<key> infile
```

NOTE: Correct usage is to either to provide a pbe-password, or to provide a sender and recipient key.

Options

-h	View help
-d	Decrypt instead of encrypt
-o	Output file name
-p	PBE password
-r	Recipient key name
-s	Sender key name
-t	Report timing info

Key Naming Syntax:

```
<token name>(<user pin>)/<key name>
 for example,  -s"Alice(0000)/Sign"
```

NOTE: The FCRYPT program is also provided as an example tutorial in Chapter 8.

Additional C Sample Programs

There are also a number of additional C sample programs provided. For further information about the functionality of these programs refer to the description provided at the top of the source file for each of them.

Java Samples

Compiling and Running the Sample Programs

The binaries for the sample programs are included in jcprovsamples.jar file. However, in order to use the sources provided, you must compile them first.

NOTE: The JDK 1.2.2 or newer is required to compile these samples.
For best results, ensure that jcprov.jar is in your CLASSPATH environment variable before compiling the applications. Since all the applications are registered under the name space "SafeNet_tech.jcprov.samples", a path that allows this namespace to be used must also be added to the CLASSPATH. If the samples are compiled in their installed locations, the path leading to the "samples" directory in the installation location will allow them to be executed as documented below.

For compiling and running under Windows NT:

- Set the CLASSPATH environment variable to point to jcprov.jar and sample programs' root path.
  ```
  C:\> set "CLASSPATH=C:\program
  files\safenet\cprovsdk\bin\jcprov.jar; C:\program
  files\safenet\cprovsdk\samples"
  ```
- Use javac program to compile the examples.
  ```
  C:\Program
  Files\Safenet\CprovSDK\samples\SafeNet_tech\jcprov\samples> javac
  GetInfo.java
  ```
- Use java program to run samples.
  ```
  C:\Program
  files\safenet\CprovSDK\samples\SafeNet_tech\jcprov\samples>
  javaSafeNet_tech.jcprov.samples.GetInfo -info
  ```

For compiling and running under UNIX:

- Create a temporary compile directory.
  ```
  % mkdir -p SafeNet_tech/jcprov/samples
  ```
- Copy the sample program and Makefile into that directory.
  ```
  % cp
  /opt/safenet/protecttoolkit5/ptk/src/SafeNet_tech/jcprov/samples/*
  SafeNet_tech/jcprov/samples
  ```
- Set the `CLASSPATH` environment variable to point to `jcprov.jar` and sample programs' root path.
  ```
  % export
  CLASSPATH=/opt/safenet/protecttoolkit5/ptk/lib/jcprov.jar:`pwd`
  ```
- Change directory to sample programs' path.
  ```
  % cd SafeNet_tech/jcprov/samples
  ```
- Use `javac` program to compile the examples.
  ```
  % javac GetInfo.java
  ```
- Use java program to run samples.
  ```
  % java SafeNet_tech.jcprov.samples.GetInfo -info
  ```

The Java Classes

DeleteKey

This class demonstrates the deletion of keys.

Usage

```
java SafeNet_tech.jcprov.samples.DeleteKey -keyType <keytype> -keyName
<keyname> [-slot <slotId>] [-password <password>]
```

Options

keytype	One of (des, des2, des3, rsa). The types of keys supported are: • **des** — single DES key • **des2** — double length Triple DES key • **des3** — triple length Triple DES key • **rsa** — RSA Key Pair
keyname	The name (label) of the key to delete.
slotId	The slot containing the token to delete the key from. The default is (0).
password	The user password of the slot. If specified, a private key is deleted.

EccDemo

This class demonstrates the generation of EC keys (prime192v1) and optionally performs sign/verify option with generated keys

Usage

```
java SafeNet_tech.jcprov.samples.EccDemo [-g]  -n<Key label>
```

Options

-g	Generate Key Pair only (do not perform sign/verify)
-n	Labels for key pair

EncDec

This class demonstrates the encryption and decryption operations.

Usage

```
java SafeNet_tech.jcprov.samples.EncDec -keyType <keytype> -keyName
<keyname> [-slot <slotId>] [-password <password>]
```

Options

keytype	One of (des, des2, des3, rsa). The types of keys supported are: • **des** — single DES key • **des2** — double length Triple DES key • **des3** — triple length Triple DES key • **rsa** — RSA Key Pair
keyname	The name (label) of the key to delete.
slotId	The slot containing the token to delete the key from. The default is (0).
password	The user password of the slot. If specified, a private key is used.

EnumAttributes

This class demonstrates the SafeNet extension to enumerate all attributes of an object.

Usage

```
java SafeNet_tech.jcprov.samples.EnumAttributes -name <objectname> [-
slot <slotId>] [password <password>]
```

Options

objectName	The name (label) of the object to enumerate over.
slotId	The slot containing the object. The default is (0).
password	The user password of the slot. If specified, a private object is used.

GenerateKey

This class demonstrates the generation of keys.

Usage

```
java SafeNet_tech.jcprov.samples.GenerateKey -keyType <keytype> -keyName
<keyname> [-slot <slotId>] [-password <password>]
```

Options

keytype	One of (des, des2, des3, rsa). The types of keys supported are: • **des** — single DES key • **des2** — double length Triple DES key • **des3** — triple length Triple DES key • **rsa** — RSA Key Pair • **ec** — EC Key Pair
keyname	The name (label) of the key to delete.
slotId	The slot containing the token to delete the key from. The default is (0).
password	The user password of the slot. If specified, a private key is created.

GetInfo

The class demonstrates the retrieval of Slot and Token Information.

Usage

```
java SafeNet_tech.jcprov.samples.GetInfo (-info, -slot, -token)
[<slotId>]
```

Options

info	Retrieve the General information.
slot	Retrieve the Slot Information of the specified slot.
token	Retrieve the Token Information of the token in the specified slot.
slotId	The related slot ID of the slot or token information to retrieve. The default is (all).

ListObjects

This class demonstrates the listing of Token objects.

Usage

```
java SafeNet_tech.jcprov.samples.ListObjects [-slot <slotId>] [-password
<password>]
```

Options

slotId	The slot containing the token objects to list. The default is (0).
password	The user password of the slot. If specified, private objects are also listed.

ReEncrypt

This class demonstrates re-encryption of variable length data.

Re-encryption is where cipher text (encrypted key or data) is decrypted with one key, and then the resulting plain text is encrypted with another key. Typically you want this operation to occur in such a way as to avoid having the intermediate plain text leaving the security of the adapter.

This is achieved in PKCS#11 via the C_UnwrapKey and C_WrapKey functions. By specifying the intermediate plain text data as a GENERIC_SECRET, SENSITIVE, Session object, you can keep variable length data securely in the adapter. This program assumes that slot 0 exists. All objects generated during program execution are session objects, and as such the contents of the token in slot 0 are not modified.

Usage

```
java SafeNet_tech.jcprov.samples.ReEncrypt
```

Threading

Sample program to show use of different ways to handle multi-threading.

This program initializes the Cryptoki library according to the specified locking model. Then a shared handle to the specified key is created. The specified number of threads is started, where each thread opens a session and then enters a loop which does a triple DES encryption operation using the shared key handle.

It is assumed that the key exists in slot 0, and is a Public Token object.

Usage

```
java ...Threading -numThreads <numthreads> -keyName <keyname> -locking
<lockingmodel> [-v]
```

Options

numthreads	The number of threads to start.
keyname	The name of the Triple DES key to use for encryption operation.
lockingmodel	The locking model, one of : • **None** — No locking performed. Some of the threads should report failures. • **OS** — Use native OS mechanisms to perform locking. • **Functions** — Use Java functions to perform locking.

CHAPTER 6
BEST PRACTICE GUIDELINES

Overview

ProtectToolkit C can be used to add cryptographic services in a standardized way to any application that requires such services. Cryptographic services are required where security policy exists and must be enforced to the full extent possible by state of the art existing technology. Currently cryptographic methods are the only way to assure authenticity, confidentiality and integrity to levels that can be mathematically shown to resist all known attacks for the foreseeable future.

Simplicity is another essential goal since complex systems are extremely difficult to analyze to an extent where all weakness can be found or shown not to exist to a level that is practicable. ProtectToolkit C is a simple and low-level key management and cryptographic service provider and its simplicity should allow it to be used easily to provide the necessary level of cryptographic service.

There are many independent, and sometimes conflicting, goals in the life cycle of developing secure products so this document shall outline the best approach to the use of ProtectToolkit C, always keeping these goals in mind. Above all the developer should always strive to keep implementation simple.

The remainder of this document assumes a basic level of understanding of the ProtectToolkit C product and the PKCS#11 (Cryptoki) system. It refers to the PKCS#11 device as a security module and this may be a stand-alone appliance, or adapter based PKCS#11 security module.

Introduction

The best place to start building a ProtectToolkit C application is with the sample applications that demonstrate how the ProtectToolkit C system should be initialized and used to perform various cryptographic operations. The samples vary quite significantly in complexity. However they are all real working ProtectToolkit C utilities and cover all ProtectToolkit C services.

Security of a system derives mainly from the following areas, confidentiality, authentication, and access control.

Confidentiality

Confidentiality is where there is data that must exist or be transferred through an environment where it may be subject to inspection by an unauthorized person and damages to the owner of the data may result from such inspection. The way to protect confidential data from inspection by unauthorized viewers is simply to encrypt it. Examples of confidential information include corporate or personal data, and cryptographic keys.

Integrity / Authentication

Integrity is the term that applies to the quality of data that it has not been modified since it was last in control of an authorized person. Integrity does not mean that no one should see the data (confidentiality), rather that no unauthorized person should be able to change it without detection. Integrity can be assured by the use of message authentication codes (MAC) that are a cryptographic digest of the message and rely on the knowledge of a secret key.

Access Control

Access control is the method of associating access to certain objects to reliable people who will not misuse those objects, or where misuse can be detected and dealt with by a higher authority. Access control brings accountability for actions to those people that perform those actions. Access control requires authenticating users before the access is granted and there are many methods to do user authentication.

NOTE: It is not necessary to know the value of a secret key to use it to encrypt or sign (MAC) something.

Getting to Know ProtectToolkit C

To become proficient at ProtectToolkit C development it is necessary to understand PKCS#11 and basic security and cryptographic fundamentals. The entire PKCS (Public Key Cryptographic Standard) suite of standards has relevance since PKCS#11 is an API that uses elements of most of the other PKCS standards. The PKCS#11 interface is described in the PKCS#11 definition that is published by RSA Laboratories and is downloadable from their web site. A copy is also included in the ProtectToolkit C SDK package.

You should also refer to Chapter 10, which details some of the many differences between the PKCS#11 standard and the SafeNet ProtectToolkit C implementation.

Another excellent starting point for getting to know ProtectToolkit C is sample PKCS#11 application code, included in the SDK installation, which may be compiled and inspected, or used directly to derive commercial PKCS#11 applications.

Application Implementation Goals

The goals and guidelines listed below are an attempt to relate application development goals to ProtectToolkit C design and implementation strategies. These have been formulated from many years of development experience in using ProtectToolkit C to solve real world application security problems.

Application Security

ProtectToolkit C applications must concern themselves with access control and confidentiality with respect to any keys used by the application. Access control, to limit cryptographic services to those people authorized to perform them and confidentiality to prevent unauthorized disclosure of the keying material.

NOTE: In PKCS#11 there are three classes of users, the public, the token user, and the token security officer (SO). Please refer to the PKCS#11 reference manual and this document for more information regarding the definition of these user classes and their roles and responsibilities.

ProtectToolkit C Security

1. Use one token per application. The tokens are all separately access controlled and should be used to collect all keys that are related to the one application and will normally be used simultaneously within that application. The application should login to the token with the appropriate PIN, use the keys, then logout before terminating. This approach provides a completely separate logical security boundary for each application, ensuring that no cross-application leakage can occur.

2. Use one key for one purpose only. That means that each key in a system should have a clearly defined purpose and not be overloaded with many usages. This limits the damage done by any key that may be exposed and makes misuse of a key less likely.

3. Always mask the input of secret values typed in on a keyboard such as PINs and clear keys. The ProtectToolkit C KMU uses this approach for user PINs and clear key components.

4. Set appropriate access control for keys. This will prevent keys from being used by unauthorized personnel even if the value of the key is safe from exposure. For example, a signature generation key (CKA_SIGN = TRUE) should not be usable for encryption (CKA_ENCRYPT = TRUE). Most keys should be “user” keys (CKA_PRIVATE = TRUE), which means that they are accessible only after a C_Loginhas been performed.

   Keys can be randomly generated with attributes such that they can never be known or extracted outside the token. More often however, keys are backed up shortly after they are generated, then locked into the token with attributes that forbid their extraction. This is often achieved using clearly specified procedures; however the application should assist where possible in enforcing these processes.

5. Use the Key Management Utility (KMU) for key backup and restore purposes.

6. Use the FIPS compliant mode of the device.

ProtectToolkit C Security Caveats

1. CKA_SENSITIVE = FALSE. This attribute setting allows key values to be extracted from the security module using C_GetAttributeValue. Set to TRUE to prevent this form of key value extraction.

2. CKA_EXTRACTABLE = TRUE. This attribute setting allows keys to be wrapped (encrypted) by another key which, if the key is known externally, can be decrypted to obtain the original key value. This is particularly easy because a wrapping key (CKA_WRAP=TRUE) may be created at any time to wrap extractable keys. To prevent this use CKA_EXPORTABLE = TRUE because keys with CKA_EXPORT can be created only by the security officer (SO).

3. Short PINs can be determined by exhaustive search. To prevent this it is advised to use PINs with more than just numeric characters and longer than 6 characters.

4. Any key that has the CKA_MODIFIABLE = TRUE can have most other attributes, particularly key usage attributes, changed. It is best to have persistent keys with this attribute set to FALSE where possible.

5. Once a session is logged on then all sessions of the same application are also logged on and can access all user keys on the token.

6. FIPS operation may be slower and have some interoperability problems for some existing PKCS#11 applications.

7. The PKCS#11 library is a dynamic library that the application attaches to, DLL under Win32 and shared object under UNIX. The library is not separately authenticated by library signing techniques used by other architectures, e.g., JCE and CryptoAPI. Instead the application should rely on the security of the operating system to assure that substitution or tampering with the library has not occurred. It is reasonable to expect modern operating systems to be capable of protecting system files in this way.

Application Usability

It is extremely important to keep usability in mind; otherwise the security requirements become more of an imposition than users are willing to accept and they are more inclined to work around the security. This effect can be seen when users forced to change their passwords too often tend to write them down, or choose simple derivatives of the same password over and over again. Secure systems simply don't work if they are not usable.

ProtectToolkit C Application Usability

1. ProtectToolkit C allows PINs to be non-numeric and can be quite long (up to 32 characters). In fact full 8-bit binary data can be used with the ProtectToolkit C API for PINs but applications tend to use printable characters.

2. When naming keys use the CKA_LABEL attribute and name the key according to both its usage, and origin (or scope), e.g. "KEK - Database" for a key-encrypting-key for use with an applications database. This will make the intent of the key more obvious to both trained and untrained users who may be able to "see" the key but not normally need to use it.

3. Use the token label where possible to find key sets that belong to a particular application rather than slot numbers. It is advisable to use separate tokens in separate slots for separate applications.

4. For server type applications it may not be possible to perform a login every time the system is re-started. This means that keys may be forced to be made non-private so that they are accessible without logging in or the application will have to obtain the login password from some static location – either hard coded or in some environment variable etc depending on the platform.

5. Learn and use the ProtectToolkit C additional libraries (CTEXTRA and CTUTIL) which have been provided to implement common PKCS#11 application features.

ProtectToolkit C Usability Caveats

1. The ProtectToolkit C token browser is a developer's tool and is therefore very low level and can be tricky to use if the user is not familiar with it or with PKCS#11.

2. Watch out for embedded and trailing spaces in token and object label names. Some PKCS#11 implementations do exact matches and will not regard labels with and without the NULL termination as equal.

3. Too many applications only work on slot 0 making interoperability between them on the same platform impossible.

Performance

The product should not perform poorly with security enabled otherwise this will create an incentive to switch it off to meet performance criteria.

ProtectToolkit C Performance

1. In tight loops it is best to remove as much invariant code as possible. This goes for ProtectToolkit C session startup, login, key generation / find, and even the cipher initialization. That way only the code that does the cryptographic operation is in the inner loop.

2. Use session keys if possible since they can be created and destroyed much quicker than token keys. Watch out for object leaks when using session objects however, since they can be very difficult to find because they will not be visible to anything but the application that creates them.

3. Avoid having too many objects on a token, since object lookups are performed by traversing all objects until the correct one is found. Once an object is found it should not need to be searched for again.

4. Multiple adapters (an adapter cluster) can be combined to increase overall throughput where independent streams of cryptographic operations can be allocated to different devices. Key replication is required if cryptographic operations need to be performed by any adapter in the cluster.

ProtectToolkit C Performance Caveats

1. Some operations are limited by some slow operation inside the security module and RSA key generation is a good example of such a slow operation. Other operations may be limited by the speed that data can cross the application – security module interface.

2. Performance figures quoted by some PKCS#11 device vendors may be difficult to obtain in a real world application. Cprov includes a PKCS#11 utility that will measure performance by using only the standard ProtectToolkit C API that any normal application would use. I.e. there is no use of undocumented calls to obtain these performance figures and any application developer should expect to obtain them from any well-written PKCS#11 application.

3. Performance is often not relevant for operations that are not performed in time critical or repetitive situations.

4. FIPS compliant operation may be slower.

Capacity

Tokens have memory that is of two kinds, persistent (token) memory, and session memory. Keys and other objects may be created and managed in either and each has their respective advantages and capacity.

PTK C does not implement a fixed limit on the number of Tokens or the number of objects in one token. Tokens and objects may be created until the persistent memory is fully consumed. However the HSM performance will reduce as the number of slots and objects increases. For all practical purposes the performance will be unacceptably low before the memory is fully consumed.

As a guideline the developer should not design a system that requires more than 50 Tokens or more than 100 objects in any one token.

ProtectToolkit C Capacity Improvement

1. Use externally stored keys encrypted under a key-encrypting-key. That way only the master key-encrypting-key needs to be resident on the device and all working keys are unwrapped (C_UnwrapKey) prior to use and destroyed afterwards.

   NOTE: They can usually be unwrapped as session keys. This technique is common for managing a large set of terminals (EFTPOS or other) that have randomly generated terminal master keys.

2. Use derived keys from a master key stored on the security module. The working key is derived by encrypting some application-supplied data with the master key and using the cipher text data to create a key value. This technique is common for managing a large set of terminals (EFTPOS or other) that have terminal master keys derived from their terminal identifiers. The terminal identifier is usually used as the application supplied data.

3. Backup and restore keys rather than leaving old key sets on-line. This is a simple case of not leaving old key sets, after a key rollover, online for any longer than necessary.

4. Multiple adapters may be used to spread keys across the separate key storages of each device. Cryptographic requests will have to be directed to the adapter that contains the necessary key however.

ProtectToolkit C Capacity Caveats

1. Keys and other objects take up room in proportion to the number and individual sizes of the attributes that make them up. The number of attributes may change for different versions of PKCS#11 also.

2. Memory leaks may happen in both token (persistent) memory and session memory. Detecting and plugging the leaks can be quite difficult. Some development tools (CTCONF) are provided that allow memory usage snapshots to be taken that can help track them down.

3. Low memory conditions may make the device fail in unexpected ways.

Setup / Configuration

An application may take on the task of initialization of the token and key sets or it may presume that they have already been set up for the application to run. The latter is normally the case and ProtectToolkit C includes initialization applications to perform this function.

The ProtectServer configuration and management strategy is based on the Administrator token that is automatically created on all adapters. Please refer to the respective adapter administrator guides for more details.

ProtectToolkit C Setup / Configuration

1. Decide the most appropriate ProtectToolkit C product based on performance, security, price selection criteria. There are many options to choose from here with the ProtectToolkit C product suite and related products so consult brochures and Sales representatives for details.

2. Decide how many tokens should be created for the adapter. Make this decision early since changing the number of tokens / slots is a significant change. One token per application is the normal rule but there may be exceptions.

3. Decide what security settings to enable remembering the FIPS mode is reached by enabling a collection of security settings (see the Administration Manual for details). Some of these settings will impact on performance so will require some consideration before enabling.

4. Decide how to manage the user and security officer (SO) PINs for each token. The PINs protect different services and it is important to note that, when in non-FIPS mode, both keys and cryptographic services can be used when no PIN has been provided.

5. Plan for backup / restore operations to disk or smart card on working key sets. This will influence what key attributes to set for various keys and may require the existence of backup / restore master keys. Refer to the KMU user documentation to obtain more information regarding what backup options are available and how to implement them.

6. Use the KMU for manually setting up key sets, or the CTKMU console application to set them up from a batch file. It is also quite common to write a simple custom application to set up a key set for an application since both KMU and CTKMU use PKCS#11 functions that any application can also call.

ProtectToolkit C Setup/Configuration Caveats

1. The administrator token in ProtectToolkit C V3.x may cause confusion since it looks like and in most respects is a standard PKCS#11 token. There are special objects on this token however and they should not be accessed by any applications other than the ProtectToolkit C supplied tools.

2. Server applications may have to be able to be brought up and running from a re-boot without any assistance, or input (including PINs) from a human operator. This may impact on how PINs are presented to the token for logging in to it.

Maintainability

Security systems must be maintainable so that they can change with changing demand relating to security policy. New algorithms are introduced and others are phased out, for example DES is now giving way to increased use of triple DES. AES shall also start becoming in more common usage.

Many changes in security applications also relate to the increased use of PKI systems with the demands of public key certification and new cryptographic demands related to that.

ProtectToolkit C Maintenance

1. Give keys meaningful names (CKA_LABEL) that relate to the usage as well as their origin.

2. Use supplied PKCS#11 helper functions from CTUTIL library since these are provided to do most common PKCS#11 operations and have been thoroughly tested.

3. Use appropriate key sizes and cryptographic algorithms and allow for key sizes to increase.

4. Write portable code. ProtectToolkit C is available on many platforms from Win32 to UNIX and the best applications are most likely to be ported.

ProtectToolkit C Maintenance Caveats

1. Watch out for spaces and NULL (‘\0’) characters in ProtectToolkit C token and object labels.

2. Attribute template handling code can become very messy and there is a tendency to use global variables. Local variables are better and can be made ‘static’ to avoid stack based initialization compiler warnings.

Debugging

Various development and debugging assistance tools are provided in the ProtectToolkit C SDK including a full software emulation variant of the PKCS#11 library. One other such tool is the ProtectToolkit C logger which is a Cryptoki library replacement that intercepts all ProtectToolkit C calls and reports the call with its arguments to a log file, then completes the call to the real Cryptoki library and reports the call results, return code and arguments to the same log file, before returning to the application.

ProtectToolkit C Debugging Techniques

1. Use the ProtectToolkit C token browser to inspect tokens and keys, and to set them up initially. The token browser can also be used to verify cryptographic operations by hand since just about any ProtectToolkit C function may be called using the browser.

2. Use the software only emulation of PKCS#11 to avoid any issues related to hardware problems including installation difficulties. This also allows effective PKCS#11 development and debugging to be done on a laptop with no PCI bus for expansion cards.

3. Use the ProtectToolkit C logger to obtain PKCS#11 activity traces. This is useful to report problems back to the support staff.

4. Make all keys token keys (CKA_TOKEN = TRUE) rather than session keys. This can help track down object leaks.

5. Make all keys CKA_SENSITIVE=FALSE so that they can be inspected with the token browser at any time.

6. Use the Key Verification Codes (KVC) to check a key's value without having to see the key's value.

7. Give every key a CKA_LABEL whether the applications uses it or not. If there is an object leak where many key objects are being managed then the label may be the only way of tracking it down to the source code that created it.

ProtectToolkit C Debugging Caveats

Remember to switch off all debugging support code once the application is working since some debugging techniques require disabling normal security options. e.g. `CKA_SENSITIVE=FALSE`. This is bad if it gets through to a production system.

Interoperability

PKCS#11 is a standard security module interface defined specifically for removable tokens like smart cards, but also applicable to non-removable devices. Many vendors have adopted this interface so the possibility of any particular application being required to interoperate with more than one PKCS#11 type device is quite high and beneficial to the application developer.

ProtectToolkit C Interoperability

1. Look for PKCS#11 security modules that have high interoperability with standard PKCS#11 applications. Common PKCS#11 applications include Netscape, Entrust, Identrus etc.

2. Test with multiple devices. It is impossible to know for sure that an application is interoperable unless interoperability testing is actually performed.

ProtectToolkit C Interoperability Caveats

1. PKCS#11 is notorious for many implementations that have low interoperability. This is a result of not having a central compliance-testing lab for generic PKCS#11 implementations. There are various application specific compliance test suites that have been used instead.

2. Vendor defined extensions will be present on one vendors implementation but not on all. These should be used only where vendor independence is not an issue, or used where there is no alternative.

Programming in FIPS Mode

When the device is placed into the FIPS compliant mode (see the ProtectToolkit C Administration Manual) each Security Mode flag that is set for FIPS mode, changes the behavior of PKCS#11 and may require the programmer to consider these restrictions when designing their application.

No Public Crypto

This flag is TRUE and each token will have the CKF_LOGIN_REQUIRED flag set and all the cryptographic C_xxxInit functions and key operation functions: C_GenerateKey, C_GenerateKeyPair, C_WrapKey, C_UnwrapKey, C_DeriveKey, C_DigestKey will fail unless the session state is in a User mode (that is, either the USER or SO must be logged in).

If the session state is not in a User mode, any attempt to write to a token will fail (that is, using the functions C_CreateObject, C_DestroyObject and C_SetAttributeValue).

No Clear PINS

This flag is TRUE and the device will not allow clear-text authentication data to pass through the host data port.

When this flag is enabled the C_InitToken function will fail with the error result CKR_ACCESS_DENIED. In order to initialize tokens it is necessary to use the SafeNet extension function CT_InitToken. The SafeNet tools ctconf and gctadmin are aware of this restriction and will automatically use the appropriate function.

The other functions which supply pins to the adapter, namely C_InitPin, C_Login , C_SetPin and CT_InitToken will encrypt the pins before supplying the request to the adapter. The C_CreateObject, C_GenerateKey, C_SeedRandom functions will also be encrypted as they may contain sensitive values. The encryption and decryption is performed by the Secure Messaging System (SMS) and any application will see the request AFTER it has been verified and decrypted by the SMS.

Because the SMS automatically encrypts the PINs there is no impact on the application.

Finally with this flag enabled secret key and private key objects will always have their CKA_SENSITIVE attribute set to true. Any attempt to create a non-sensitive key (that is, set CKA_SENSITIVE=FALSE) or specify CKA_SENSITIVE=FALSE for any object on the device will fail.

An application will fail if it attempts to create, derive or unwrap keys with CKA_SENSITIVE=FALSE.

Authentication Protection

This flag is TRUE and all requests coming from an authenticated user (i.e. a request from a logged in user) must be cryptographically signed.

The signature verification is performed by the SMS and any application will see the request AFTER it has been verified by the SMS. This flag does not impact on an application.

Security Mode Locked

This flag is TRUE and means the settings of the other flags in this mode structure may not be changed (they are Read Only).

This flag may be set to TRUE when FALSE but never FALSE when TRUE. The only way to set this flag to FALSE once it has been set to TRUE is to tamper the device.

Tamper Before Upgrade

This flag is TRUE and all keys, objects and PINs stored in the device's Secure Memory will automatically be erased during any OS Firmware Upgrade, FM Upgrade or FM Disable operation.

Designers should consider their key backup and recovery plans when using FIPS mode.

Only FIPS Approved Algorithms

This flag is TRUE and restricts the PKCS#11 mechanisms available to only the FIPS approved mechanisms. Some algorithms will have their key sizes limited when this flag is true.

Refer to Table 41 for the list of FIPS approved mechanisms.

Key Management

Key management is critical to successful deployment of a secure application. It is important to use the right tools and follow standard techniques wherever possible.

Backup and Restore

The KMU provides key backup and restore facilities for keys that were created with attributes that allow backup operations to be performed on them.

The recommended procedure for key backup is to use the CKA_EXPORT and CKA_EXPORTABLE attributes for the key-encrypting-key and working keys respectively. This is preferred rather than the CKA_WRAP and CKA_EXTRACTABLE attributes that have a security weakness (see above ProtectToolkit C security section) because there is no control on setting the CKA_WRAP attribute. The CKA_EXPORT attribute can be set to TRUE only on a key when the security officer (SO) is logged in to the token. This prevents working key exposures by introducing a known key-encrypting-key to the device. In other words the SO controls the existence of export keys while the user is able to use them but not to create them.

Only keys that have the CKA_EXPORTABLE set to TRUE can be exported by keys that have the CKA_EXPORT attribute set to TRUE. This allows the existence of keys that can never, or no longer be exported from the device.

NOTE: The backup/restore master key-encrypting-key will need to be managed in clear components for split key entry or will have to be backed up with redundancy separately to either disk or smart cards. The redundancy is a defense against one of the master key sets being physically damaged of one of the custodians being unable or unwilling to participate in the restore operation. This is normal in any key-encrypting-key hierarchy for the highest-level keys to be managed by a semi-manual process in control of highly trusted personnel. These highest-level keys are critical to the restore operation and their loss would make restore operations impossible.

Key Replication

Key replication is done for one of two reasons normally:

- Fault tolerant redundancy
- Load balancing

NOTE: The normal key backup with a restore per replication is all that is required to do this job. There is no special key replication procedure. The backup/restore key will need to be present in all devices that the key-set shall be replicated to. For root level keys a semi-manual procedure is required as in key restorations. That is, clear components or Smart Card key injection.

Operator Authentication

Protect toolkit C provides several methods to authenticate the operator.

- The conventional C_Login allows the user Pin to be presented directly to the Token.
- The Pin Challenge feature provides the operator the ability to authenticate to a token by first requesting and then responding to a random challenge. This is a form of bi-directional authentication protocol. The main advantage of this authentication system over the normal PKCS#11 C_Login command is that the clear PIN value never leaves the proximity of the operator. It is particularly useful in the situation where the operator is physically remote from the HSM
- Temporary Pins are an authentication technique that gives the ability to a process to pass user authentication to another process without having to hold a long term sensitive authentication data (such as the PIN) or repeatedly require the operator to authenticate.

A new CKO_HW_FEATURE object called CKH_VD_USER is provided by the firmware to allow the application to obtain the random challenge for either the User Password or SO Password.

The Object has an attribute that an application can read to generate and obtain a random challenge.
A new challenge value will generated each time the attribute is read. A separate Challenge is held for each registered application. The same challenge can be used for User or SO authentication. See CT_GetAuthChallenge function description.

The calling application converts the challenge into a Response by using the following algorithm:

```
Response = SHA-256( challenge | PVC)
Where PVC = LEFT64BIT( SHA1(password | userTypeByte)
```

A host side static library function CT_Gen_Auth_Response is provided in the SDK to assist developers in using this scheme.

The CKH_VD_USER has an attribute that an application can read to generate and obtain a Temporary Pin. Only one SO and one User Temporary pin may exist at any one time in any single Token. Each read from this attribute will generate a new Temporary Pin. See CT_GetTmpPin function description.
Any Temporary Pins in a Token are automatically destroyed when the generating process logs off or is terminated or the HSM has reset – whichever comes first.

Under Cryptoki all authentication of users to the HSM is valid for the calling process only. Each application must authenticate separately. Once a process has authenticated is granted appropriate access to the services of the token.
With PTK C - if a process forks a new process then the new process must authenticate itself - it can not inherit the authentication of the parent.
The Temporary Pin feature is a method where a parent process can pass on its authentication to a child process without having to pass the sensitive pin value.

The Response and Temporary Pin are passed to the HSM using the C_Login function. The Function will be extended such that unused bits in the userType parameter will be set to indicate that a Response value or Temporary PIN is being used instead of the normal password.

The following bits are added to the userType parameter of the C_Login Function to specify the type of authentication required.

```
#define CKF_AUTH_RESPONSE           0x00000100
#define CKF_AUTH_TEMP_PIN           0x00001000
```

Operator Authentication Use Cases

Setup

User sets the User and SO pins in the normal manner (using ctkmu or ctconf tools or other applications)

Programmatic Challenge Response Activation

- Remote client initiates activation by sending a message to the server
- Server Process registers itself to HSMs using C_Initialise
- Server Process opens a session to a Token
- Server Process obtains a Random challenge by calling CT_GetAuthChallenge
- Server Process sends challenge to Remote client
- Client computes the response value using CT_Gen_Auth_Response and returns it to the Server
- Server Process supplies response as PIN value to the C_Login function using a special userType parameter value

Pass Authentication to a New Process

- Server Primary Process authenticates using Programmatic Challenge Response Activation
- Server Primary Process obtains a temporary pin by calling CT_GetTmpPin
- For each spawned process, the Primary Process passes the temporary PIN to it using an appropriate inter process communication method (or by forking).
- New Process registers itself to HSMs using C_Initialise
- New Process opens a session to the Required Token
- New Process authenticates to Token with C_Login function and the temporary pin using a special userType parameter value

Key Usage Limits

Each private key object on a token may have usage limits applied to them by the use of START_DATE, END_DATE, DESTROY_ON_COPY, USAGE_COUNT and USAGE_LIMIT plus the CKA_ADMIN_CERT attributes.

The START_DATE and END_DATE attributes enforce limits on the use of a key based on the date.

The USAGE_COUNT and USAGE_LIMIT attributes enforce limits on the use of a key based on the number of operations of that key. The USAGE_COUNT attribute increases with each use of the key until USAGE_LIMIT is reached. If USAGE_COUNT equals or is greater than USAGE_LIMIT then the key is locked and cannot be used.

In order to stop abuse of the USAGE_COUNT/USAGE_LIMIT controls any Object with a non-empty CKA_USAGE_LIMIT attribute will be automatically deleted after a successful Copy operation. Without this rule a key and its attributes may be copied and therefore the number of operation remaining is automatically doubled.

The START_DATE, END_DATE, USAGE_COUNT and USAGE_LIMIT attributes can be supplied in the template when a key is created or generated or imported. The C_SetAttributeValue command can be used to add these attributes to a key if the object is modifiable. But the C_SetAttributeValue command cannot be used to modify these attributes.

The CKM_SET ATTRIBUTES ticket mechanism is a mechanism which will change the START_DATE, END_DATE, USAGE_COUNT and USAGE_LIMIT attributes of a specified object is used with the CT_PresentTicket function.

Programmatic Use Cases for a Developer

Create Usage Limited Key Object

- Developer uses C_GenerateKeyPair to create a new Key pair. The private key template should include limitation attributes and specify CKA_MODIFIABLE=False.

Set Usage Limits of an Object Directly

- Developer uses CT_SetLimitsAttributes() to set usage limitation attributes. Note the key must have CKA_MODIFIABLE=True.
- Developer sets CKA_MODIFIABLE=False by calling CT_MakeObjectNonModifiable().

Update Usage Limits of an Object Indirectly

- Developer calls `CT_GetObjectDigest` on the remote machine (Recommend use of SHA-256 algorithm).
- Developer sends Object Digest to the Master machine.
- Optionally - on Master machine Developer locates signing key and reads its CKA_SUBJECT_STR and CKA_USAGE_COUNT attributes. The CKA_SUBJECT_STR value can be used as the issuerRDN value to identify the signing key in the certificate. The CKA_USAGE_COUNT attribute can be used as the certificate serial number.
- Developer uses CT_Create_Set_Attributes_Ticket_Info() to create a ticketInfo data block. The CT_SetCKDateStrFromTime() function can help to construct CKA_START_DATE and CKA_END_DATE values.
- Developer uses the signing key to create a signature of the ticketInfo data block. For RSA signing key the CKM_SHA256_RSA_PKCS mechanism is recommended.
- Developer uses CT_Create_Set_Attributes_Ticket() to construct the Ticket data block.
- Developer arranges that the Ticket data block is sent to the remote server machine.
- Developer uses CT_PresentTicket() with CKM_SET_ATTRIBUTES mechanism on remote machine to change limits attributes on target key.

C H A P T E R 7
CTBROWSE – TOKEN BROWSER

Overview

The CTBROWSE utility is a Win32 GUI application. The utility enables you to create tokens and objects that perform simple operations, such as encrypt, decrypt, sign and verify a signature based on the mechanisms provided by the token.

This utility enables you to create/browse a key pair and certificates. By selecting an object you can view its properties. If a certificate object is selected, you can view the structure (ASN.1 format) of the certificate and encode it to various formats such as Base64, DER.

With CTBROWSE you can create and verify a signature based on the signing mechanism.

CTBROWSE is part of the ProtectToolkit C SDK and is installed as part of that product. See the ProtectToolkit C Installation Guide for further information.

Compliance

This application expects PKCS#11 V 2.10 compliant implementation and will use SafeNet extensions (see the next section) if they are available.

PKCS#11 Extensions Used

SafeNet's PKCS#11 implementation provides additional services beyond the standard definition of PKCS#11, particularly in the area of Certificate services. For example:

- Uses non-standard Attribute enumeration extension although this version will fall back to standard methods to enumerate attributes where this extension is not available.
- PKCS#10 and X.509 creation from public key (see Drag and Drop on page 109).
- ASN.1 decoder/dumper (see Attribute Editing).
- Allows use of additional vendor defined mechanisms and extensions to PKCS#11.

See Mechanisms for a table of SafeNet vendor-defined mechanisms and extensions to PKCS#11.

Operation

User Interface

When started, CTBROWSE displays a window with left and right panels (Figure **7**). The left panel shows a representation of slots and tokens; the right panel has service buttons.

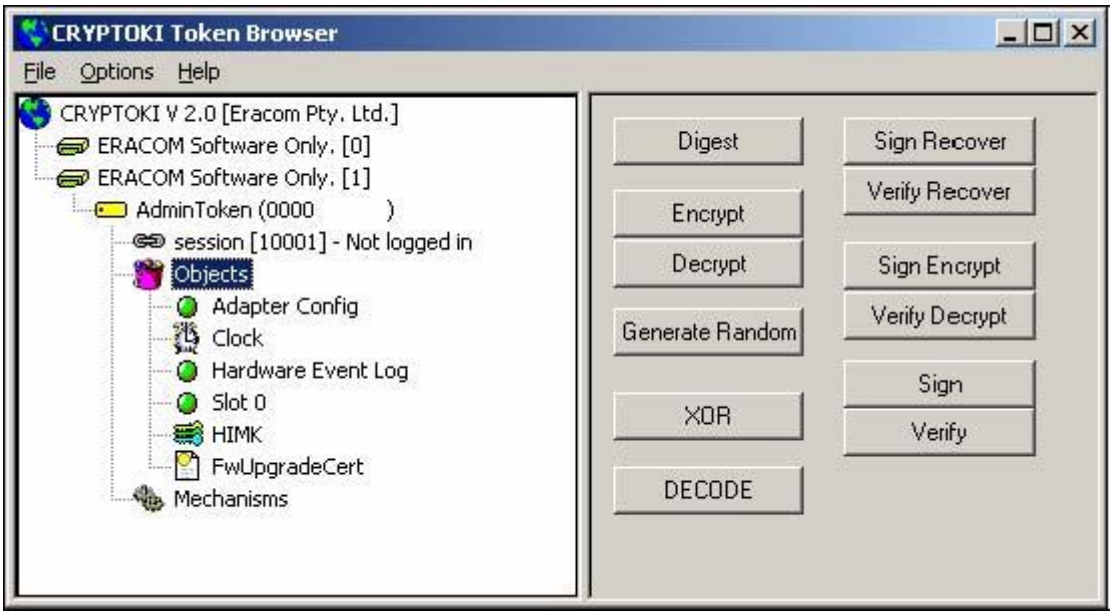

Figure 7 - Token Browser Window

The left panel initially shows only one item, representing the PKCS#11 implementation that CTBROWSE has linked to. This item represents a tree control (Figure 8). Double-clicking on the tree will show the available slots. New slot items may be double-clicked to show tokens in slots.

NOTE: More than one slot containing a token may be available. All slots can be opened and browsed independently.

The left panel shows a typical CTBROWSE session, where the first token has been opened to show all its objects and mechanisms. The numbers in square brackets [ ] show the numeric identifiers (slot identifiers) used to address these items.

The browser can show more than one slot and can be combined with other ProtectToolkit C products, such as the remote client/server, ProtectToolkit C ProtectServer (PCI adapter) and ProtectToolkit C ProtectHost, to allow it to show slots from other PKCS#11 devices including foreign (non-SafeNet) PKCS#11 devices.

Tree View

The next figure shows the hierarchy of the tree. Tree items are identified by labeled icons. The * indicates more than one item at that level of the tree.

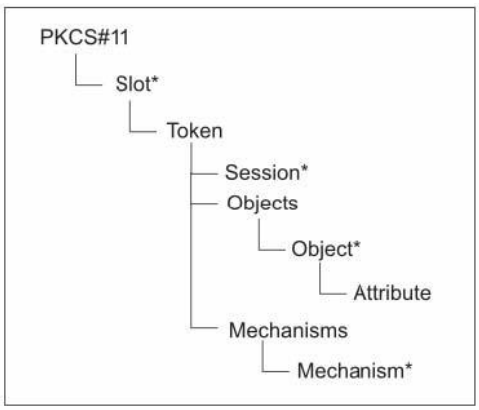

Figure 8 – Tree Hierarchy

Token Management Services

Token management operations on particular tree items are invoked by clicking the right-mouse button while the mouse cursor is over the desired tree item. This action causes a context menu to popup over the selected item.

The next table lists the services available from the popup context menu.

Tree Item	Service	Description
CRYPTOKI	Get info	Shows CRYPTOKI version, manufacturer and description.
Slot	Create token	Allows a token to be created. Note that this uses a non-standard extension to PKCS#11.
	Get info	Shows slot type, manufacturer and description
Token	Init token	Initializes a token and sets the security officer PIN. Note this will erase all the token's contents.
	Open Session	Opens a session to the token.
	Close all Sessions	Closes all open sessions for the token.
	Get info	Shows token type, manufacturer etc.
Session	Close session	Closes the session. Note the session that closes is the one under the mouse when you perform the right click.
	Login	Logs into the token.
	Logout	Logs out from the token.
	Init user PIN	Initializes the user PIN. Note the security officer must be logged in to perform this operation.
	Set PIN	Set the PIN of the current user. This may be the security officer or normal user.
	Get info	Shows the session status and flags.
Objects	Create Object	Allows a new object to be created.
	Create Secret Key	Create a secret key. The key value is entered via the keyboard.
	Unwrap	Unwraps a previously wrapped key.
	Generate Key	Generate a secret key. The key value is randomly generated.
	Generate Key Pair	Generate an asymmetric key pair. The key value is randomly generated.
Object	Destroy	Deletes an object.
	Copy	Makes a copy of an object.
	Set attribute	Sets an attribute for an object.
	Wrap	Wraps a key value.
	Derive key	Derives a shared secret key using Diffie Hellmann. Derives a certificate request, or X.509 certificate.
	Show KVC	Calculates and displays the KVC of the object
	Get info	Shows object size and object handle number.
Attribute	Edit	Allows an attribute's value to be changed, imported or exported. Note that some attributes are defined by PKCS#11 to be unchangeable after being initially set. Attributes can be edited in ASCII or HEX and can also be viewed in Base-64 or decoded ASN.1 syntax for encoded values.
Mechanism	Get info	Shows mechanism info.

Example Service - Generate Key Pair

Generating a key pair is one of the management services available. The Generate Key Pair dialog is opened by right-clicking on an objects tree item in the Token Browser window and choosing Generate Key Pair from the popup context menu.

Figure 9 and Figure 10 show how the labels and fields of the Generate Key Pair dialog box typically change according to the mechanism selected for key pair generation.

NOTE: The check boxes are enabled and disabled according to the selected Mechanism.

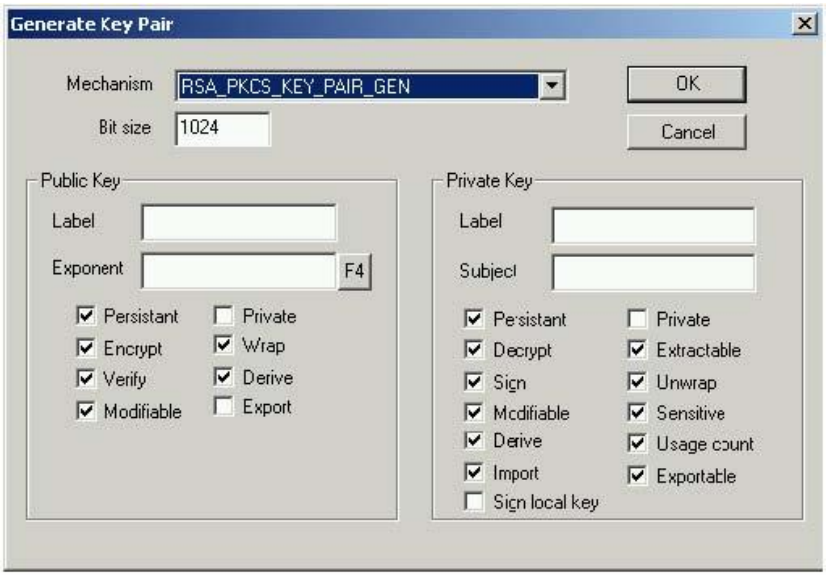

Figure 9 - Generate Key Pair dialog – when RSA mechanism selected

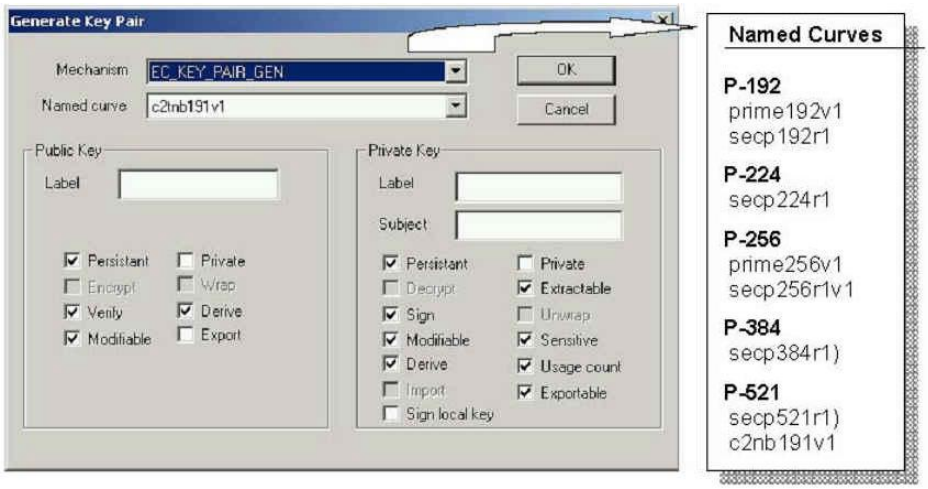

Figure 10 – Generate Key-pair Dialog – Elliptic Curves Selected

Cryptographic Services

The service push-buttons in the right-hand panel of the Token Browser window allow the use of key objects for cryptographic operations such as encryption and digital signatures. To use these services, select the key item from the tree and then click the required button.

Clicking a button opens the associated dialog to guide the user through the operation of that service.

The next figure shows a typical dialog for Encrypt/decrypt and sign/verify services.

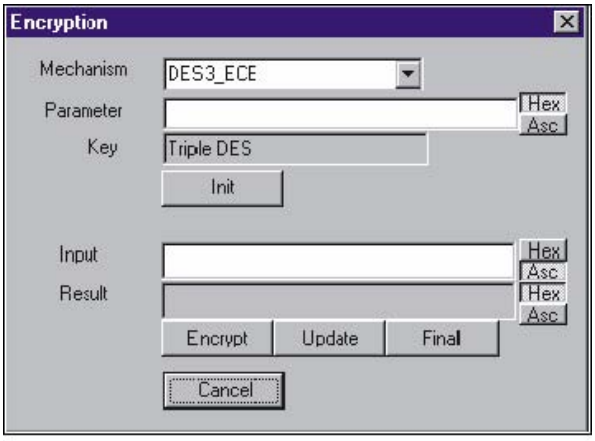

Figure 11 – Encryption Dialog

The key field is taken from the most recently selected key from the tree and the mechanism list shows mechanisms valid for the chosen key. A parameter for the mechanism should be entered if required.

The parameter, input and result fields all allow display of the field in either hexadecimal or ASCII (text) format. The hexadecimal display is useful for the input, or display, of binary data that cannot normally be displayed. Use the [Hex]/[Asc] pushbuttons to select between the two display options.

NOTE: These entry fields support cut-and-paste for easier input.

Operation

Operation of the service requires:

1. Entry of a parameter (if required by the mechanism).
2. Pressing the **Init** button.
3. Entry of an input value.
4. Pressing the **Encrypt** button.

This causes the encryption result to be displayed in the result field.

Drag and Drop

Objects such as key values can be dragged from one token and dropped on to another token, to copy the object.

NOTE: The object must have the CKA_EXTRACTABLE attribute set to TRUE to allow this operation to succeed.

Dropping a public key object onto a private key object will create an X.509 certificate request (PKCS #10 format). This is used to encode a public key together with a subject name (the owner of the key) for distribution to a Certification Authority (CA).

The public key used is from the object being dragged. The subject's name is taken from the CKA_SUBJECT or CKA_SUBJECT_STR attributes of that public key. These attributes were supplied when the key was generated.

NOTE: Certificate Requests should be signed with the private key that matches the public key inside the certificate request. The certificate request is created as an object on the token from where the public key was taken.

The secret key, used to sign the PKCS#10 encoding, may be from another token but should be the secret key that matches the public key being encoded.

Dropping a PKCS#10 certificate request object onto a private key object will create an X.509 certificate. X.509 certificates are the standard way to securely bind a public key together with a subject name (the owner of the key) for public distribution. X.509 certificates are normally signed by a trusted Certification Authority (CA), also known as the certificate's "issuer". The public key and subject name is extracted from the PKCS#10 object (the one being dragged) and the issuer's name is taken from the CKA_SUBJECT or CKA_SUBJECT_STR attributes of the private key that that is used to sign the certificate (the target of the drag).

X.509 certificates also have a serial number that is taken from the CKA_USAGE_COUNT attribute that must also be present on the signing key. The certificate is created as an object on the token from where the certificate was requested. The secret key used to sign the X.509 encoding may be from another token and is normally a highly trusted (CA) signing key.

Using CTBROWSE With Protect Toolkit J

Protect Toolkit J is SafeNet*'s* Java Cryptography Architecture (JCA) and Java Cryptography Extension provider (JCE) software.

CTBROWSE may be used to set up tokens and keys for use with Protect Toolkit J. The tokens and keys that are managed with CTBROWSE are fully compatible and may be utilized by Protect Toolkit J. CTBROWSE may also be used to see and manipulate keys that have been created by Protect Toolkit J. For more information consult the *Key Management* section in the *Protect Toolkit J Reference Manual*.

Please contact SafeNet for further details on its Protect Toolkit J products.

CHAPTER 8
API TUTORIAL: DEVELOPMENT OF A SAMPLE APPLICATION

This tutorial deals with one of the sample applications that are provided with ProtectToolkit C, namely FCrypt.

The FCrypt application enables files to be encrypted for a given recipient and then decrypted by that recipient. Since the encrypted file contains a Message Authentication Code (MAC), the recipient of a document will also be able to verify that the encrypted file was not modified.

In order to follow this example effectively, the reader is strongly encouraged to open or print the source of the application as a reference. The source code for fcrypt can be found in the file "fcrypt.c" within your chosen install directory.

Required Header Files

You will note in the initial code segments that, apart from the standard header files, we include the ProtectToolkit C set of required library files.

```
#include "cryptoki.h"
#include "ctextra.h"
#include "ctutil.h"
#include "chkret.h"
```

Whereas "cryptoki.h" is the required PKCS#11 header, the remainder implement some of the advanced or extended features of the ProtectToolkit C implementation, such as error feedback.

Runtime Switches

We want to develop FCrypt to be able to take a series of command line inputs to allow us to decrypt a message, use password-based encryption (pbe) or to display time information for a cipher operation. With that in mind, the following flags are defined appropriately.

```
static int dflag = 0;
/* 1 - decrypt */static int tflag = 0;
/* 1 - time */static int pflag = 0;
/* 1 - use pbe */
```

Encrypt Functions

For our file encryption and subsequent decryption we define the following two functions.

```
int encryptFile( char * sender, char * recipient, char *ifile,char *
ofile );
int decryptFile( char * sender, char * recipient, char *ifile,char *
ofile );
```

We want the encrypt function to take the public key of the receiving party (recipient), encrypt the data (ifile) with the given key and sign the encrypted data with the senders private key (sender), before outputting and encoding the file to the output file (ofile).

For error handling purposes we define the function as follows:

```
#undef FN
#define FN "encryptFile:"


int encryptFile( char * sender, char * recipient,char * ifile, char *
ofile )
```

We now need to define the required PKCS#11 data types pertaining to the session, slot identification and object handles that we will use for the sender and recipient keys.

```
/* sender slot key session handles */
CK_SLOT_ID hsSlot;
CK_OBJECT_HANDLE hsKey = 0;
CK_SESSION_HANDLE hsSession;
/* recipient slot key session handles */
CK_SLOT_ID hrSlot;
CK_OBJECT_HANDLE hrKey;
CK_SESSION_HANDLE hrSession;
```

In the same manner it is also required that we allocate variables which are used to define the type of mechanism, digest and key information during encryption.

```
CK_RV rv; /* Return Value for PKCS#11 function */
CK_MECHANISM mech; /* Structure for cipher mechanism
*/
CK_BYTE iv[8]; /* Init. Vector used with CBC
encryption */
CK_BYTE digest[80];
CK_SIZE len;
CK_OBJECT_HANDLE hKey; /* random encrypting key */
CK_BYTE wrappedKey[2 * 1024];
CK_SIZE wrappedKeyLen;
CK_BYTE signature[2 * 1024];
unsigned long fileSize;
unsigned long encodedSize;
```

Earlier we said that we wanted to be able to perform password-based encryption via a runtime switch, so accordingly this is the first instance that we check for with our pflag variable.

Our next step is to then define our secret key that we will use to encrypt the data with. The key type to be used is double-length DES. The CK_BBOOL refers to a byte sized Boolean flag that we have defined as either `TRUE`or `FALSE` for easier reference.

CK_ATTRIBUTE is a structure that includes the type, value, and length of an attribute. Since every PKCS#11 key object is required to be assigned certain attributes, this structure is later used during our key derivation and generation to assign those attributes to the key.

```
if ( pflag ) {
/* use PBE to do the encryption */
static CK_OBJECT_CLASS at_class = CKO_SECRET_KEY;
static CK_KEY_TYPE kt = CKK_DES2;
static const CK_BBOOL True = TRUE;
static const CK_BBOOL False = FALSE;
CK_ATTRIBUTE attr[] = {


{CKA_CLASS, &at_class, sizeof(at_class)},{CKA_KEY_TYPE, &kt,
sizeof(at_class)},{CKA_EXTRACTABLE, (void*)&True,
sizeof(True)},{CKA_SENSITIVE, (void*)&False, sizeof(False)},{CKA_DERIVE,
(void*)&True, sizeof(True)}};
```

The `params` variable is defined using the PKCS#11 definition CK_PBE_PARAMS which is a structure that provides all of the necessary information required by the PKCS#11 password based encryption mechanisms.

```
CK_BYTE iv[8];
CK_PBE_PARAMS params;
memset(&params, 0x0, sizeof(CK_PBE_PARAMS));
params.pInitVector = iv;
params.pPassword = sender;
params.passwordLen = strlen(sender);
params.pSalt = NULL;
params.saltLen = 0;
params.iteration = 1;
```

PKCS#11 also uses a structure for defining the mechanism. Within `CK_MECHANISM`we need to specify the mechanism type, a pointer to the parameters we defined earlier and the size of the parameters. The mechanism type we will use is `CKM_PBE_SHA1_DES2_EDE_CBC` that is used for generating a 2-key triple-DES secret key and IV from a password and a salt value by using the SHA-1 digest algorithm and an iteration count.

```
memset(&mech, 0x0, sizeof(CK_MECHANISM));
mech.mechanism = CKM_PBE_SHA1_DES2_EDE_CBC;
mech.pParameter = &params;
mech.parameterLen = sizeof(CK_PBE_PARAMS);
```

We have now set up our required structures and the next logical step is to open a session between the application and a token in a particular slot using the PKCS#11 call `C_OpenSession`. This call requires the slot ID flags which indicate the type of session, an application-defined pointer to be passed to the notification callback; an address of the notification callback function and a pointer to the location that receives the handle for the new session.

```
rv = C_OpenSession(0, CKF_RW_SESSION|CKF_SERIAL_SESSION,NULL,NULL,
&hsSession);
```

```
if ( rv ) return 1;
hrSession = hsSession;
```

Once we have successfully opened a session with the token, we now want to generate the key that we will use to encrypt our input file. The C_GenerateKey function will generate a secret key and thereby create a new key object. This function call requires the session's handle, a pointer to the key generation mechanism, a pointer to the template for the new key, the number of attributes in the template and a pointer to the location that receives the handle of the new key.

The CHECK_RV() function call is part of the ProtectToolkit C extended capability for better error feedback and handling.

```
rv = C_GenerateKey(hsSession, &mech, attr, NUMITEMS(attr),&hKey);
```

```
CHECK_RV(FN "C_GenerateKey:CKM_PBE_SHA1_DES2_EDE_CBC", rv);if ( rv )
return 1;
```

If we are not using the password based encryption switch at program execution, the desired reaction is to perform file encryption using RSA, and hence we will need to generate the secret key value for the operation.

The function FindKeyFromName is part of the ProtectToolkit C CTUTIL library to provide extended functionality. It is used here to locate the keys which are passed into FCRYPTat the command line and return the slot ID, session handle and object handle of those keys.

```
else {
/* use RSA to encrypt the file */
/* locate encrypting key */
rv = FindKeyFromName(sender, CKO_PRIVATE_KEY,

&hsSlot, &hsSession, &hsKey);if ( rv ) {fprintf( stderr, "Unable to
access sender (%s)key\n",
sender );CHECK_RV(FN "FindKeyFromName", rv);if ( rv ) return 1;
}
/* locate signing key */
rv = FindKeyFromName(recipient, CKO_CERTIFICATE,

&hrSlot, &hrSession, &hrKey);if ( rv ) {rv = FindKeyFromName(recipient,
CKO_PUBLIC_KEY,
  &hrSlot, &hrSession, &hrKey);
}
if ( rv ) {

fprintf( stderr, "Unable to access recipient (%s)
key\n", recipient );CHECK_RV(FN "FindKeyFromName", rv);if ( rv ) return
1;}
```

To achieve acceptable performance during file encryption and decryption we need to use a symmetric key cipher such as DES. The DES key we generate for this purpose is to be wrapped with the recipient's RSA key so it can later be unwrapped and used for decryption without the value of the key ever being know.

Rather than simply using the same key for each file encryption, we will generate a random DES key for each encryption of the input file. The mechanism used here is `CKM_DES2_KEY_GEN` that is used for generating double-length DES keys.

The key wrapping is performed with the `C_WrapKey` function that encrypts (wraps) a private or secret key. The function requires the session handle, the wrapping mechanism, the handle of the wrapping key, the handle of the key to be wrapped, a pointer to the location that receives the wrapped key and a pointer to the location that receives the length of the wrapped key.

For the wrapping mechanism we will choose `CKM_RSA_PKCS` that is a multi-purpose mechanism based on the RSA public-key cryptosystem and the block formats defined in PKCS #1. It supports single-part encryption and decryption, single-part signatures and verification with and without message recovery, key wrapping and key unwrapping.

```
/* create a random des key for the encryption */
memset(&mech,0,sizeof(mech));
mech.mechanism = CKM_DES2_KEY_GEN;
/* generate the key */
rv = C_GenerateKey(hrSession, &mech,
wrappedKeyTemp, NUMITEMS(wrappedKeyTemp), &hKey);
CHECK_RV(FN "C_GenerateKey", rv);
if ( rv ) return 1;
/* wrap the encryption key with the recipients public key */
memset(&mech,0,sizeof(mech));
mech.mechanism = CKM_RSA_PKCS;
```

```
memset(wrappedKey,0,sizeof(wrappedKey));

wrappedKeyLen = sizeof(wrappedKey);

rv = C_WrapKey(hrSession, &mech, hrKey, hKey,

wrappedKey, &wrappedKeyLen);

CHECK_RV(FN "C_WrapKey", rv);

if ( rv ) return 1;
```

Now that we have a random secret key to perform the encryption with, we will need to set the required mechanism and parameters prior to encrypting the input file. As a mechanism for the encryption we will choose CKM_DES3_CBC_PAD which is using triple-DES in Cipher Block Chaining mode and PKCS#1 padding.

An application cannot call `C_Encrypt` in a session without having called `C_EncryptInit` first to activate an encryption operation. `C_EncryptInit`requires the session's handle, a pointer to the encryption mechanism and the handle of the encryption key.

In the same manner as we initialized and set up, our digest operation is to be the signature verification to send along to the recipient with the encrypted data. The mechanism used for our digest is SHA-1 that is defined in PKCS#11 terms as `CKM_SHA_1`.

```
/* set up the encryption operation using the random key */
memset(&mech, 0, sizeof(CK_MECHANISM));
mech.mechanism = CKM_DES3_CBC_PAD;
memset(iv, 0, sizeof(iv));
mech.pParameter = iv;
mech.parameterLen = sizeof(iv);
rv = C_EncryptInit(hrSession, &mech, hKey);
CHECK_RV(FN"C_EncryptInit", rv);
if ( rv ) return 1;


/* Set up the digest operation */
memset(&mech, 0, sizeof(CK_MECHANISM));
mech.mechanism = CKM_SHA_1;
rv = C_DigestInit(hrSession, &mech);
CHECK_RV(FN "C_DigestInit", rv);
if ( rv ) return 1;
```

We are now ready to process our input file by encrypting the data, generating the message digest and writing the output to file.

```
/*
** Process the file.
*/
{


FILE * ifp; /* input */
FILE * ofp; /* output */
 CK_SIZE curLen;
 CK_SIZE slen;
unsigned char buffer[10 * 1024];
unsigned char encbuffer[10 * 1024];
unsigned int br; /* bytes read */
unsigned int totbw; /* total bytes written */


/* open input and output file pointers */
ifp = fopen(ifile, "rb");
if ( ifp == NULL ) {


fprintf( stderr, "Cannot open %s for input\n",ifile );
```

```
return -1;
}
ofp = fopen(ofile, "wb");
if ( ofp == NULL ) {

fprintf( stderr, "Cannot open %s for input\n",ofile ); return -1; }
```

If the password based encryption switch wasn't set, the first instance we write to file is the DES secret key wrapped by the recipient's public key.

```
if ( ! pflag ) {/* write the encrypted key to the output file
*/encodedSize = htonl((unsigned long) wrappedKeyLen);br =
fwrite(&encodedSize, 1, sizeof(encodedSize), ofp);br =
fwrite(wrappedKey, 1, (int)wrappedKeyLen, ofp);

}

/* get the file length */

{
struct _stat buf;
 int result;
result = _fstat( _fileno(ifp), &buf );
if( result != 0 ) {


fprintf( stderr, "Cannot get file size for

%s\n",
ofile );
return -1;


}
fileSize = buf.st_size;
/*
fileSize = _filelength(_fileno(ifp));
*/


}
fileSize = (fileSize + 8) & ~7; /* round up for padding */


/* write file size to output file */
encodedSize = htonl(fileSize); /* big endian */
br = fwrite(&encodedSize, 1, sizeof(encodedSize), ofp);
```

Since our mode of encryption is cipher block chaining (CBC) we need to perform our output using four definitive looping steps until our data is processed.

For the digest we use the PKCS#11 function C_Digest_Updatewhich continues a multiple-part message-digesting operation, processing another data part. The function requires the session handle, a pointer to the data part and the length of the data part.

For the encryption we use `C_EncryptUpdate` which continues a multiple-part encryption operation, processing another data part. The function requires the session handle, a pointer to the data part; the length of the data part; a pointer to the location that receives the encrypted data part and a pointer to the location that holds the length in bytes of the encrypted data part.

```
/* read, encrypt, digest and write the cipher text in chunks

*/ totbw = 0;

for ( ;; ) {

br = fread(buffer, 1, sizeof(buffer), ifp);

if ( br == 0 )
```

```
break;
/* digest */
rv = C_DigestUpdate(hrSession, buffer, (CK_SIZE)br); CHECK_RV(FN
"C_DigestUpdate", rv);
if ( rv ) return 1;
/* encrypt */
curLen = sizeof(encbuffer);
rv = C_EncryptUpdate(hrSession, buffer, (CK_SIZE)br,   encbuffer,
&curLen);
CHECK_RV(FN "C_EncryptUpdate", rv);
if ( rv ) return 1;
/* write cipher text */
br = fwrite(encbuffer, 1, (int)curLen, ofp);
totbw += br;}
```

Once all the data has been processed, we need to finalize the encryption and digest operation. To finish the encryption we use the C_EncryptFinal call that finishes a multiple-part encryption operation. The function requires the session handle, a pointer to the location that receives the last encrypted data part, if any, and a pointer to the location that holds the length of the last encrypted data part.

For finalizing the digest we call C_DigestFinal which finishes a multiple-part message-digesting operation, returning the message digest. The function requires the session's handle, a pointer to the location that receives the message digest and a pointer to the location that holds the length of the message digest.

```
/* finish off the encryption */
curLen = sizeof(encbuffer);
rv = C_EncryptFinal(hrSession, encbuffer, &curLen);
CHECK_RV(FN "C_EncryptFinal", rv);
if ( rv ) return 1;
if ( curLen ) {
br = fwrite(encbuffer, 1, (int)curLen, ofp);
totbw += br;}
if ( totbw != fileSize ) {
fprintf( stderr, "size prediction incorrect %ld,
%ld\n", totbw, fileSize );}
/* finish off the digest */
len = sizeof(digest);
rv = C_DigestFinal(hrSession, digest, &len);
CHECK_RV(FN "C_DigestFinal", rv);
if ( rv ) return 1;
```

If the password based encryption flag was set, we use the digest created in the above process as our signature, since there is no recipient key to sign the data with. For our DES encryption we will sign the digest with our recipient's public key.

The function `C_SignInit` is our first call and initializes a signature operation, where the signature is an appendix to the data. The function requires the session's handle, a pointer to the signature mechanism and the handle of the signature key.

We also need to specify a mechanism to use for our signature operation, in this case `CKM_RSA_PKCS`, which is an RSA PKCS #1 mechanism.

The signature generation is performed with the call to `C_Sign` that signs data in a single part, where the signature is an appendix to the data. The function requires the session's handle, a pointer to the data, the length of the data, a pointer to the location that receives the signature, and a pointer to the location that holds the length of the signature.

```
if ( pflag ) {
slen = len;
memcpy(signature, digest, slen);


}
else {/* Set up the signature operation */memset(&mech, 0,
sizeof(CK_MECHANISM));mech.mechanism = CKM_RSA_PKCS;rv =
C_SignInit(hsSession, &mech, hsKey);CHECK_RV(FN "C_SignInit", rv);if (
rv ) return 1;slen = sizeof(signature);rv = C_Sign(hsSession, digest,
len, signature, &slen);CHECK_RV(FN "C_SignInit", rv);if ( rv ) return 1;
}
/* write the signature to the file */
encodedSize = htonl((unsigned long) slen);
br = fwrite(&encodedSize, 1, sizeof(encodedSize), ofp);
br = fwrite(signature, 1, (int)slen, ofp);


/* clean up */
fclose(ifp);
fclose(ofp);
}


C_CloseSession(hrSession);
C_CloseSession(hsSession);


return 0;
}
```

Decrypt Function

For our decryption we want to basically reverse the processes that were covered previously in the encryption section.

Following the initial function setup, we firstly check for our input and output files. Once file existence is established, we test for our password based encryption runtime switch. It can be seen that once again we generate the same secret key from the input password that we will need for the decryption. Since this was a secret key cipher we use the same key for encryption as well as decryption.

```
#undef FN
#define FN "decryptFile:"


int decryptFile( char * sender, char * recipient,char * ifile, char *
ofile )

{
 CK_SLOT_ID hsSlot;
 CK_OBJECT_HANDLE hsKey;
 CK_SESSION_HANDLE hsSession;
 CK_SLOT_ID hrSlot;
 CK_OBJECT_HANDLE hrKey;
 CK_SESSION_HANDLE hrSession;
 CK_RV rv;
 CK_MECHANISM mech;
 CK_BYTE digest[80];
```

```
 CK_SIZE len;
 CK_OBJECT_HANDLE hKey;
CK_BYTE wrappedKey[2 * 1024];
 CK_SIZE wrappedKeyLen;
CK_BYTE signature[2 * 1024];
 CK_BYTE iv[8];
unsigned long encodedSize;
FILE * ifp;
FILE * ofp;
 int br;


ifp = fopen(ifile, "rb");if ( ifp == NULL ) {fprintf( stderr, "Cannot
open %s for input\n",ifile );

return -1;
}
ofp = fopen(ofile, "wb");
if ( ofp == NULL ) {


fprintf( stderr, "Cannot open %s for input\n",ofile ); return -1; }

if ( pflag ) {/* use PBE to do the encryption */static CK_OBJECT_CLASS
at_class = CKO_SECRET_KEY;static CK_KEY_TYPE kt = CKK_DES2; static const
CK_BBOOL True = TRUE;static const CK_BBOOL False = FALSE;CK_ATTRIBUTE
attr[] = {

  {CKA_CLASS, &at_class, sizeof(at_class)},   {CKA_KEY_TYPE, &kt,
sizeof(at_class)},   {CKA_EXTRACTABLE, (void*)&True,

sizeof(True)},
  {CKA_SENSITIVE, (void*)&False,
sizeof(False)},


  {CKA_DERIVE, (void*)&True, sizeof(True)} };CK_BYTE iv[8];
CK_PBE_PARAMS params; memset(&params, 0x0,
sizeof(CK_PBE_PARAMS));params.pInitVector = iv;params.pPassword =
sender;params.passwordLen = strlen(sender);params.pSalt =
NULL;params.saltLen = 0;params.iteration = 1;

memset(&mech, 0x0, sizeof(CK_MECHANISM));mech.mechanism =
CKM_PBE_SHA1_DES2_EDE_CBC;mech.pParameter = &params;mech.parameterLen =
sizeof(CK_PBE_PARAMS);

rv = C_OpenSession(0,
CKF_RW_SESSION|CKF_SERIAL_SESSION, NULL,


NULL, &hsSession);
if ( rv ) return 1;
hrSession = hsSession;


rv = C_GenerateKey(hsSession, &mech, attr,
NUMITEMS(attr),


&hKey);CHECK_RV(FN "C_GenerateKey:CKM_PBE_SHA1_DES2_EDE_CBC", rv);if (
rv ) return 1;

memset(&mech, 0x0, sizeof(CK_MECHANISM));mech.mechanism =
CKM_SHA1_KEY_DERIVATION;

rv = C_DeriveKey(hsSession, &mech, hKey,
attr,NUMITEMS(attr),&hrKey);CHECK_RV(FN
"C_DeriveKey:CKM_SHA1_KEY_DERIVATION", rv); if ( rv ) return 1;}
```

For our public key cipher, we will use the recipient's private RSA key to unwrap the secret DES key contained in the input file. The DES key will then be used to decrypt the file.

The PKCS#11 function C_UnwrapKey is used to decrypt (unwrap) a wrapped key, creating a new private key or secret key object. This function requires the session handle, a pointer to the unwrapping mechanism, the handle of the unwrapping key, a pointer to the wrapped key, the length of the wrapped key, a pointer to the template for the new key, the number of attributes in the template, and a pointer to the location that receives the handle of the recovered key.

```
else {
/* decrypting */
rv = FindKeyFromName(sender, CKO_CERTIFICATE,


&hsSlot, &hsSession, &hsKey);if ( rv ) {rv = FindKeyFromName(sender,
CKO_PUBLIC_KEY,

  &hsSlot, &hsSession, &hsKey);
}
if ( rv ) {


fprintf( stderr, "Unable to access sender (%s)key\n",

sender );CHECK_RV(FN "FindKeyFromName", rv);if ( rv ) return 1;

}rv = FindKeyFromName(recipient, CKO_PRIVATE_KEY,&hrSlot, &hrSession,
&hrKey);if ( rv ) {fprintf( stderr, "Unable to access recipient (%s)

key\n", recipient );CHECK_RV(FN "FindKeyFromName", rv);if ( rv ) return
1;}

/* read the encrypted key to the file */br = fread(&encodedSize, 1,
sizeof(encodedSize), ifp);wrappedKeyLen = (CK_SIZE) ntohl((unsigned
long)

encodedSize);br = fread(wrappedKey, 1, (int)wrappedKeyLen, ifp);

/* unwrap decryption key with the recipients private key

*/
memset(&mech,0,sizeof(mech));
mech.mechanism = CKM_RSA_PKCS;
rv = C_UnwrapKey(hrSession, &mech, hrKey,


  wrappedKey, wrappedKeyLen,   wrappedKeyTemp, NUMITEMS(wrappedKeyTemp),
&hKey );

CHECK_RV(FN "C_UnwrapKey", rv);
if ( rv ) return 1;
}
```

Now that we have recovered the decryption key, we perform our initialization in exactly the same manner as for our encryption, but using the function `C_DecryptInit`. The digest is calculated in the same manner used for the encryption.

For the file decryption we are using the functions `C_DecryptUpdate`and `C_DecryptFinal` which take the same parameters as their encrypt counterparts.

```
/* set up the decryption operation using the random key */
memset(&mech, 0, sizeof(CK_MECHANISM));
mech.mechanism = CKM_DES3_CBC_PAD;
memset(iv, 0, sizeof(iv));
mech.pParameter = iv;
mech.parameterLen = sizeof(iv);
rv = C_DecryptInit(hrSession, &mech, hKey);
CHECK_RV(FN"C_EncryptInit", rv);
if ( rv ) return 1;


/* Set up the digest operation */
memset(&mech, 0, sizeof(CK_MECHANISM));
mech.mechanism = CKM_SHA_1;
rv = C_DigestInit(hrSession, &mech);
CHECK_RV(FN "C_DigestInit", rv);
if ( rv ) return 1;


{
 CK_SIZE curLen;
 CK_SIZE slen;
unsigned char buffer[10 * 1024];
unsigned char decbuffer[10 * 1024];
unsigned int br;


br = fread(&encodedSize, 1, sizeof(encodedSize), ifp);
encodedSize = htonl(encodedSize);
for ( ;encodedSize > 0; ) {


br = sizeof(buffer);
if ( encodedSize < br )


br = (unsigned int)encodedSize;br = fread(buffer, 1, br,
ifp);encodedSize -= br;if ( br ) {

  curLen = sizeof(decbuffer); rv = C_DecryptUpdate(hrSession,
buffer,(CK_SIZE) br,

   decbuffer, &curLen);   CHECK_RV(FN "C_DecryptUpdate", rv); if ( rv )
return 1;rv = C_DigestUpdate(hrSession, decbuffer,

curLen);  CHECK_RV(FN "C_DigestUpdate", rv); if ( rv ) return 1;br =
fwrite(decbuffer, 1, (unsigned

int)curLen,
ofp);


}}curLen = sizeof(decbuffer);rv = C_DecryptFinal(hrSession, decbuffer,
&curLen);CHECK_RV(FN "C_DecryptFinal", rv);

if ( rv ) return 1;if ( curLen ) {br = fwrite(decbuffer, 1, (unsigned
int)curLen,

ofp); rv = C_DigestUpdate(hrSession, decbuffer, curLen);CHECK_RV(FN
"C_DigestUpdate", rv);

}
len = sizeof(digest);
```

```
rv = C_DigestFinal(hrSession, digest, &len);
CHECK_RV(FN "C_DigestFinal", rv);
if ( rv ) return 1;
```

The final act to perform is to verify the signature contained in the data file. Since the signature is identical to the digest when using the password based encryption option, it is a simple matter of comparing the two. For our DES encryption on the other hand, we need to verify the signature against the sender's public key.

To perform this we start by calling C_VerifyInit that initializes a verification operation, where the signature is an appendix to the data. This function requires the session's handle, a pointer to the structure that specifies the verification mechanism and the handle of the verification key.

```
/* read the signature from the file */br = fread(&encodedSize, 1,
sizeof(encodedSize), ifp);slen = (CK_SIZE) ntohl((unsigned long)
encodedSize);br = fread(signature, 1, (unsigned int)slen, ifp);

if ( pflag ) {

if ( memcmp(digest, signature, len) ) {fprintf( stderr, "Verify
failed\n" );return 1;

}
}
else {


/* Set up the signature verify operation */
memset(&mech, 0, sizeof(CK_MECHANISM));
mech.mechanism = CKM_RSA_PKCS;
rv = C_VerifyInit(hsSession, &mech, hsKey);
CHECK_RV(FN "C_VerifyInit", rv);
if ( rv ) return 1;
rv = C_Verify(hsSession, digest, len, signature, slen);
if ( rv ) {


C_ErrorString(rv,ErrorString,sizeof(ErrorString));fprintf( stderr,
"Verify failed 0x%x, %s\n", rv,   ErrorString ); }}

/* clean up */
fclose(ifp);
fclose(ofp);


}

C_CloseSession(hrSession);
C_CloseSession(hsSession);


return (int)rv;
}
```

FCRYPT Usage

When no command line inputs are received by the application, it can be useful to show the required inputs on screen in a help context.

```
void usage(void){ printf( "usage fcrypt -d [-s<sender>] [-r<recipient>]

[-o<output file>] <input file>\n" );printf( " or\n" );printf( "usage
fcrypt -d [-p<password>] [-o<outputfile>]

<input file>\n" );printf( " -d decrypt\n" );printf( " -p PBE password\n"
);printf( " -s Sender name\n" );printf( " -r Recipient name\n" );printf(
" -o output file name\n" );printf( " -t Report timing info\n" );printf(
"\nKey naming syntax :\n");printf( " <token name>(<user pin>)/<key
name>\n" );printf( " for example, -sAlice(0000)/Sign\n" );}
```

Wrapped Encryption Key Template

The DES encryption key that we wrap with the user RSA key will need to have its attributes specified within a template as follows.

```
/* Wrapped encryption key template */static char True = TRUE;static
CK_OBJECT_CLASS Class = CKO_SECRET_KEY;static CK_KEY_TYPE Kt =
CKK_DES2;static CK_ATTRIBUTE wrappedKeyTemp[] = {

{CKA_CLASS, &Class, sizeof(Class)},{CKA_KEY_TYPE, &Kt,
sizeof(Kt)},{CKA_EXTRACTABLE, &True, 1},{CKA_ENCRYPT, &True, 1},};
```

Assembling the Application

Now bring all the required components for the FCRYPT application together in the main application body.

```
#undef FN

#define FN "main:"

int main(int argc, char ** argv)

{  CK_RV rv; int err = 0;char * arg;char * sender = NULL; /* provides
signing key */char * recipient = NULL; /* provides encryption key */char
* ofile = "file.enc"; /* default output file name

*/ printf( "Cryptoki File Encryption $Revision: 1.1 $\n" );printf(
"Copyright (c) SafeNet, Inc 1999-2006\n" );
```

The first call within a PKCS#11 application must be C_Initialize which initializes the PKCS#11 library. The function takes as an argument either value NULL_PTR or points to a CK_C_INITIALIZE_ARGS structure containing information on how the library should deal with multi-threaded access – for the ProtectToolkit C product no threading information is required so a NULL_PTR is used as the argument.
The function call to CT_ErrorString is part of the ProtectToolkit C extended capability within CTUTIL.H and converts a PKCS#11 error code into a printable string.

```
/* This must be the first PKCS#11 call made */
rv = C_Initialize(NULL_PTR);
if ( rv ) {

C_ErrorString(rv,ErrorString,sizeof(ErrorString));fprintf(stderr,
"C_Initialize error %x, %s\n", rv,ErrorString);}
```

Since two versions of PKCS#11 are supported by SafeNet that are incompatible to one another, the `CheckCryptokiVersion` function is called to ensure that an application compiled for V1.0 compliance is not going to fail if it links against a V 2 compliant DLL and vice-versa. This function is part of the extended ProtectToolkit C functionality within CTUTIL.H and ensures that the version of PKCS#11 is correct.

```
/* Check PKCS#11 version */
rv = CheckCryptokiVersion();


if ( rv ) {printf( "Incompatible PKCS#11 version (0x%x)\n", rv ); return
-1;

}

/* process command line arguments */
for ( argv++; (arg = *argv) != NULL; argv++ ) {


if ( arg[0] == '-' || arg[0] == '/' ) {
switch( arg[1] ) {
case 'd':


dflag = 1;break;

case 't':
  tflag = 1;
break;


case 'o':
  ofile = arg+2;
break;


case 's':
  sender = arg+2;
break;


case 'r':
  recipient = arg+2;
break;


case 'p': recipient = sender = arg+2;  pflag = 1; break;

default:
usage();
  return 1;


}
}
 else {


time_t now, t1, t2; /* we will time the     operation */

if ( sender == NULL || recipient == NULL ) {usage();  return 2;

}

if ( tflag ) {/* Mark the time now */for ( t1 = now = time(NULL); now ==
t1; )

   t1 = time(NULL);

}
```

```
/* process the file */if ( dflag )err = decryptFile( sender, recipient,
arg,ofile );else err = encryptFile( sender, recipient, arg,ofile );

/* report error or timing */if ( err ) {fprintf(stderr, "Error
%scrypting file

    %s\n", dflag?"de":"en", arg ); }else if ( tflag ) {

  t2 = time(NULL);
  printf("%d seconds\n", t2-t1);
}
}
}


/* shut down PKCS#11 operations */
```

When the application is done using PKCS#11, it calls the PKCS#11 function `C_Finalize` and ceases to be a PKCS#11 application. It should be the last PKCS#11 call made by an application. The parameter is reserved for future versions and should be set to NULL_PTR.

```
rv = C_Finalize(NULL_PTR);

if ( rv )
{C_ErrorString(rv,ErrorString,sizeof(ErrorString));fprintf(stderr,
"C_Finalize error %x, %s\n", rv,

ErrorString);
}

return err;
```

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 9
PKCS#11 LOGGER LIBRARY

Overview

The logger library produces a log of all PKCS#11 function calls called by an application. It is a useful tool for debugging applications that are developed using the ProtectToolkit C API.

This library can be used with ProtectToolkit C in any of the three operating modes; hardware, client/server or software only.

Logger Architecture and Functionality

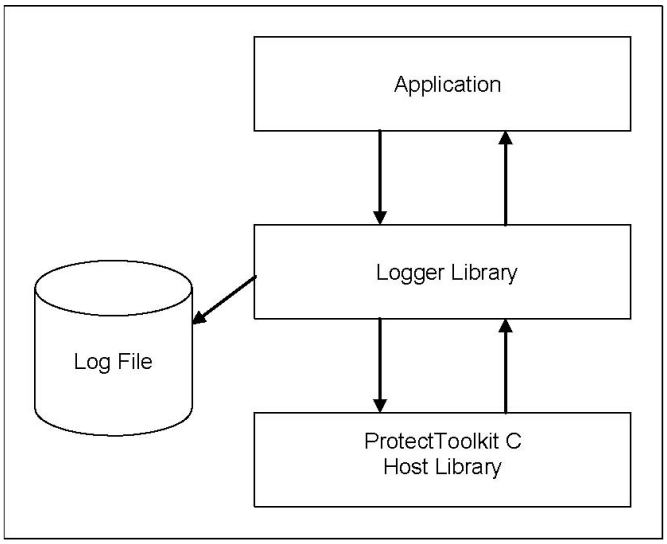

Figure 12 – PKCS#11 Logger Architecture Model

The logger is interposed between the application and the ProtectToolkit C host library. There, it intercepts PKCS#11 function calls and responses. Details are logged to the log file before the messages are passed through to their intended destination.

For each PKCS#11 call, the logger creates an entry in the log file. By default, these entries contain the following details:

- the calling process ID (PID)
- the thread ID (TID)
- the date and time of the call
- all numeric data
- buffer addresses
- contents of buffer addresses at the input and output of functions (excluding PIN values)

Optionally, the logger may be configured to:

- return the PIN values used to login to tokens that are provided to the C_Login function
- remove any or all of the following from the output:
- the calling process ID (PID)
- the thread ID (TID)
- the date and time of the call
- contents of buffer addresses at the input and output of functions

Logger Setup

As discussed above, the logger logs information passing between an application and the ProtectToolkit C host library to a log file. So that this will occur, the following configuration steps must be carried out before starting the application.

1. Activate logging by setting up redirection of ProtectToolkit C host library calls sent from the application so that they are instead delivered to the logger.

2. Store the name of the ProtectToolkit C host library file and the path to it for use by the logger when forwarding the redirected calls that it receives on to their intended destination.

   In addition, you may, if required:

3. Change the name and location of the log file used by the logger to record information, from the default values that will otherwise apply.

4. Change the amount of detail that is recorded by the logger, from the default settings.

Each of these steps is covered in detail in the sections that follow. Once they have been carried out, the logger is active whenever the application is running. To deactivate the logger see the *Deactivating Logger Operation* section below.

Activating Logging

Logging is activated by setting up redirection of ProtectToolkit C host library calls sent from the application so that they are instead delivered to the logger. The method for doing this differs between Windows and UNIX systems. To activate logging consult the section below that covers your operating system.

Windows Systems

To activate logging on a Windows based system ProtectToolkit C host library calls are redirected to the logger by replacing the path to the ProtectToolkit C host library (Cryptoki provider) that was added to the *Path* environment variable during installation, with the path to the logger. The ProtectToolkit C host library and the logger are both named *cryptoki.dll* so the application does not detect any difference and is unaffected by this change.

The path to the logger that must replace the host library path is:

<installation directory>\bin\logger

For example, if the installation path is:

C:\Program files\Safenet\ProtectToolkit C SDK\bin\hsm

Replace it with:

C:\Program Files\Safenet\ProtectToolkit C SDK\bin\logger

To access the *Path* environment variable for editing, follow standard procedure for your system. Typically, the following steps are followed:

1. Right click **My Computer** on the desktop and select **Properties**.

2. In the *System Properties* dialog box select the **Advanced** tab and then select the **Environment Variables** button.

3. In the *Environment Variables* dialog box, locate and select the **Path** variable under **System Variables** and select the **Edit** button.

4. In the *Edit System Variable* dialog box, make the change as outlined above to the **Variable Value** and click the **OK** button to action this change and close the dialog box. Close all other dialog boxes to complete the operation.

UNIX Systems

To activate logging on a UNIX based system, ProtectToolkit C host library calls are redirected to the logger by:

- re-assigning the *libcryptoki.so* (*libcryptoki.sl* for HP-UX on PA-RISC, *libcryptoki.a* for AIX) symbolic link from the ProtectToolkit C host library (Cryptoki provider) that was set up during installation to the logger shared library *liblogger.so (liblogger.sl* for HP-UX on PA-RISC, *liblogger.a* for AIX).
- including the logger library in the LD_LIBRARY_PATH (SHLIB_PATH for HP-UX on PA-RISC, LIBPATH on AIX) environment variable.

The application does not detect any difference and is unaffected by this change.

For example, use the following commands to re-assign the *libcryptoki.so* symbolic link: # cd /opt/safenet/protecttoolkit5/ptk/lib # ln –sf liblogger.so libcryptoki.so

Storing ProtectToolkit C Host Library File Details

To store the name of the ProtectToolkit C host library file and the path to it for use by the logger when forwarding redirected calls, create the configuration item:

ET_PTKC_LOGGER_PKCS11LIB

and set its value to that of the full path required. For example: "*C:\Program Files\Safenet\ProtectToolkit C SDK\bin\hsm\cryptoki.dll*" should be added for Windows Systems.

This change can be made at the temporary, user or system levels on both UNIX and Windows platforms. Refer to the Configuration Items section in the *ProtectToolkit C Administration Manual* for further details on how to go about this if required.

NOTE: There are no default values for this item so this step must be completed, otherwise calls cannot be forwarded and the system will fail.

Storing Log File Details

By default log entries are written to a text file named *ctlog.log*. The full path is:

- *\ctlog.log on* Windows systems or
- *$HOME/ctlog.log* on UNIX systems

To change the file name and or location to something other than the default, create the configuration item, ET_PTKC_LOGGER_FILE, and set its value to that of the full path required.

This change can be made at the temporary, user or system levels on both UNIX and Windows platforms. Refer to the Configuration Items section in the *ProtectToolkit C Administration Manual* for further details on how to go about this if required.

Changing Detail Recorded by the Logger

The table below lists the configuration items which can be used to control the level of detail recorded by the logger when active. In the table, the meaning of each configuration item is given along with the default values that apply in the absence of each particular configuration item.

To change the level of detail recorded, override any of the default values shown. To do this, create the corresponding configuration item and set its value to either *TRUE* or *FALSE* as required.

The changes can be made at the temporary, user or system levels on both UNIX and Windows platforms. Refer to the *Configuration Items* section in the *ProtectToolkit C Administration Manual* for further details on how to go about this if required.

Configuration Item	Meaning
ET_PTKC_LOGGER_LOGPID	If TRUE, the calling process ID (PID) is included in log messages. Default=TRUE
ET_PTKC_LOGGER_LOGTID	If TRUE, the thread ID (TID) is included in log messages. Default=TRUE
ET_PTKC_LOGGER_LOGTIME	If TRUE, the date and time of each message is included in the log. Default=TRUE
ET_PTKC_LOGGER_LOGMEM	If TRUE, all numeric data, buffer addresses and the contents of buffer addresses at the input and output of functions (excluding PIN values)is included in log messages. If FALSE then the contents of buffer addresses at the input and output of functions is omitted. Numeric data and buffer addresses are retained. Default=TRUE
ET_PTKC_LOGGER_LOGPIN	If TRUE, the PIN values passed to C_Login, that are used to login to tokens, are included in log messages. Default=FALSE

Deactivating Logger Operation

To deactivate the logger the steps taken under *Activating Logging* must be reversed. For more information consult the section below for your operating system.

Windows Systems

The path to the logger added to the `PATH` environment variable must be replaced by the path to the ProtectToolkit C host library required.

For example, if ProtectToolkit C is being used in hardware mode in conjunction with a ProtectServer adapter and the path to the logger is:

```
C:\Program Files\Safenet\ProtectToolkit C SDK\bin\logger In the PATH
replace: C:\Program Files\Safenet\ProtectToolkit C SDK\bin\logger
```

with:

```
C:\Program Files\Safenet\ProtectToolkit C SDK\bin\hsm
```

UNIX Systems

The symbolic link `libcryptoki.so` (`libcryptoki.sl` for HP-UX on PA-RISC, `libcryptoki.a f`or AIX) must be re-assigned to the ProtectToolkit C host library required.

For example, if ProtectToolkit C is being used in hardware or client/server mode, the commands to use would be:

```
# cd /opt/safenet/protecttoolkit5/ptk/lib# ln –sf liblogger.so
libcthsm.so
```

In software-only mode, use the following commands:

```
# cd /opt/safenet/protecttoolkit5/ptk/lib# ln –sf liblogger.so
libctsw.so
```

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 10
PKCS#11 COMMAND REFERENCE

General Purpose Functions

C_Initialize

Synopsis

```
C_Initialize(
CK_VOID_PTR pInitArgs);
```

Description

C_INITIALIZE initializes the Cryptoki library.

The *pInitArgs* either has the value NULL_PTR or points to a CK_C_INITIALIZE_ARGS structure containing information on how the library should deal with multi-threaded access.

If the system is currently uninitialized this function will perform a full initialization. This means that any configuration changes since the last full initialization will now take effect. If the system is already initialized this function will simply prepare it for the new application.

Operation in WLD Mode

When ProtectToolkit is configured to operate in WLD mode and C_INITIALIZE () is invoked, the user slots that are associated with WLD slots are interrogated to assess their availability. User slots are defined as associated with a WLD slot when they contain a token with a token label that matches that of the WLD slot.

If, for every WLD slot, there are no associated user slots available, the error CKR_TOKEN_NOT_PRESENT is returned. If, however, at least one associated user slot is available for at least one WLD slot the error CKR_TOKEN_NOT_PRESENT will not be returned.

NOTE: The token labels for WLD slots are defined in the WLD environment variables ET_PTKC_WLD_SLOT_n. Refer to the ProtectToolkit C Administration Manual for details regarding the configuration of WLD environment variables.

C_Finalize

Synopsis

```
C_Finalize(
CK_VOID_PTR pReserved);
```

Description

This function behaves as specified in PKCS#11 but with the following additional features –
If there are no other active applications, ProtectToolkit C will free all allocated resources. The next call to C_INITIALIZE will therefore perform a full initialization of the system updating for any configuration changes.

C_GetInfo

Synopsis

```
C_GetInfo(
CK_INFO_PTR pInfo
);
```

Description

This function behaves as specified in PKCS#11.

The cryptokiVersion value is 2.11.

The manufacturerId is "SafeNet, Inc.”

The flags are all zero.

The libraryDescription is “ProtectServer ”, “CSA8000”, “CSA7000” or “Software Only” as appropriate.*

The libraryVersion represents the current version release number

C_GetFunctionList

Synopsis

```
C_GetFunctionList(CK_FUNCTION_LIST_PTR_PTR_PTR ppFunctionList);
```

Description

This function behaves as specified in PKCS #11.

Slot and Token Management Functions

C_GetSlotList

Synopsis

```
C_GetSlotList(
 CK_BBOOL tokenPresent,
 CK_SLOT_ID_PTR pSlotList,
 CK_ULONG_PTR pulCount
 );
```

Description

This function operates as specified in PKCS#11.

Note however that when multiple devices are installed in a single machine they will appear as a set of consecutive slots. For example, for two devices using their default configuration, 4 slots are visible. The first and third slots are normal user slots, the second and fourth slots are the Admin slots for their respective adapters.

Operation in WLD Mode

When ProtectToolkit is configured to operate in WLD mode, this function returns the list of slots specified in the WLD configuration. Specifically:

- When tokenPresent is FALSE, and pSlotList is NULL_PTR, the value *pulcount is set to hold the number of WLD Slots.
- When tokenPresent is FALSE, and pSlotList is not NULL_PTR, the value *pulcount is set to hold the number of WLD Slots and pSlotList contains the list of WLD Slots.
- When tokenPresent is TRUE, and pSlotList is NULL_PTR, the value *pulcount is set to hold the number of WLD Slots that have available HSM Tokens.
- When tokenPresent is TRUE, and pSlotList is not NULL_PTR, the value *pulcount is set to hold the number of WLD Slots that have available HSM Tokens and pSlotList contains the list of WLD Slots that have available HSM Tokens.

C_GetSlotInfo

Synopsis

```
C_GetSlotInfo(
 CK_SLOT_ID slotID,
 CK_SLOT_INFO_PTR pInfo

 );
```

Description

This function operates as specified in PKCS#11.

The information returned will vary depending on the ProtectToolkit C runtime in use as well as the actual slot type, for example, if it is a ProtectToolkit C user slot or a Smart Card slot.

This information is returned in the CK_SLOT_INFO structure.

SlotDescription	“ProtectServer :xxxx, “Safenet Software Only.” or smart card reader type.* WHERE XXXX IS THE SLOT SERIALNUMBER
ManufacturerID	"SafeNet, Inc.” or smart card reader manufacturer.
Flags	CKF_HW_SLOT (hardware only), CKF_REMOVABLE_DEVICE (smart card slots only).
HardwareVersion	Current hardware revision or 0.0 for software only.
FirmwareVersion	Current firmware version or 0.0 for software only.

Operation in WLD Mode

When ProtectToolkit is configured to operate in WLD mode, a random slot from the HSM Token List for the provided slot ID is chosen, so as not to overload a particular device and the command is forwarded to that slot. The following WLD specific information is returned in the CK_SLOT_INFO structure:

SlotDescription	The slot description specified for the virtual WLD Slot in environment variables ET_PTKC_WLD_SLOT_n. Refer to *ProtectToolkit C Administration Manual* for details.
Flags	The CKF_WLD_SLOT bit is set to indicate that it is a WLD Slot. If there are no HSM Tokens available for the particular slot, then the CKF_TOKEN_PRESENT bit in is set to 0.[1]

[1] This breaks PKCS#11 compliance, as this bit should be set to 0 if and only if CKF_REMOVABLE_DEVICE is set. The CKF_REMOVABLE_DEVICE bit is set only for Smart card Slots in the SafeNet implementation.

C_GetTokenInfo

Synopsis

```
C_GetTokenInfo(
  CK_SLOT_ID slotID,
  CK_TOKEN_INFO_PTR pInfo

  );
```

Description

This function operates as specified in PKCS#11. The information returned will vary depending on the type of slot specified by the slotID parameter. This information is returned in the `CK_TOKEN_INFO` structure.

Label	This is the string specified by the user during the C_InitToken command, unless the token is the administration token, in which case the value is: AdminToken(ssss) Where ssss is the HSM serial number.
ManufacturerID	"SafeNet, Inc."
Model	"PSI-E2:PLxxx" Where xxx is the performance level or smartcard manufacturer.
SerialNumber	"xxxx-xxxx" Where the first field is the HSM serial number and the second field is a randomly assigned token serial number or the smartcard serial number.
Flags	CKF_RNG (for non-smart card slots only) + CKF_CLOCK_ON_TOKEN (if the module's clock has been set) + CKF_DUAL_CRYPTO_OPERATIONS + Other flags based on the current state of the slot. CKF_LOGIN_REQUIRED flag is set if the security mode specifies "no public crypto". Admin slot have CKF_ADMIN_TOKEN and CKF_LOGIN_REQUIRED set.
ulMaxSessionCount	The value of that CKA_MAX_SESSIONS for the associated slot object.

ulSessionCount	Determined at run time – this is the total number of session to this Token by all applications.
ulMaxRwSessionCount	The value of that CKA_MAX_SESSIONS for the associated slot object.
ulRwSessionCount	Determined at run time – this is the number of RW sessions the calling application has to the Token.
ulMaxPinLen	CK_MAX_PIN_LEN = 32.
UlMinPinLength	This is the value specified in the configuration as shown by the CKA_MIN_PIN attribute of the slot object**.**
UlTotalPublicMemory	Determined at run time.
ulFreePublicMemory	Determined at run time.
ulTotalPrivateMemory	Determined at run time.
ulFreePrivateMemory	Determined at run time.
hardwareVersion	'G'.0 (or later)
FirmwareVersion	1.0 (or later)
UtcTime	Current time is returned if the modules clock has been set (else ASCII zeros are returned).

Operation in WLD Mode

When ProtectToolkit is configured to operate in WLD mode, a random slot from the HSM Token List for the provided slot ID is chosen, so as not to overload a particular device and the command is forwarded to that slot. The following WLD specific information is returned in the CK_TOKEN_INFO structure:

SerialNumber	The serial number specified for the virtual WLD Slot in environment variables ET_PTKC_WLD_SLOT_n. Refer to ProtectToolkit C Administration Manual for details.
Flags	The CKF_WLD_TOKEN bit is set to indicate that it is a WLD Token.

C_WaitForSlotEvent

Synopsis

```
C_WaitForSlotEvent(
  CK_FLAGS flags,
  CK_SLOT_ID_PTR pSlot,
  CK_VOID_PTR pReserved

  );
```

Description

This function operates as specified in PKCS#11 except –
The library cannot block while waiting for an event therefore the `CKF_DONT_BLOCK` must always be set.

If `CKF_DONT_BLOCK` is not set and there is no event pending on any slot then `CKR_FUNCTION_FAILED` is returned.

Slot Events supported -
There are no events supported by this library.

C_GetMechanismList

Synopsis

```
C_GetMechanismList(
 CK_SLOT_ID slotID,
 CK_MECHANISM_TYPE_PTR pMechanismList,
 CK_ULONG_PTR pulCount
 );
```

Description

This function operates as specified in PKCS#11.

See the section Mechanisms for a description of the mechanisms supported by this module.

Please note the list of mechanisms may vary at run time depending on Mode settings and other configuration values. For example the smart card slots do not support any mechanisms.

Operation in WLD Mode

When ProtectToolkit is configured to operate in WLD mode, a random slot from the HSM Token List for the provided slot ID is chosen, so as not to overload a particular device and the command is forwarded to that slot.

C_GetMechanismInfo

Synopsis

```
C_GetMechanismInfo(
 CK_SLOT_ID slotID,
 CK_MECHANISM_TYPE type,
 CK_MECHANISM_INFO_PTR pInfo
 );
```

Description

This function operates as specified in PKCS#11 with the following exception. Normally ProtectToolkit C will return `CKR_MECHANISM_INVALID` if the mechanism type is not recognized, however, if the EntrustReady Mode is set, the structure pointed to by *pInfo* is cleared and `CKR_OK` is returned.

See the section Mechanisms for a description of the mechanisms supported by this module.

Operation in WLD Mode

When ProtectToolkit is configured to operate in WLD mode, a random slot from the HSM Token List for the provided slot ID is chosen, so as not to overload a particular device and the command is forwarded to that slot.

C_InitToken

Synopsis

```
C_InitToken(
 CK_SLOT_ID slotID,
 CK_CHAR_PTR pPin,
 CK_ULONG ulPinLen,
 CK_CHAR_PTR pLabel
 );
```

Description

This function operates as specified in PKCS#11 but with these following extensions. This function is **disabled** if the NoClearPINs flag is set in the Mode register. Any attempt to call this function in this mode will result in a result in the `CKR_ACCESS_DENIED`error being returned. The Administrator must use the `CT_ResetToken`function instead.

The "protected authentication path" is not applicable to this module.

The module will detect if a session is active on the token and, if so, return CKR_SESSION_EXISTS.

If the token has been already initialized and the module is not in Entrust-ready modes then the supplied pin is checked against the current SO pin. If the pin is correct, the token is wiped and the label is set (the SO pin is not changed).

If the token is currently uninitialized, or the module is in Entrust-ready mode, the token is wiped, and the new label and SO pin are set.

The Admin token may not be re-initialized, this function will return `CKR_SLOT_ID_INVALID` if the specified slot id is the admin token.

Operation in WLD Mode

When ProtectToolkit is configured to operate in WLD mode, this function is not supported and will return the error CKR_FUNCTION_NOT_SUPPORTED.

CT_InitToken

Synopsis

```
CT_InitToken(
  CK_SESSION_HANDLE hSession,
  CK_SLOT_ID slotID,
  CK_CHAR_PTR pPin,
  CK_ULONG ulPinLen,
  CK_CHAR_PTR pLabel
  );
```

Description

This function is a SafeNet extension to PKCS#11, it allows the Administrator to initialize a new Token.

It initializes the token indicated by *slotId* with the SO pin (*pPin* and *ulPinLen*) and *pLabel*.
The session *hSession*, must be a session to the Admin Token of the adapter and be in RW User Mode for this function to succeed otherwise `CKR_SESSION_HANDLE_INVALID` is returned.

The *slotId* value must refer to a valid slot where the token in the slot must be in an un-initialized state, otherwise `CKR_SLOT_ID_INVALID` is returned. If the *slotID* is valid but the token is not present then `CKR_TOKEN_NOT_PRESENT` is returned.

Operation in WLD Mode

When ProtectToolkit is configured to operate in WLD mode, this function is not supported and will return the error CKR_FUNCTION_NOT_SUPPORTED.

CT_ResetToken

Synopsis

```
CT_ResetToken(
  CK_SESSION_HANDLE hSession,
  CK_CHAR_PTR pPin,
  CK_ULONG ulPinLen,
  CK_CHAR_PTR pLabel

  );
```

Description

This function is a SafeNet extension to PKCS#11, it will erase (reset) the token which the session is connected to.

The session must be in RW SO Mode for this function to succeed otherwise `CKR_USER_NOT_LOGGED_IN` is returned.

This function allows Token Security Officers to reset a Token. The module will detect if other sessions are active on the token and, if so, return `CKR_SESSION_EXISTS`.

This function will erase all objects it can from the token – depending on the token type some objects will no be erased. The token is left in an initialized state where the SO pin and label are set as specified by the *pPin* and *pLabel* parameters.

NOTE: *pPin* becomes the new SO pin and need not match the old SO pin value. The session is automatically terminated by this call.

Operation in WLD Mode

When ProtectToolkit is configured to operate in WLD mode, this function is not supported and returns the error CKR_FUNCTION_NOT_SUPPORTED.

C_InitPIN

Synopsis

```
C_InitPIN(
  CK_SESSION_HANDLE hSession,
  CK_CHAR_PTR pPin,
  CK_ULONG ulPinLen

  );
```

Description

This function operates as specified in PKCS#11 with the following extensions. When the module is in the NoClearPins mode, the host library protection system will encrypt the sensitive material before presenting it to the adapter.

The function returns an error if the Token has already had the user pin specified, that is, the SO does not have the rights to replace a user pin, only initialize it.

Operation in WLD Mode

When ProtectToolkit is configured to operate in WLD mode, this function is not supported and will return the error CKR_FUNCTION_NOT_SUPPORTED.

C_SetPIN

Synopsis

```
C_SetPIN(
  CK_SESSION_HANDLE hSession,
  CK_CHAR_PTR pOldPin,
  CK_ULONG ulOldLen,
  CK_CHAR_PTR pNewPin,
  CK_ULONG ulNewLen
);
```

Description

This function operates as specified in PKCS#11.

When the module is in the NoClearPINs mode the host library protection system will encrypt the sensitive material before presenting it to the adapter.

Operation in WLD Mode

When ProtectToolkit is configured to operate in WLD mode, this function is not supported and will return the error CKR_FUNCTION_NOT_SUPPORTED.

Session Management Functions

NOTE: ProtectToolkit C allows an application to have concurrent sessions with more than one token. It is also possible for a token to have concurrent sessions with more than one application.

C_OpenSession

Synopsis

```
C_OpenSession(
  CK_SLOT_ID slotID,
  CK_FLAGS flags,
  CK_VOID_PTR pApplication,
  CK_NOTIFY Notify,
  CK_SESSION_HANDLE_PTR phSession
);
```

Description

This function operates as specified in PKCS#11 with the following exceptions:

- The Notify parameter is ignored.
- The `CKF_SERIAL_SESSION`flag is ignored.
- PKCS#11 states “If the application calling `C_OpenSession` already has a R/W SO session open with the token, then any attempt to open a R/O session with the token fails with error code `CKR_SESSION_READ_WRITE_SO_EXISTS`” this is not enforced with ProtectToolkit C.

Operation in WLD Mode

When ProtectToolkit is configured to operate in WLD mode, the first C_OpenSession() call selects a random token from the list of available WLD tokens to open the session with. Subsequent C_OpenSession() calls, randomly select a token from those with the least number of sessions.

If successful, a WLD session handle is returned. The WLD session handle is internally mapped to the appropriate HSM token and session handle.

If unsuccessful, for ANY reason, another token is chosen and ProtectToolkit C retries to open a session utilizing this token. This is repeated until either a session is opened successfully or no more tokens are available.

If the HSM token used did not result in a session opening successfully for one of the following error conditions, the token will no longer be considered for WLD for the life of the application:

- CKR_GENERAL_ERROR
- CKR_DEVICE_ERROR
- CKR_MESSAGE_ERROR number space (SafeNet vendor defined)

NOTE: When the any of the above error conditions are detected C_OpenSession() will not return the associated error code as ProtectToolkit C will retry to open a session using another token until all tokens are exhausted. If there are no tokens available the error CKR_TOKEN_NOT_PRESENT are returned.

C_CloseSession

Synopsis

```
C_CloseSession(
  CK_SESSION_HANDLE hSession
  );
```

Description

This function operates as specified in PKCS#11 with the following exception

- ProtectToolkit C has no capability to “eject” the token from its reader.

C_CloseAllSessions

Synopsis

```
C_CloseAllSessions(
  CK_SLOT_ID slotID
  );
```

Description

This function operates as specified in PKCS#11 with the following exception

- ProtectToolkit C has no capability to “eject” the token from its reader. Further, this function will perform a “logout” on each token if necessary.

C_GetSessionInfo

Synopsis

```
C_GetSessionInfo(
 CK_SESSION_HANDLE hSession,
 CK_SESSION_INFO_PTR pInfo
);
```

Description

This function operates as specified in PKCS#11 with the following exception

- Any non-zero ulDeviceError value is cleared by this operation.

Operation in WLD Mode

When ProtectToolkit is configured to operate in WLD mode, the following WLD specific information is returned in the CK_SESSION_INFO structure:

SlotID	The Slot Number specified for the virtual WLD Slot in environment variables ET_PTKC_WLD_SLOT_n. Refer to the *ProtectToolkit C Administration Manual*.
Flags	The CKF_WLD_SESSION bit is set to indicate that it is a WLD Session.

C_GetOperationState

Synopsis

```
C_GetOperationState(
 CK_SESSION_HANDLE hSession,
 CK_BYTE_PTR pOperationState,
 CK_ULONG_PTR pulOperationStateLen
);
```

Description

`C_GetOperationState` obtains a copy of the cryptographic Operation State for a session, encoded as a string of Bytes. *hSession* is the session's handle; *pOperationState* points to the location that receives the state; *pulOperationStateLen* points to the location that receives the length in bytes of the state.

ProtectToolkit C implements a subset of the full PKCS#11 specification – only the current Message Digest state and object attribute search state may be saved and restored. This means that the current encryption, decryption, signing and verification states are *not* saved by this function.

The state need not have been obtained from the same session (the "source session") as it is being restored to (the "destination session"). However, the source session and destination session should have a common session state (e.g., CKS_RW_USER_FUNCTIONS), and should be with a common token. Message digest operation states may be carried across logins but not across different Cryptoki implementations.

Operation in WLD Mode

When ProtectToolkit is configured to operate in WLD mode, this function is not supported and will return the error CKR_FUNCTION_NOT_SUPPORTED.

C_SetOperationState

Synopsis

```
C_SetOperationState(
  CK_SESSION_HANDLE hSession,
  CK_BYTE_PTR pOperationState,
  CK_ULONG ulOperationStateLen,
  CK_OBJECT_HANDLE hEncryptionKey,
  CK_OBJECT_HANDLE hAuthenticationKey
  );
```

Description

`C_SetOperationState` restores the cryptographic Operations State of a session from a string of bytes obtained with `C_GetOperationState`. ProtectToolkit C implements a subset of the full PKCS#11 specification – only the current Message Digest state and object search state may be saved and restored.

hSession is the session's handle; *pOperationState* points to the location holding the saved state; *ulOperationStateLen* holds the length of the saved state; *hEncryptionKey* and *hAuthenticationKey* must be zero.

The state need not have been obtained from the same session (the "source session") as it is being restored to (the "destination session"). However, the source session and destination session should have a common session state (for example, `CKS_RW_USER_FUNCTIONS`), and should be with a common tokenMessage digest operation states may be carried across logins but not across different Cryptoki implementations.

If `C_SetOperationState` is supplied with a saved cryptographic Operations State, which it determines is not a valid saved State, it fails with the error `CKR_SAVED_STATE_INVALID`. Invalid States include cryptographic Operations State from a session with a different session state and cryptographic Operations State from a different token.

`C_SetOperationState` can successfully restore the message digest Operations State to a session, even if that session has an active message digest or object search operation when `C_SetOperationState`is called. The ongoing operations are abruptly cancelled. However if the saved state did not contain an active message digest operation and the current session does, then the `C_SetOperationState`function will have no effect on the current operation.

Operation in WLD Mode

When ProtectToolkit is configured to operate in WLD mode, this function is not supported and will return the error CKR_FUNCTION_NOT_SUPPORTED.

C_Login

Synopsis

```
C_Login(
  CK_SESSION_HANDLE hSession,
  CK_USER_TYPE userType,
  CK_CHAR_PTR pPin,
  CK_ULONG ulPinLen
  );
```

Description

This function operates as specified in PKCS#11 with the following exceptions

- If the security mode NoClearPINs is enabled, then the pin value is encrypted by the host library before it is supplied to the module.
- To negate a brute force attack on the PIN, after the third failed attempt, a delay is imposed delay (incrementing in multiples of 5 seconds) until the next presented PIN is checked.

  For example, after the third failed attempt, the device imposes a delay of 1*5 seconds, after the fourth the delay is 2*5=10 seconds, after the fifth, the delay is 3*5=15 seconds, and so on.

  If a PIN presentation occurs before the delay period has expired, the attempt fails with CKR_PIN_LOCKED.

Operation in WLD Mode

When ProtectToolkit is configured to operate in WLD mode, the login state is replicated across all tokens in user slots associated with the same WLD slot. For example, if an application has 3 sessions, across 3 HSMs, with one session on each HSM then any change in the login state in one session, will result in the session on the other 2 HSMs being changed to the same session state.

Temporary Pin Login

Under Cryptoki all authentication of users to the HSM is valid for the calling process only. Each application must authenticate separately. Once a process has authenticated is granted appropriate access to the services of the token.

With PTK C - if a process forks a new process then the new process must authenticate itself - it can not inherit the authentication of the parent.

The Temporary Pin feature in this spec describes a new applications authentication method where a parent process can pass on its authentication to a child process without having to pass the sensitive pin value.

Challenge Response Login

A new type of User Authentication is provided. Instead of having to present the Pin value directly to the HSM the user will request a random challenge, for a specified password, from the HSM and then present a response computed from the challenge and password using a One Way Function.

The HSM will authenticate the user by verifying the response with the specified password and the most recently issued random challenge.

A new CKO_HW_FEATURE object called CKH_VD_USER is provided by the firmware to allow the application to obtain the random challenge for either the User Password or SO Password.

The Object has an attribute that an application can read to generate and obtain a random challenge.
A new challenge value will generated each time the attribute is read. A separate Challenge is held for each registered application. The same challenge can be used for User or SO authentication.
The calling application converts the challenge into a Response by using the following algorithm:-

```
Response = SHA-256( challenge | PVC)
Where PVC = LEFT64BIT( SHA1(password | userTypeByte)
```

A host side static library function CT_Gen_Auth_Response is provided in the SDK to assist developers in using this scheme.

The CKH_VD_USER has an attribute that an application can read to generate and obtain a Temporary Pin. Only one SO and one User Temporary pin may exist at any one time in any single Token. Each read from this attribute will generate a new Temporary Pin.
Any Temporary Pins in a Token are automatically destroyed when the generating process logs off or is terminated or the HSM has reset – whichever comes first.

The Response and Temporary Pin are passed to the HSM using the C_Login function. The Function will be extended such that unused bits in the userType parameter will be set to indicate that a Response value or Temporary PIN is being used instead of the normal password.

The following bits are added to the userType parameter of the C_Login Function to specify the type of authentication required.

```
#define CKF_AUTH_RESPONSE           0x00000100
#define CKF_AUTH_TEMP_PIN           0x00001000
```

C_Logout

Synopsis

```
C_Logout(
  CK_SESSION_HANDLE hSession

  );
```

Description

This function operates as specified in PKCS #11.

Operation in WLD Mode

When ProtectToolkit is configured to operate in WLD mode, the login state is replicated across all tokens in user slots associated with the same WLD slot. For example, if an application has 3 sessions, across 3 HSMs, with one session on each HSM then any change in the login state in one session, will result in the session on the other 2 HSMs being changed to the same session state.

Object Management Functions

C_CreateObject

Synopsis

```
C_CreateObject(
  CK_SESSION_HANDLE hSession,
  CK_ATTRIBUTE_PTR pTemplate,
  CK_ULONG ulCount,
  CK_OBJECT_HANDLE_PTR phObject

  );
```

Description

This function operates as specified in PKCS#11 with the following exceptions. If the security mode NoClearPINs is enabled then the host library version of the function will encrypt the template before submitting it to the module and the module function will verify the data was encrypted.

If the object is of type `CKO_PUBLIC_KEY`, `CKO_PRIVATE_KEY`, `CKO_CERTIFICATE`or `CKO_CERTIFICATE_REQUEST` and the key type is `CKK_RSA`or `CKK_DSA` then the key is checked for validity.

C_CopyObject

Synopsis

```
C_CopyObject(
  CK_SESSION_HANDLE hSession,
  CK_OBJECT_HANDLE hObject,
  CK_ATTRIBUTE_PTR pTemplate,
  CK_ULONG ulCount,
  CK_OBJECT_HANDLE_PTR phNewObject


  );
```

Description

This function operates as specified in PKCS#11. except that if the base object has a valid CKA_USAGE_LIMIT attribute then the base object is deleted after a successful copy.

NOTE: If the "Increased Security" flag is set as part of the security policy, then C_CopyObject does not allow changing the CKA_MODIFIABLE flag from FALSE to TRUE.

Operation in WLD Mode

When ProtectToolkit is configured to operate in WLD mode, this function is not supported and will return the error CKR_FUNCTION_NOT_SUPPORTED.

CT_CopyObject

Synopsis

```
CT_CopyObject(
  CK_SESSION_HANDLE hDestSession,
  CK_SESSION_HANDLE hSourceSession,
  CK_OBJECT_HANDLE hObject,
  CK_ATTRIBUTE_PTR pTemplate,
  CK_ULONG ulCount,
  CK_OBJECT_HANDLE_PTR phNewObject

  );
```

Description

This function is a SafeNet extension to PKCS #11. It is identical to the `C_CopyObject` function with the exception that it is capable of copying objects from one token to another token where the two tokens belong to the same adapter.

This function copies an object from one session to another session, creating a new object for the copy. *hSession* is the source session's handle; *hObject* is the destination's session handle; *hObject* is the object's handle; *pTemplate* points to the template for the new object; *ulCount* is the number of attributes in the template; *phNewObject* points to the location that receives the handle for the copy of the object.

If the base object has a valid CKA_USAGE_LIMIT attribute then the base object is deleted after a successful copy.

The template may specify new values for any attributes of the object that can ordinarily be modified (*e.g.*, in the course of copying a secret key, a key's CKA_EXTRACTABLE attribute may be changed from TRUE to FALSE, but not the other way around. If this change is made, the new key's CKA_NEVER_EXTRACTABLE attribute will have the value FALSE.

Similarly, the template may specify that the new key's CKA_SENSITIVE attribute be TRUE; the new key will have the same value for its CKA_ALWAYS_SENSITIVE attribute as the original key). It may also specify new values of the CKA_TOKEN and CKA_PRIVATE attributes (*e.g.*, to copy a session object to a token object).

If the template specifies a value of an attribute which is incompatible with other existing attributes of the object, the call fails with the return code `CKR_TEMPLATE_INCONSISTENT`.

If a call to CT_CopyObject cannot support the precise template supplied to it, it will fail and return without creating any object.

Only session objects can be created during a read-only session. Only public objects can be created unless the normal user is logged in.

NOTE: If the "Increased Security" flag is set as part of the security policy, then C_CopyObject does not allow changing the CKA_MODIFIABLE flag from FALSE to TRUE.

C_DestroyObject

Synopsis

```
C_DestroyObject(
  CK_SESSION_HANDLE hSession,
  CK_OBJECT_HANDLE hObject
  );
```

Description

This function operates as specified in PKCS#11.

If the object has the optional attribute `CKA_DELETABLE`set to `FALSE` the object cannot be deleted with this function and `CKR_OBJECT_READ_ONLY` is returned.

C_GetObjectSize

Synopsis

```
C_GetObjectSize(
  CK_SESSION_HANDLE hSession,
  CK_OBJECT_HANDLE hObject,
  CK_ULONG_PTR pulSize
  );
```

Description

This function operates as specified in PKCS#11.

ProtectToolkit C interprets the object size to be the amount of memory guaranteed to be sufficient to encode the object's attributes.

C_GetAttributeValue

Synopsis

```
C_GetAttributeValue(
  CK_SESSION_HANDLE hSession,
  CK_OBJECT_HANDLE hObject,
  CK_ATTRIBUTE_PTR pTemplate,
  CK_ULONG ulCount

);
```

Description

This function operates as specified in PKCS#11 with the following extensions. With ProtectToolkit C it is possible to enumerate through all attributes for a given object. This extension is supported as follows.

The first call `C_GetAttributeValue` operates as follows to initialize the enumeration.

```
CK_ATTRIBUTE at;
  rv = C_GetAttributeValue(hSession, hObject, &at, 0);
  then to get all the attributes loop as follows
  for (;;) {


at.type = CKA_ENUM_ATTRIBUTE;
  at.pValue = 0;
  rv = C_GetAttributeValue(hSession, hObject, &at, 1);
  if ( rv == CKR_ATTRIBUTE_TYPE_INVALID )

break; /* got all the attributes */
  }
```

Sensitive attributes are returned with the type and length information but an empty value, and also return a result value of CKR_ATTRIBUTE_SENSITIVE. On implementations where this extension is not supported, the calls to C_GetAttributeType are likely to fail with the CKR_ATTRIBUTE_TYPE_INVALIDerror code.

With a result code of CKR_OKor CKR_ATTRIBUTE_SENSITIVE the CK_ATTRIBUTE structure has the type and valueLen fields set appropriately for the next attribute, however the pValuefield will be NULL_PTR. To retrieve the actual value of the attribute it is necessary to allocate the required room for the value and then make a second call to C_GetAttributeValue.

In addition special processing or access checks may be made if the object is a Hardware Feature. See the section Hardware Feature Objects for more details on hardware features.

C_SetAttributeValue

Synopsis

```
C_SetAttributeValue(
  CK_SESSION_HANDLE hSession,
  CK_OBJECT_HANDLE hObject,
  CK_ATTRIBUTE_PTR pTemplate,
  CK_ULONG ulCount

);
```

Description

This function operates as specified in PKCS#11.

In addition special processing or access checks may be made if the object is a Hardware Feature. See the section Hardware Feature Objects for more details on hardware features.

C_FindObjectsInit

Synopsis

```
C_FindObjectsInit(
  CK_SESSION_HANDLE hSession,
  CK_ATTRIBUTE_PTR pTemplate,
  CK_ULONG ulCount
  );
```

Description

This function operates as specified in PKCS#11 with the following exception:

PKCS#11 states that to match `CKO_HW_FEATURE` objects this class must be specified in the supplied template. ProtectToolkit C does not enforce this requirement.

C_FindObjects

Synopsis

```
C_FindObjects(CK_SESSION_HANDLE hSession,CK_OBJECT_HANDLE_PTR
  phObject,CK_ULONG ulMaxObjectCount,CK_ULONG_PTR pulObjectCount
);
```

Description

This function operates as specified in PKCS#11.

C_FindObjectsFinal

Synopsis

```
C_FindObjectsFinal(CK_SESSION_HANDLE hSession );
```

Description

This function operates as specified in PKCS#11.

Encryption Functions

C_EncryptInit

Synopsis

```
C_EncryptInit(
  CK_SESSION_HANDLE hSession,
  CK_MECHANISM_PTR pMechanism,
  CK_OBJECT_HANDLE hKey

);
```

Description

This function operates as specified in PKCS#11.

The session will retain its initialized state even when a `C_Encrypt`or `C_EncryptFinal` operation has occurred.

If the `CKF_LOGIN_REQUIRED` flag is set for the Token associated with the provided session the session state must be either `CKS_RW_USER_FUNCTIONS`, or `CKS_RO_USER_FUNCTIONS`otherwise the error result `CKR_USER_NOT_LOGGED_IN` is returned.

If the *hKey* parameter refers to a certificate object this function will perform the same certificate verification as specified in the `C_VerifyInit` function.

If the object referenced by the *hKey* parameter has the `CKA_USAGE_COUNT`attribute its value is incremented by this function.

C_Encrypt

Synopsis

```
C_Encrypt(
  CK_SESSION_HANDLE hSession,
  CK_BYTE_PTR pData,
  CK_ULONG ulDataLen,
  CK_BYTE_PTR pEncryptedData,
  CK_ULONG_PTR pulEncryptedDataLen

);
```

Description

This function operates as specified in PKCS#11 except for the following:

- Symmetric cipher operations are terminated by this function.
- `C_Encrypt` can be used to terminate a multi-part operation.

- Although this function will terminate the current encryption operation, the session's encryption state will not be cleared.

  NOTE: If the mechanism in use is a multi-part mechanism and the data supplied exceeds a single block, that portion of the data is processed regardless of the result returned by the call. For example if 12 bytes are passed to a DES ECB operation, 8 bytes are processed even though an error result (due to the padding requirements not being met) is returned.

- Cryptoki specifies that a successful return from one of these functions (when not used for length prediction) should result in the cipher state of that session being reset (e.g. to the uninitialized state). ProtectToolkit C however leaves the state initialized so that another operation (using the same key) may be preformed without calling the appropriate C_xxxInit function.

C_EncryptUpdate

Synopsis

```
C_EncryptUpdate(CK_SESSION_HANDLE hSession,CK_BYTE_PTR pPart,CK_ULONG
 ulPartLen,CK_BYTE_PTR pEncryptedPart,CK_ULONG_PTR pulEncryptedPartLen

);
```

Description

This function operates as specified in PKCS#11.

C_EncryptFinal

Synopsis

```
C_EncryptFinal(
 CK_SESSION_HANDLE hSession,
 CK_BYTE_PTR pLastEncryptedPart,
 CK_ULONG_PTR pulLastEncryptedPartLen


);
```

Description

This function operates as specified in PKCS#11.

Although this function will terminate the current encryption operation the session's encryption state will not be cleared.

NOTE: Cryptoki specifies that a successful return from one of these functions (when not used for length prediction) should result in the cipher state of that session being reset (e.g. to the uninitialized state). ProtectToolkit C however leaves the state initialized so that another operation (using the same key) may be preformed without calling the appropriate C_xxxInit function.

Decryption Functions

C_DecryptInit

Synopsis

```
C_DecryptInit(
  CK_SESSION_HANDLE hSession,
  CK_MECHANISM_PTR pMechanism,
  CK_OBJECT_HANDLE hKey

);
```

Description

This function operates as specified in PKCS#11.

The session will retain its initialized state even when a `C_Decrypt`or `C_DecryptFinal` operation has occurred.

If the `CKF_LOGIN_REQUIRED` flag is set for the Token associated with the provided session the session state must be either `CKS_RW_USER_FUNCTIONS`or `CKS_RO_USER_FUNCTIONS`,otherwise the error result `CKR_USER_NOT_LOGGED_IN` is returned.

If the object referenced by the *hKey* parameter has the `CKA_USAGE_COUNT`attribute its value is incremented by this function.

C_Decrypt

Synopsis

```
C_Decrypt(
  CK_SESSION_HANDLE hSession,
  CK_BYTE_PTR pEncryptedData,
  CK_ULONG ulEncryptedDataLen,
  CK_BYTE_PTR pData,
  CK_ULONG_PTR pulDataLen

);
```

Description

This function operates as specified in PKCS#11 except for the following:

Symmetric cipher operations are terminated by this function. Although this function terminates the current decryption operation the session's decryption state is not cleared.

NOTE: If the mechanism in use is a multi-part mechanism and the data supplied exceeds a single block, that portion of the data is processed regardless of the result returned by the call. For example if 12 bytes are passed to a DES ECB operation, 8 bytes are processed even though an error result (due to the padding requirements not being met) is returned.

Cryptoki specifies that a successful return from one of these functions (when not used for length prediction) should result in the cipher state of that session being reset (e.g. to the uninitialized state). ProtectToolkit C however leaves the state initialized so that another operation (using the same key) may be preformed without calling the appropriate C_xxxInit function.

C_DecryptUpdate

Synopsis

```
C_DecryptUpdate(CK_SESSION_HANDLE hSession,CK_BYTE_PTR
 pEncryptedPart,CK_ULONG ulEncryptedPartLen,CK_BYTE_PTR
 pPart,CK_ULONG_PTR pulPartLen

);
```

Description

This function operates as specified in PKCS#11.

C_DecryptFinal

Synopsis

```
C_DecryptFinal(
 CK_SESSION_HANDLE hSession,
 CK_BYTE_PTR pLastPart,
 CK_ULONG_PTR pulLastPartLen


);
```

Description

This function operates as specified in PKCS#11.

Although this function will terminate the current encryption operation the session's decryption state will not be cleared.

NOTE: Cryptoki specifies that a successful return from one of these functions (when not used for length prediction) should result in the cipher state of that session being reset (e.g. to the uninitialized state). ProtectToolkit C however leaves the state initialized so that another operation (using the same key) may be preformed without calling the appropriate C_xxxInit function.

Message Digesting Functions

C_DigestInit

Synopsis

```
C_DigestInit(
  CK_SESSION_HANDLE hSession,
  CK_MECHANISM_PTR pMechanism


);
```

Description

This function operates as specified in PKCS#11. Note that it is not required for the user to be logged in to access this function.

C_Digest

Synopsis

```
C_Digest(CK_SESSION_HANDLE hSession,CK_BYTE_PTR pData,CK_ULONG
  ulDataLen,CK_BYTE_PTR pDigest,CK_ULONG_PTR pulDigestLen
);
```

Description

This function operates as specified in PKCS#11.

C_DigestUpdate

Synopsis

```
C_DigestUpdate(CK_SESSION_HANDLE hSession,CK_BYTE_PTR pPart,CK_ULONG
  ulPartLen
);
```

Description

This function operates as specified in PKCS#11.

C_DigestKey

Synopsis

```
C_DigestKey(
  CK_SESSION_HANDLE hSession,
  CK_OBJECT_HANDLE hKey


);
```

Description

This function operates as specified in PKCS#11, although it may be used on any PKCS#11 object.

If the CKF_LOGIN_REQUIRED flag is set for the Token associated with the provided session the session state must be either CKS_RW_USER_FUNCTIONSor CKS_RO_USER_FUNCTIONS, otherwise the error result CKR_USER_NOT_LOGGED_IN is returned.

C_DigestFinal

Synopsis

```
C_DigestFinal(CK_SESSION_HANDLE hSession,CK_BYTE_PTR pDigest,CK_ULONG_PTR
 pulDigestLen

);
```

Description

This function operates as specified in PKCS#11.

Signing and MACing Functions

C_SignInit

Synopsis

```
C_SignInit(
 CK_SESSION_HANDLE hSession,
 CK_MECHANISM_PTR pMechanism,
 CK_OBJECT_HANDLE hKey


);
```

Description

This function operates as specified in PKCS#11.

In addition it is required to specify the signing key and signing mechanism used to create X509 certificates with the CKM_ENCODE_X_509, CKM_ENCODE_LOCAL_CERTand CKM_ENCODE_PKCS10 mechanisms.

If the CKF_LOGIN_REQUIRED flag is set for the Token associated with the provided session, the session state must be either CKS_RW_USER_FUNCTIONS, or CKS_RO_USER_FUNCTIONSotherwise the error result CKR_USER_NOT_LOGGED_IN is returned.

If the object referenced by the *hKey* parameter has the CKA_USAGE_COUNTattribute its value is incremented by this function.

C_Sign

Synopsis

```
C_Sign(CK_SESSION_HANDLE hSession,CK_BYTE_PTR pData,CK_ULONG
 ulDataLen,CK_BYTE_PTR pSignature,CK_ULONG_PTR pulSignatureLen

);
```

Description

This function operates as specified in PKCS#11.

C_SignUpdate

Synopsis

```
C_SignUpdate(CK_SESSION_HANDLE hSession,CK_BYTE_PTR pPart,CK_ULONG
 ulPartLen

);
```

Description

This function operates as specified in PKCS#11.

C_SignFinal

Synopsis

```
C_SignFinal(CK_SESSION_HANDLE hSession,CK_BYTE_PTR pSignature,CK_ULONG_PTR
 pulSignatureLen

);
```

Description

This function operates as specified in PKCS#11.

C_SignRecoverInit

Synopsis

```
C_SignRecoverInit(
 CK_SESSION_HANDLE hSession,
 CK_MECHANISM_PTR pMechanism,
 CK_OBJECT_HANDLE hKey


);
```

Description

This function operates as specified in PKCS#11.

If the `CKF_LOGIN_REQUIRED` flag is set for the Token associated with the provided session the session state must be either `CKS_RW_USER_FUNCTIONS`, or `CKS_RO_USER_FUNCTIONS`otherwise the error result `CKR_USER_NOT_LOGGED_IN` is returned.

If the object referenced by the *hKey* parameter has the `CKA_USAGE_COUNT` attribute its value is incremented by this function.

C_SignRecover

Synopsis

```
C_SignRecover(CK_SESSION_HANDLE hSession,CK_BYTE_PTR pData,CK_ULONG
 ulDataLen,CK_BYTE_PTR pSignature,CK_ULONG_PTR pulSignatureLen
);
```

Description

This function operates as specified in PKCS#11.

Functions for Verifying Signatures and MACs

C_VerifyInit

Synopsis

```
C_VerifyInit(
 CK_SESSION_HANDLE hSession,
 CK_MECHANISM_PTR pMechanism,
 CK_OBJECT_HANDLE hKey

);
```

Description

This function operates as specified in PKCS#11.

If the CKF_LOGIN_REQUIRED flag is set for the Token associated with the provided session the session state must be either CKS_RW_USER_FUNCTIONSor CKS_RO_USER_FUNCTIONS, otherwise the error CKR_USER_NOT_LOGGED_IN is returned.

If the object referenced by the hKey parameter has the CKA_USAGE_COUNTattribute its value is incremented by this function.

ProtectToolkit C also allows that *hKey* may specify a certificate object in place of a public key. In this case the certificate object is verified with the algorithm below. If this verification succeeds the session is initialized using the public key stored in the certificate. If the verification fails CKR_INVALID_KEY is returned and the session is not initialized. Further the certificate object's CKA_TRUST_LEVEL is updated to indicate that the verification has failed.

To perform the certificate verification the object's CKA_TRUSTED is checked. If it has the value TRUE the verification succeeds. If the attribute has the value FALSE the certificate is validated.

For self-signed certificates (that is, where the subject and the issuer are the same) the certificate is validated if the CKA_TRUSTED is TRUE and the certificate's signature is correct. If CKA_TRUSTED is FALSE for a self-signed certificate then the validation fails with CKR_CERT_NOT_VALIDATED. If the certificate is not self-signed, a search is made for the issuer's certificate which is the certificate whose CKA_SUBJECT matches the CKA_ISSUER of the current certificate. If the issuer's certificate is not found, the verification fails. If a matching issuer's certificate is found the verification algorithm is performed on that certificate, and if that succeeds the original certificate's signature is verified. Issuer certificate validation will continue recursively up the certificate chain until a trusted certificate (self signed or not) is reached or a certificate in the chain fails validation for any reason including not being present.

NOTE: This function does not enforce certificate expiry or key usage flags store in the certificate. Rather it relies on the standard Cryptoki attributes. This function will not always fail when an inappropriate key type is supplied. For example, if a private key is supplied to the function, it may succeed. In this case, however, the C_Verify will never return CKA_OK.

C_Verify

Synopsis

```
C_Verify(CK_SESSION_HANDLE hSession,CK_BYTE_PTR pData,CK_ULONG
 ulDataLen,CK_BYTE_PTR pSignature,CK_ULONG ulSignatureLen

);
```

Description

This function operates as specified in PKCS#11.

C_VerifyUpdate

Synopsis

```
C_VerifyUpdate(CK_SESSION_HANDLE hSession,CK_BYTE_PTR pPart,CK_ULONG
 ulPartLen

);
```

Description

This function operates as specified in PKCS#11.

C_VerifyFinal

Synopsis

```
C_VerifyFinal(CK_SESSION_HANDLE hSession,CK_BYTE_PTR pSignature,CK_ULONG
 ulSignatureLen

);
```

Description

This function operates as specified in PKCS#11.

C_VerifyRecoverInit

Synopsis

```
C_VerifyRecoverInit(
 CK_SESSION_HANDLE hSession,
 CK_MECHANISM_PTR pMechanism,
 CK_OBJECT_HANDLE hKey


);
```

Description

This function operates as specified in PKCS#11.

- If the `CKF_LOGIN_REQUIRED` flag is set for the Token associated with the provided session the session state must be either `CKS_RW_USER_FUNCTIONS`or `CKS_RO_USER_FUNCTIONS`, otherwise the error `CKR_USER_NOT_LOGGED_IN` is returned.
- If the object referenced by the *hKey* parameter has the `CKA_USAGE_COUNT`attribute its value is incremented by this function.
- If the *hKey* parameter refers to a certificate object this function will perform the same certificate verification as specified in the `C_VerifyInit` function.

C_VerifyRecover

Synopsis

```
C_VerifyRecover(CK_SESSION_HANDLE hSession,CK_BYTE_PTR pSignature,CK_ULONG
 ulSignatureLen,CK_BYTE_PTR pData,CK_ULONG_PTR pulDataLen

);
```

Description

This function operates as specified in PKCS#11.

Dual-function Cryptographic Functions

NOTE: ProtectToolkit C provides the following functions to perform two cryptographic operations "simultaneously" within a session. These functions are provided so as to avoid unnecessarily passing data back and forth to and from a token.

C_DigestEncryptUpdate

Synopsis

```
C_DigestEncryptUpdate(
 CK_SESSION_HANDLE hSession,
 CK_BYTE_PTR pPart,
 CK_ULONG ulPartLen,
 CK_BYTE_PTR pEncryptedPart,
 CK_ULONG_PTR pulEncryptedPartLen


);
```

Description

This function operates as specified in PKCS#11.

C_DecryptDigestUpdate

Synopsis

```
C_DecryptDigestUpdate(CK_SESSION_HANDLE hSession,CK_BYTE_PTR
 pEncryptedPart,CK_ULONG ulEncryptedPartLen,CK_BYTE_PTR
 pPart,CK_ULONG_PTR pulPartLen

);
```

Description

This function operates as specified in PKCS#11.

C_SignEncryptUpdate

Synopsis

```
C_SignEncryptUpdate(CK_SESSION_HANDLE hSession,CK_BYTE_PTR pPart,CK_ULONG
 ulPartLen,CK_BYTE_PTR pEncryptedPart,CK_ULONG_PTR pulEncryptedPartLen

);
```

Description

This function operates as specified in PKCS#11.

C_DecryptVerifyUpdate

Synopsis

```
C_DecryptVerifyUpdate(CK_SESSION_HANDLE hSession,CK_BYTE_PTR
 pEncryptedPart,CK_ULONG ulEncryptedPartLen,CK_BYTE_PTR
 pPart,CK_ULONG_PTR pulPartLen

);
```

Description

This function operates as specified in PKCS#11.

Key Management Functions

C_GenerateKey

Synopsis

```
C_GenerateKey(
  CK_SESSION_HANDLE hSession
  CK_MECHANISM_PTR pMechanism,
  CK_ATTRIBUTE_PTR pTemplate,
  CK_ULONG ulCount,
  CK_OBJECT_HANDLE_PTR phKey


);
```

Description

This function operates as specified in PKCS#11.

If the CKF_LOGIN_REQUIRED flag is set for the Token associated with the provided session the session state must be either CKS_RW_USER_FUNCTIONSor CKS_RO_USER_FUNCTIONS,otherwise the error CKR_USER_NOT_LOGGED_IN is returned.

C_GenerateKeyPair

Synopsis

```
C_GenerateKeyPair(
  CK_SESSION_HANDLE hSession,
  CK_MECHANISM_PTR pMechanism,
  CK_ATTRIBUTE_PTR pPublicKeyTemplate,
  CK_ULONG ulPublicKeyAttributeCount,
  CK_ATTRIBUTE_PTR pPrivateKeyTemplate,
  CK_ULONG ulPrivateKeyAttributeCount,
  CK_OBJECT_HANDLE_PTR phPublicKey,
  CK_OBJECT_HANDLE_PTR phPrivateKey


);
```

Description

This function operates as specified in PKCS#11.

If the CKF_LOGIN_REQUIRED flag is set for the Token associated with the provided session the session state must be either CKS_RW_USER_FUNCTIONSor CKS_RO_USER_FUNCTIONS,otherwise the error CKR_USER_NOT_LOGGED_IN is returned.

If CKA_ID is not specified in either template then the library sets default values for these that are the same for both public and private object with a high likelihood of being unique. The value is a SHA1 hash of the modulus.

C_WrapKey

Synopsis

```
C_WrapKey(
  CK_SESSION_HANDLE hSession,
  CK_MECHANISM_PTR pMechanism,
  CK_OBJECT_HANDLE hWrappingKey,
  CK_OBJECT_HANDLE hKey,
  CK_BYTE_PTR pWrappedKey,
  CK_ULONG_PTR pulWrappedKeyLen


);
```

Description

This function operates as specified in PKCS#11.

If the CKF_LOGIN_REQUIRED flag is set for the Token associated with the provided session the session state must be either CKS_RW_USER_FUNCTIONSor CKS_RO_USER_FUNCTIONS,otherwise the error CKR_USER_NOT_LOGGED_IN is returned.

C_UnwrapKey

Synopsis

```
C_UnwrapKey(
  CK_SESSION_HANDLE hSession,
  CK_MECHANISM_PTR pMechanism,
  CK_OBJECT_HANDLE hUnwrappingKey,
  CK_BYTE_PTR pWrappedKey,
  CK_ULONG ulWrappedKeyLen,
  CK_ATTRIBUTE_PTR pTemplate,
  CK_ULONG ulAttributeCount,
  CK_OBJECT_HANDLE_PTR phKey


);
```

Description

This function operates as specified in PKCS#11.

If the CKF_LOGIN_REQUIRED flag is set for the Token associated with the provided session the session state must be either CKS_RW_USER_FUNCTIONSor CKS_RO_USER_FUNCTIONS,otherwise the error CKR_USER_NOT_LOGGED_IN is returned.

C_DeriveKey

Synopsis

```
C_DeriveKey(
  CK_SESSION_HANDLE hSession,
  CK_MECHANISM_PTR pMechanism,
  CK_OBJECT_HANDLE hBaseKey,
  CK_ATTRIBUTE_PTR pTemplate,
  CK_ULONG ulAttributeCount,
  CK_OBJECT_HANDLE_PTR phKey


);
```

Description

This function operates as specified in PKCS#11.

If the CKF_LOGIN_REQUIRED flag is set for the Token associated with the provided session the session state must be either CKS_RW_USER_FUNCTIONSor CKS_RO_USER_FUNCTIONS, otherwise the error CKR_USER_NOT_LOGGED_IN is returned.

Simple derivation mechanisms are restricted to working on secret keys of type CKK_GENERIC_SECRET.

Random Number Generation Functions

C_SeedRandom

Synopsis

```
C_SeedRandom(
  CK_SESSION_HANDLE hSession,
  CK_BYTE_PTR pSeed,
  CK_ULONG ulSeedLen


);
```

Description

This function operates as specified in PKCS#11, however, it is not required to be called as the ProtectServer adapter has a hardware random generation source.

Also note this function will only operate for those tokens with the CKF_RNG flag set in their CK_TOKEN_INFO flags.

C_GenerateRandom

Synopsis

```
C_GenerateRandom(
  CK_SESSION_HANDLE hSession,
  CK_BYTE_PTR pRandomData,
  CK_ULONG ulRandomLen


);
```

Description

This function operates as specified in PKCS#11.

Also note this function will only operate for those tokens with the CKF_RNGflag set in their CK_TOKEN_INFOflags.

Parallel Function Management Functions

NOTE: ProtectToolkit C provides the following functions for managing parallel execution of cryptographic functions. These functions exist only for backward compatibility.

C_GetFunctionStatus

Synopsis

```
C_GetFunctionStatus(
  CK_SESSION_HANDLE hSession


);
```

Description

This function operates as specified in PKCS#11.

`C_GetFunctionStatus` is a legacy function, which will simply return the value `CKR_FUNCTION_NOT_PARALLEL`.

C_CancelFunction

Synopsis

```
C_CancelFunction(
  CK_SESSION_HANDLE hSession
  );
```

Description

This function operates as specified in PKCS#11.

`C_GetFunctionStatus` is a legacy function, which will simply return the value `CKR_FUNCTION_NOT_PARALLEL`.

Extra Functions

CT_PresentTicket

```
CK_DEFINE_FUNCTION(CK_RV, CT_PresentTicket)(
    CK_SESSION_HANDLE hSession,
    CK_OBJECT_HANDLE hObj,
    CK_MECHANISM_PTR pMechanism,
    CK_BYTE_PTR pTicket,
    CK_ULONG ulTicketLen
);
```

This function is a SafeNet extension to PKCS#11.

This function allows a process to present a security related cryptogram to the HSM. The cryptogram is specified by pTicket and ulTicketLen.

When Secure Messaging System is in 'No Clear Pins' mode then this function will expect all request data to be encrypted.

This function introduces a new category of mechanism of type CKF_TICKET which has value (CKF_EXTENSION | 0x40000000).

The table below lists the Ticket Mechanisms:

Mechanism	Description
CKM_SET_ATTRIBUTES	A mechanism to specify attribute changes for an object. It is used to extend the usage limit on a key.

CT_SetHsmDead

```
CK_DEFINE_FUNCTION(CK_RV, CT_SetHsmDead)(
  CK_ULONG hsmIDx,
  CK_BBOOL bDisable
);
```

This function can be used by an application to simulate the behavior of the WLD or HA system when an HSM fails. See also CT_GetHSMId

Return Values:

CKR_OK: Successfull.

CKR_ARGUMENTS_BAD: The supplied hsmID is invalid.

CKR_FUNCTION_NOT_SUPPORTED: The library is not in WLD mode

This function is an SafeNet extension to PKCS #11.

CT_GetHSMId

```
CK_DEFINE_FUNCTION(CK_RV, CT_GetHSMId)(
  CK_SESSION_HANDLE hSession,
  CK_ULONG_PTR pHsmid
  );
```

This function can be used to identify the HSM that a particular WLD or HA session has been assigned to.

Return Values:

CKR_OK: Successfull.

CKR_ARGUMENTS_BAD: The supplied pHsmID is NULL.

CKR_FUNCTION_NOT_SUPPORTED: The library is not in WLD mode

This function is an SafeNet extension to PKCS #11.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 11
CTUTIL.H FUNCTIONALITY REFERENCE

Overview

The ProtectToolkit C Software Development Kit offers a number of extended API libraries with functionality that is extended to that of the standard PKCS#11 function set.

The following additional features do not form part of the standard PKCS#11 functionality, but are provided by SafeNet as part of the ProtectToolkit C API within the CTLUTIL.H library.

BuildDhKeyPair

Synopsis

```
CK_RV BuildDhKeyPair(
 CK_SESSION_HANDLE hSession,
 char * txt,
 int tok,
 int priv,
 CK_OBJECT_HANDLE * phPub,
 CK_OBJECT_HANDLE * phPri,
 char * prime,
 char * base,
 char * pub,
 char * pri);
```

Description

Create a DH key pair given the required components.

Parameters

hSession	Open session handle
txt	Optional label
tok	1 for a Token object, 0 for Session object
priv	1 for Private object, 0 for Public object
phPub	Reference to object handle to hold created public key
phPri	Reference to object handle to hold created private key
prime	Prime
base	Base
pub	Public key value
pri	Private key value

On successful return

***phPub** — handle to newly created public key
***phPri** — handle to newly created private key

In addition to the Public key attributes set via the parameters, the following are set:

```
CKA_CLASS CKO_PUBLIC_KEY
 CKA_KEY_TYPE CKK_DH
 CKA_EXTRACTABLE TRUE
```

In addition to the Private key attributes set via the parameters, the following are set:

```
CKA_CLASS CKO_PRIVATE_KEY
 CKA_KEY_TYPE CKK_DH
 CKA_EXTRACTABLE TRUE
```

BuildDsaKeyPair

Synopsis

```
CK_RV BuildDsaKeyPair(
 CK_SESSION_HANDLE hSession,
 char * txt,
 int tok,
 int priv,
 CK_OBJECT_HANDLE * phPub,
 CK_OBJECT_HANDLE * phPri,
 char * prime,
 char * subprime,
 char * base,
 char * pub,
 char * pri);
```

Description

Create DSA key pair given required components.

Parameters

hSession	Open session handle
txt	Optional label
tok	1 for a Token object, 0 for Session object
priv	1 for Private object, 0 for Public object
phPub	Reference to object handle to hold created public key
phPri	Reference to object handle to hold created private key
prime	Prime
subprime	SubPrime
base	Base
pub	Public key value
pri	Private key value

On successful return

*phPub handle to newly created public key
*phPri handle to newly created private key

In addition to the Public key attributes set via the parameters, the following are set:

```
CKA_CLASS CKO_PUBLIC_KEY
 CKA_KEY_TYPE CKK_DSA
 CKA_EXTRACTABLE TRUE
```

In addition to the Private key attributes set via the parameters, the following are set:

```
CKA_CLASS CKO_PRIVATE_KEY
 CKA_KEY_TYPE CKK_DSA
 CKA_EXTRACTABLE TRUE
```

BuildRsaCrtKeyPair

Synopsis

```
CK_RV BuildRsaCrtKeyPair(
 CK_SESSION_HANDLE hSession,
 char * txt,
 int tok,
 int priv,
 CK_OBJECT_HANDLE * phPub,
 CK_OBJECT_HANDLE * phPri,
 char * modulusStr,
 char * pubExpStr,
 char * priExpStr,
 char * priPStr,
 char * priQStr,
 char * priE1Str,
 char * priE2Str,
 char * priUStr);
```

Description

Create an RSA key pair given the modulus and exponents, as well as the additional arguments used in Chinese Remainder Theorem processing. If the values for P, Q, E1, E2 and U are not specified, a normal RSA key pair is created.

Parameters

hSession	Open session handle
txt	Optional label
tok	1 for a Token object, 0 for Session object
priv	1 for Private object, 0 for Public object
phPub	Reference to object handle to hold created public key
phPri	Reference to object handle to hold created private key
modulusStr	Key modulus
pubExpStr	Public key exponent
priExpStr	Private key exponent

priPStr	Optional Private key Prime1
priQStr	Optional (optionality set by priPStr) Private key Prime2
priE1Str	Optional (optionality set by priPStr) Private key Exponent1
priE2Str	Optional (optionality set by priPStr) Private key Exponent2
priUStr	Optional (optionality set by priPStr) Private key Coefficient

On successful return

***phPub** — handle to newly created public key

***phPri** — handle to newly created private key

In addition to the Public key attributes set via the parameters, the following are set:

```
CKA_CLASS CKO_PUBLIC_KEY CKA_KEY_TYPE CKK_RSA CKA_VERIFY TRUE CKA_SIGN
 FALSE CKA_DECRYPT FALSE CKA_ENCRYPT TRUE CKA_EXTRACTABLE TRUE CKA_WRAP
 FALSE
```

In addition to the Private key attributes set via the parameters, the following are set:

```
CKA_CLASS CKO_PRIVATE_KEY CKA_KEY_TYPE CKK_RSA CKA_VERIFY FALSE CKA_SIGN
 TRUE CKA_DECRYPT TRUE CKA_ENCRYPT FALSE CKA_EXTRACTABLE TRUE CKA_WRAP
 FALSE
```

BuildRsaKeyPair

Synopsis

```
CK_RV BuildRsaKeyPair(
 CK_SESSION_HANDLE hSession,
 char * txt,
 int tok,
 int priv,
 CK_OBJECT_HANDLE * phPub,
 CK_OBJECT_HANDLE * phPri,
 char * modulusStr,
 char * pubExponentStr,
 char * priExponentStr);
```

Description

Create an RSA key pair given the modulus and exponents.

Parameters

hSession	Open session handle
txt	Optional label
tok	1 for a Token object, 0 for Session object
priv	1 for Private object, 0 for Public object
phPub	Reference to object handle to hold created public key
phPri	Reference to object handle to hold created private key
modulusStr	Key modulus
pubExponentStr	Public key exponent
priExponentStr	Private key exponent

On successful return

***phPub** — handle to newly created public key

***phPri** — handle to newly created private key

In addition to the Public key attributes set via the parameters, the following are set:

```
CKA_CLASS CKO_PUBLIC_KEY CKA_KEY_TYPE CKK_RSA CKA_VERIFY TRUE CKA_SIGN
 FALSE CKA_DECRYPT FALSE CKA_ENCRYPT TRUE CKA_EXTRACTABLE TRUE CKA_WRAP
 FALSE
```

In addition to the Private key attributes set via the parameters, the following are set:

```
CKA_CLASS CKO_PRIVATE_KEY CKA_KEY_TYPE CKK_RSA CKA_VERIFY FALSE CKA_SIGN
 TRUE CKA_DECRYPT TRUE CKA_ENCRYPT FALSE CKA_EXTRACTABLE TRUE CKA_WRAP
 FALSE
```

calcKvc

Synopsis

```
CK_RV calcKvc(
 CK_SESSION_HANDLE hSession,
 CK_OBJECT_HANDLE hKey,
 unsigned char * kvc,
 int kvclen,
 int * pkvclen);
```

Description

Calculate and return an AS2805 KVC for a key. The key must be capable of doing an encryption operation using the mechanism determined from the key type (see mechFromKt) for this to succeed. Note that mechanism parameters are set to NULL.

NOTE: The CKA_CHECK_VALUE attribute can be used to get the KVC of a key that does not support the encryption operation.

Parameters

hSession	Open session handle
hKey	Handle to the key to use for the encryption
kvc	Buffer to hold the encryption result
kvclen	Total number of bytes referenced by kvc
pkvclen	Reference to int to hold number of bytes copied into kvc

On successful return

kvc — holds the encryption result
***pkvclen** — number of bytes copied into kvc

If kvclen is smaller than the encryption result, then only kvclen bytes are copied into kvc.

calcKvcMech

Synopsis

```
CK_RV calcKvcMech(
  CK_SESSION_HANDLE hSession,
  CK_OBJECT_HANDLE hKey,
  CK_MECHANISM_TYPE mt,
  unsigned char * kvc,
  int kvclen,
  int * pkvclen);
```

Description

Calculate and return an AS2805 KVC for a key. The key must be capable of doing an encryption operation using the supplied mechanism for this to succeed. Note that mechanism parameters are set to NULL.

NOTE: the CKA_CHECK_VALUE attribute can be used to get the KVC of a key that does not support the encryption operation.

Parameters

hSession	Open session handle
hKey	Handle to the key to use for the encryption
mt	Encryption mechanism to use
kvc	Buffer to hold the encryption result
kvclen	Total number of bytes referenced by kvc
pkvclen	Reference to int to hold number of bytes copied into kvc

On successful return

kvc — holds the encryption result
***pkvclen** — number of bytes copied into kvc

If kvclen is smaller than the encryption result, then only kvclen bytes are copied into kvc.

cDump

Synopsis

```
int cDump(char * title,unsigned char * buf,unsigned int len);
```

Description

Dump buf contents in hex via printf.

Parameters

title	Heading
buf	Bytes to dump
len	Number of bytes to dump

CreateDesKey

Synopsis

```
CK_RV CreateDesKey(
 CK_SESSION_HANDLE hSession,
 char * txt,
 int tok,
 int priv,
 CK_BYTE * keyValue,
 int len,
 CK_OBJECT_HANDLE * phKey);
```

Description

Create a secret key object, and set the key type to `CKK_DES`, `CKK_DES2`or `CKK_DES3` (based on len).

Parameters

hSession	Open session handle
txt	Optional label
tok	1 for a Token object, 0 for Session object
priv	1 for private object, 0 for public object
keyValue	Key value
len	Length of key value
phKey	Reference to object handle to hold created key

On successful return

***phKey** — handle to newly created key

In addition to the key attributes set via the parameters, the following are set:

```
CKA_CLASS CKO_SECRET_KEY
 CKA_KEY_TYE CKK_DES, CKK_DES2 OR CKK_DES3
 CKA_ID "ID"
 CKA_DERIVE TRUE
 CKA_EXTRACTABLE TRUE
 CKA_UNWRAP TRUE
 CKA_WRAP FALSE
```

CreateSecretKey

Synopsis

```
CK_RV CreateSecretKey(
 CK_SESSION_HANDLE hSession,
 char * txt,
 int tok,
 int priv,
 CK_KEY_TYPE kt,
 CK_BYTE * keyValue,
 int len,
 CK_OBJECT_HANDLE * phKey);
```

Description

Create a secret key object.

Parameters

hSession	Open session handle
txt	Optional label
tok	1 for a Token object, 0 for Session object
priv	1 for private object, 0 for public object
kt	Key type
keyValue	Key value
len	Length of key value
phKey	Reference to object handle to hold created key

On successful return

***phKey** — handle to newly created key

In addition to the key attributes set via the parameters, the following are set:

```
CKA_CLASS CKO_SECRET_KEY
 CKA_ID "ID"
 CKA_DERIVE TRUE
 CKA_EXTRACTABLE TRUE
 CKA_UNWRAP TRUE
 CKA_WRAP FALSE
```

CT_AttrToString

Synopsis

```
CK_RV CT_AttrToString(CK_ATTRIBUTE_PTR pAttr,char* pStringVal,CK_SIZE*
 pStringValLen);
```

Description

Get the value of the given attribute as a printable string

Parameters

param pAttr	pointer to the attribute whose value is to be stringified
pStringVal	location to hold the value as a string (if NULL, the length required to hold the string is still copied into pStringValLen)
pStringValLen	location to store the length of the value as a string (if pStringVal was supplied, this contains the number of bytes copied into the buffer or, if pStringVal is NULL, this contains the required size of the buffer to hold the value as a string).

On successful return

* **pStringVal** — pointer to the returned string value
* **pStringValLen** — length of the string

CT_CreateObject

Synopsis

```
CK_RV CT_CreateObject(
  CK_SESSION_HANDLE hSession,
  CK_OBJECT_CLASS cl,
  char * name,
  CK_OBJECT_HANDLE * phObj);
```

Description

Create a private token object of the specified class with the defined label.

Parameters

hSession	Open session on the slot to create the object in
cl	Class of the object
name	Label of the object
phObj	Reference to object handle to hold created object

On successful return

***phObj** — handle to the newly created object

CT_CreatePublicObject

Synopsis

```
CK_RV CT_CreatePublicObject(
  CK_SESSION_HANDLE hSession,
  CK_OBJECT_CLASS cl,
  char * name,
  CK_OBJECT_HANDLE * phObj);
```

Description

Create a public token object of the specified class with the defined label.

Parameters

hSession	Open session on the slot to create the object in
cl	Class of the object
name	Label of the object
phObj	Reference to object handle to hold created object

On successful return

***phObj** — handle to the newly created object

CT_Create_Set_Attributes_Ticket_Info()

Synopsis

```
CK_RV CT_Create_Set_Attributes_Ticket_Info(
   /* specify the target */
     CK_MECHANISM_TYPE objectDigestAlg, /* digest alg */
     unsigned char * objectDigest, /* digest of target object */
      unsigned int objectDigestLen,
     /* specify issuer */
     char * issuerRDN,       /* may be NULL or
                              * DER of DistName or
                              * Common Name string or
                              * RDN Seq string (CN=Fred+C=USA) */
     unsigned int issuerRDNLen,

   /* ticket details */
     CK_MECHANISM_TYPE signatureAlg, /* signature alg */
     unsigned long sno,      /* Attrib Cert serial number */
     char * notBefore,       /* YYYYMMDD string */
     char * notAfter,        /* YYYYMMDD string */

   /* attributes on key to modify */
     unsigned long * limit,  /* NULL if no CKA_USAGE_LIMIT */
     char * start,           /* NULL if no CKA_START_DATE */
     char * end,             /* NULL if no CKA_END_DATE */
     char * cert,            /* NULL if no CKA_ADMIN_CERT */
     unsigned int certLen,

   /* output */
      void * pTicketInfo,    /* OUT new unsigned ticket returned here */
      unsigned int* puiTicketLen; /* IN/OUT pTicketInfo buffer length */
);
```

Description

The function creates an unsigned CKM_SET_ATTRIBUTES ticket.

The function supports length prediction.

See CT_Create_Set_Attributes_Ticket.

CT_Create_Set_Attributes_Ticket()

Synopsis

```
CK_RV CT_Create_Set_Attributes_Ticket(
 void * pTicketInfo,           /* IN unsigned ticket */
 unsigned int uiTicketInfoLen; /* IN pTicketInfo buffer length */

 CK_MECHANISM_TYPE signatureAlg, /* signature alg */
 unsigned char * pSignature,   /* signature of pTicketData */
 unsigned int uiSigLen;        /* IN pSignature buffer length */

 void * pTicketData,           /* OUT new unsigned ticket returned here
*/
 unsigned int * puiTicketLen;  /* IN/OUT pTicketData buffer length */
);
```

Description

The function combines the AttributeCertificateInfo DER encoding returned from the CT_Create_Set_Attributes_Ticket_Info function with a digital signature to form the DER encoded AttributeCertificate that may be passed to a CT_PresentTicket function using the CKM_SET_ATTRIBUTES mechanism.

CT_DerEncodeNamedCurve

Synopsis

```
CK_RV CT_DerEncodeNamedCurve(
 CK_BYTE_PTR buf,
 CK_SIZE_PTR len,
 const char *name);
```

Description

Helper function to provide the DER encoding of a supported named curve. This function is typically used to populate the `CKA_EC_PARAMS` attribute of the template used during EC key pair generation.

Supported curve names are:

Name	**OID**
c2tnb191v1	{ iso(1) member-body(2) US(840) x9-62(10045) curves(3) characteristicTwo(0) c2tnb191v1(5) }
P-192 (also known as “prime192v1 ” “secp192r1”)	{ iso(1) member-body(2) US(840) x9-62(10045) curves(3) prime(1) prime192v1(1) }
P-224 (also known as “secp224r1”)	{ iso(1) identified-organization(3) Certicom(132) certicom_ellipticCurve(0) secp224r1(33) }
P-256 (also known as (“prime256v1 ” “secp256r1”)	{ iso(1) member-body(2) US(840) x9-62(10045) curves(3) prime(1) prime256v1(7) }
P-384 (also known as “secp384r1”)	{ iso(1) identified-organization(3) Certicom(132) certicom_ellipticCurve(0) secp384r1(34) }
P-521 (also known as “secp521r1”)	{ iso(1) identified-organization(3) Certicom(132) certicom_ellipticCurve(0) secp521r1(35) }
c2tnb191v1e (Non FIPS curve)	{ iso(1) member-body(2) US(840) x9-62(10045) curves(3) characteristicTwo(0) c2tnb191v1e (15) }

Parameters

buf	Buffer to hold the DER encoding
len	*len is total number of bytes referenced by buf
name	String name of the curve to get the encoding for

On successful return

buf — contains a string.

Example: “hh:mm:ss DD/MM/YYYY“ *len Number of bytes copied to buf

To determine the encoding length, pass in `NULL` for buf and check the resulting value of `*len`.

CT_GetAuthChallenge

Synopsis

```
CK_DEFINE_FUNCTION(CK_RV, CT_GetAuthChallenge)(

    CK_SESSION_HANDLE hSession,

    CK_BYTE_PTR pChallenge,

    CK_ULONG_PTR pulChallengeLen

);
```

Description

This function is a PTK C extension to PKCS#11provided with the PTK C SDK as a host side library function.

The function requests the HSM to generate a random 16 byte challenge value and to return the challenge to the calling application. The function uses the CKH_VD_USER object to fetch the Challenge.

The Application can use authentication data (pin) to create a Response from the challenge. See description of CT_Gen_AUTH_Response**Error! Reference source not found.** for more details.

The Response can be used with the C_Login function to authenticate the user to the Token. See description of C_Login for more details.

CT_GetObjectDigest

Synopsis

```
CK_RV CT_GetObjectDigest(
       CK_SESSION_HANDLE hSession,  /* IN */
       CK_OBJECT_HANDLE hObject,    /* IN */
       CK_MECHANISM_PTR pDigestMech,/* IN */

       CK_BYTE_PTR * ppDigest,  /* OUT returned buffer must be freed */
       CK_ULONG    * pulDigest  /* OUT length of returned buffer */
              );
```

Description

Compute the object digest as used by SET Attributes Ticket to identify the target object.

CT_GetECCDomainParameters

#include "ctutil.h"

	Windows	UNIX
Library	ctutil.lib	Libctutil.a

```
CK_RV CT_GetECCDomainParameters(

CK_SESSION_HANDLE hSession,

CK_ATTRIBUTE_PTR  attr,

const char *name)
```

Description

This function returns the DER encoded Domain Parameters for a curve specified by name.

First the CT_DerEncodeNamedCurve function is used to see if the curve is known to the HSM. If not, then this function looks up the appropriate Domain Parameter object in the token indicated by hSession.

Parameters

param hSession	Session where Domain Parameter object can be found
param attr	ptr to attribute structure to hold encoding of domain parameters (length prediction supported)
param name	Label of Domain Parameter object or known named curve
return	Cryptoki error returned, CKR_OK if successful

CT_GetObjectDigestFromParts

Synopsis

```
CK_RV CT_GetObjectDigestFromParts(
         CK_SESSION_HANDLE hSession,  /* IN */
         CK_MECHANISM_PTR pDigestMech,/* IN */
         char * tokenSerialNumber,    /* IN */
         char * tokenLabel,           /* IN */
         char * objLabel,             /* IN */
         CK_BYTE_PTR  objID,          /* IN */
         CK_ULONG     objIDlen,       /* IN */

         CK_BYTE_PTR * ppDigest,   /* OUT returned buffer
                                      (must be freed by caller) */
         CK_ULONG    * pulDigest   /* OUT length of returned buffer */
                );
```

Description

Compute the object digest as used by SET Attributes Ticket to identify the target object by using parts.

See also CT_GetObjectDigest.

CT_GetTmpPin

Synopsis

```
CK_DEFINE_FUNCTION(CK_RV, CT_GetTmpPin)(

    CK_SESSION_HANDLE hSession,

    CK_BYTE_PTR pPin,

    CK_ULONG_PTR pulPinLen

);
```

Description

This function is a PTK C extension to PKCS#11provided with the PTK C SDK as a host side library function.

The function requests the HSM to generate a random Temporary Pin value and to return the pin to the calling application. The function uses the CKH_VD_USER object to fetch the Pin.

A User or SO must be already logged on or this function will fail with error CKR_USER_NOT_LOGGED_ON.

The Application can pass this Temporary Pin to another process which can then use it to authenticate to the HSM (as the same user type only).

The Temporary Pin can be passed to the C_Login function to authenticate the user to the Token. See description of C_Login for more details.

CT_ErrorString

Synopsis

```
CK_RV C_ErrorString(
 CK_RV ret,
 char * errstr,
 unsigned int len);
```

Description

Get a printable string representation of a Cryptoki error code.

Parameters

ret	Cryptoki error code
errstr	buffer to hold the printable string
len	number of characters referenced by errstr

On successful return

errstr — contains the printable string, or as much as will fit

CT_GetECKeySize

Synopsis

```
CK_RV CT_GetECKeySize(const CK_ATTRIBUTE_PTR ecParam,CK_SIZE_PTR size);
```

Description

Helper function to return key size (in bits) for a given EC parameter

Parameters

ecParam	handle that points to EC parameter
size	returned key size

On successful return

size — pointer to the value of key size

CT_MakeObjectNonModifiable

Synopsis

```
CK_RV CT_MakeObjectNonModifiable(
        CK_SESSION_HANDLE hSession,   /* IN */
        CK_OBJECT_HANDLE  hObj,       /* IN */
        CK_OBJECT_HANDLE *phObj       /* OUT (may be NULL) */
                                );
```

Description

Change an object CKA_MODIFIABLE attribute from true to False.

This involves copying the object - so the handle of the object will change.

The original object is deleted.

CT_OpenObject

Synopsis

```
CK_RV CT_OpenObject(
 CK_SESSION_HANDLE hSession,
 CK_OBJECT_CLASS cl,
 char * name,
 CK_OBJECT_HANDLE * phObj);
```

Description

Get a handle to an object with the specified class and label. This function returns the first object satisfying the criteria.

Parameters

hSession	open session on the slot containing the object
cl	class of the object
name	label of the object
phObj	reference to object handle to hold opened object

On successful return

***phObj** — handle to the opened object

CT_ReadObject

Synopsis

```
CK_RV CT_ReadObject(
 CK_SESSION_HANDLE hSession,
 CK_OBJECT_HANDLE hObj,
 unsigned char * buf,
 unsigned int len,
 unsigned int * pbr);
```

Description

Get the value of an object.

Parameters

hSession	open session on the slot containing the object
hObj	object whose value is to be returned
buf	buffer to hold the object value
len	total number of bytes referenced by buf
pbr	reference to int to hold number of bytes copied into buf

On successful return

buf — contains the object value
***pbr** — number of bytes copied into buf

If buf is too small to hold the attribute value (that is, len is < attribute value length), then CKR_ATTRIBUTE_TYPE_INVALID is returned.

To determine the attribute value length, pass in 0 for len, and check the resulting value of *pbr.

CT_RenameObject

Synopsis

```
CK_RV CT_RenameObject(
 CK_SESSION_HANDLE hSession,
 CK_OBJECT_CLASS cl,
 char * oldName,
 char * newName);
```

Description

Change the label of the object with the specified class and label.

Parameters

hSession	open session on the slot containing the object
cl	class of the object
oldName	current label of the object
newName	new label for the object

CT_SetCKDateStrFromTime

Synopsis

```
void  CT_SetCKDateStrFromTime(
 char pd[9], /* OUT - pointer to a buffer at least 9 bytes*/
 time_t *t);  /* IN - time value to convert */
```

Description

convert time_t structure to the DATE format used by CT_SetLimitsAttributes and CT_Create_Set_Attributes_Ticket_Info

CT_Structure_To_Armor

Synopsis

```
CK_RV CT_Structure_To_Armor(

    char *    pLabel,           /* IN Armor label (string) */
    char *    pComment,         /* optional comment string */
    CK_VOID_PTR    pData,       /* IN data to armor */
    CK_ULONG ulDataLen          /* IN length of data */

    CK_BYTE_PTR *  pArmor,      /* OUT Armored structure created
                                       (free after use) */
    CK_ULONG_PTR   pulArmorLen /* IN/OUT pArmor buffer length */
);
```

Description

Armoring is the term used in PGP and MIME to describe the formatting of binary data such that it can be unambiguously embedded in a printable document such as an email.

The Base 64 encoding method is used to make binary data printable and the encoding is clearly marked with BEGIN and END statements.

The function formats an arbitary structure – such as a ticket - into an Armored (printable format).
The result is returned as a buffer that the caller must free after use.

Example:

If Armoring the binary data 01h 23h 45h 67h 89h abh cdh efh with the label “SETATTRIBUTE TICKET” and the comment “This is a trial certificate\n”.

You get:

This is a trial certificate

-----BEGIN SETATTRIBUTE TICKET-----

ASNFZ4mrze8=

-----END SETATTRIBUTE TICKET-----

CT_Structure_From_Armor

Synopsis

```
CK_RV CT_Structure_From_Armor (

    Char *    pLabel,           /* IN Armor label (string) */
    CK_BYTE_PTR    pArmor,           /* IN Armored structure */
    CK_ULONG ulArmorLen  /* IN pArmor buffer length */

    CK_VOID_PTR *  pData,            /* OUT extracted structure */
    CK_ULONG_PTR   pulDataLen  /* OUT *pData buffer length */
);
```

Description

Armoring is the term used in PGP and MIME to describe the formatting of binary data such that it can be unambiguously embedded in a printable document such as an email.

The function extracts a data structure from an Armored (printable format) buffer.
The result is returned as a buffer that the caller must free after use.

CT_SetLimitsAttributes

Synopsis

```
CK_RV CT_SetLimitsAttributes(
     CK_SESSION_HANDLE hSession, /* IN */
     CK_OBJECT_HANDLE  hObj,     /* IN */
     CK_VOID_PTR pCertData,    /* IN - optional CKA_ADMIN_CERT value */
     CK_ULONG    ulCertDataLen, /* IN - length of pCertData */
     CK_ULONG    * usage_limit,  /* IN - optional CKA_USAGE_LIMIT */
     CK_ULONG    * usage_count,  /* IN - optional CKA_USAGE_COUNT */
     char *        start_date,   /* IN - optional CKA_START_DATE */
     char *        end_date      /* IN - optional CKA_END_DATE */
               );
```

Description

Apply limit attributes to an object. The optional inputs maybe set to NULL to indicte that that Attributes should not be set.

NOTE: Object should have CKA_MODIFIABLE-false for this function to work.

CT_WriteObject

Synopsis

```
CK_RV CT_WriteObject(
  CK_SESSION_HANDLE hSession,
  CK_OBJECT_HANDLE hObj,
  const unsigned char * buf,
  unsigned int len,
  unsigned int * pbr);
```

Description

Set the value of an object.

Parameters

hSession	Open session on the slot containing the object
hObj	Object whose value is to be set
buf	Value of the object to set
len	:ength of buf
pbr	Reference to int to hold number of bytes copied from buf

On successful return

***pbr** — set to equal len

DateConvertGmtToLocal

Synopsis

```
DateConvertGmtToLocal(char * fmt,const char * ts);
```

Description

Converts a GMT date string of the format YYYYMMDDhhmmssxx into the Local Time format "DD/MM/YYYY hh:mm:ss (TimeZone)".

Parameters

fmt	pointer to the buffer that holds the converted value
ts	GMT date string

On Successful Return

***fmt** — pointer to the buffer that holds the converted value

DateConvert

Synopsis

```
void DateConvert(
 char * fmt,
 const char * ts);
```

Description

Convert “YYYYMMDDhhmmss00” to “hh:mm:ss DD/MM/YYYY“.

Parameters

fmt	Destination string
ts	Source string

On Successful Return

fmt — contains a string like “hh:mm:ss DD/MM/YYYY“

DumpAttributes

Synopsis

```
void DumpAttributes(CK_ATTRIBUTE * na,CK_COUNT attrCount);
```

Description

Dumps attribute details via logtrace.

Parameters

na	Array of attributes to dump
attrCount	Number of attributes in na

DumpDHKeyPair

Synopsis

```
CK_RV DumpDHKeyPair(
  int cStyle,
  CK_SESSION_HANDLE hSession,
  CK_OBJECT_HANDLE hPub,
  CK_OBJECT_HANDLE hPri);
```

Description

Dump DH key pair details via printf.

Parameters

cStyle	1 for a form which can be included in C code, 0 for standard dump
hSession	Open session handle
hPub	Handle to public key
hPri	Handle to private key

DumpDSAKeyPair

Synopsis

```
CK_RV DumpDSAKeyPair(
  int cStyle,
  CK_SESSION_HANDLE hSession,
  CK_OBJECT_HANDLE hPub,
  CK_OBJECT_HANDLE hPri);
```

Description

Dump DSA key pair details via printf.

Parameters

cStyle	1 for a form which can be included in C code, 0 for standard dump
hSession	Open session handle
hPub	Handle to public key
hPri	Handle to private key

DumpRSAKeyPair

Synopsis

```
CK_RV DumpRSAKeyPair(
  int cStyle,
  CK_SESSION_HANDLE hSession,
  CK_OBJECT_HANDLE hPub,
  CK_OBJECT_HANDLE hPri);
```

Description

Dump RSA key pair details via printf.

Parameters

cStyle	1 for a form which can be included in C code, 0 for standard dump
hSession	Open session handle
hPub	Handle to public key
hPri	Handle to private key

FindAttribute

Synopsis

```
CK_ATTRIBUTE * FindAttribute(
 CK_ATTRIBUTE_TYPE attrType,
 const CK_ATTRIBUTE * attr,
 CK_COUNT attrCount);
```

Description

Find the first attribute of the specified type in an attribute template.

Parameters

attrType	Type of the attribute to locate
attr	Attribute temple (that is, array of CK_ATTRIBUTE)
attrCount	Number of attributes referenced by attr

On Successful Return

Return a pointer to the attribute of the specified type.

FindKeyFromName

Synopsis

```
CK_RV FindKeyFromName(
 const char * keyName,
 CK_OBJECT_CLASS cl,
 CK_SLOT_ID * phSlot,
 CK_SESSION_HANDLE * phSession,
 CK_OBJECT_HANDLE * phKey);
```

Description

Find the key with a given class and label within the specified token, and open a session to this token.

Parameter

keyName	String identifying the key to locate format "token(pin)/key" or "token/key" token name of the Token containing the key pin optional user pin key label of the key in the Token
cl	Class of the object
phSlot	Reference to slot id to hold located slot id
phSession	Reference to session handle to hold opened session
phKey	Reference to object handle to hold located key handle

On Successful Return

***phSlot** — slot holding the key
***phSession** — open session handle
***phKey** — handle to the key object

FindTokenFromName

Synopsis

```
CK_RV FindTokenFromName(
 char * label,
 CK_SLOT_ID * pslotID);
```

Description

Find a token with the specified label and return the corresponding slot id.

Parameters

label	String identifying Token to find
pslotID	Reference to slot id to hold located slot id

On Successful Return

***pslotID** — slot which contains the Token

GenerateDhKeyPair

Synopsis

```
CK_RV GenerateDhKeyPair(
 CK_SESSION_HANDLE hSession,
 char * txt,
 int ftok,
 int priv,
 int param,
 CK_SIZE valueBits,
 CK_OBJECT_HANDLE * phPublicKey,
 CK_OBJECT_HANDLE * phPrivateKey);
```

Description

Generate a DH key pair.

Parameters

hSession	Open session handle
txt	Optional label
ftok	1 for a Token object, 0 for Session object
priv	1 for private object, 0 for public object
param	Not used
valueBits	Number of prime bits
phPublicKey	Reference to object handle to hold created public key
phPrivateKey	Reference to object handle to hold created private key

On Successful Return

***phPublicKey** — handle to newly created public key
***phPrivateKey** — handle to newly created private key

In addition to the Public key attributes set via the parameters, the following are set:

```
CKA_CLASS CKO_PUBLIC_KEY
 CKA_KEY_TYPE CKK_DH
 CKA_VERIFY TRUE
 CKA_EXTRACTABLE TRUE
```

In addition to the Private key attributes set via the parameters, the following are set:

```
CKA_CLASS CKO_PRIVATE_KEY
 CKA_KEY_TYPE CKK_DH
 CKA_SUBJECT_STR "SUBJECT"
 CKA_ID 123
 CKA_SENSITIVE TRUE
 CKA_SIGN TRUE
 CKA_EXTRACTABLE TRUE
```

GenerateDsaKeyPair

Synopsis

```
CK_RV GenerateDsaKeyPair(
 CK_SESSION_HANDLE hSession,
 char * txt,
 int ftok,
 int priv,
 int param,
 CK_SIZE valueBits,
 CK_OBJECT_HANDLE * phPublicKey,
 CK_OBJECT_HANDLE * phPrivateKey);
```

Description

Generate DSA key pair.

Parameters

hSession	Open session handle
txt	Optional label
ftok	1 for a Token object, 0 for Session object
priv	1 for private object, 0 for public object
param	1 to generate new DSA parameters, 0 to use defaults (see below)
valueBits	Number of bits in Prime
phPublicKey	Reference to object handle to hold created public key
phPrivateKey	Reference to object handle to hold created private key

On Successful Return

***phPublicKey** — handle to newly created public key
***phPrivateKey** — handle to newly created private key

In addition to the Public key attributes set via the parameters, the following are set:

```
CKA_CLASS CKO_PUBLIC_KEY
 CKA_KEY_TYPE CKK_DSA
 CKA_VERIFY TRUE
 CKA_EXTRACTABLE TRUE
```

In addition to the Private key attributes set via the parameters, the following are set:

```
CKA_CLASS CKO_PRIVATE_KEY
 CKA_KEY_TYPE CKK_DSA
 CKA_SUBJECT_STR "SUBJECT"
 CKA_ID 123
 CKA_SENSITIVE TRUE
 CKA_SIGN TRUE
 CKA_EXTRACTABLE TRUE
```

The default values for the DSA parameters are:

512 P = fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae01f35b91a47e6df6
3413c5e12ed0899bcd132acd50d99151bdc43ee737592e17

512 Q = 962eddcc369cba8ebb260ee6b6a126d9346e38c5

512 G = 678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9e35030b71fd73da
179069b32e2935630e1c2062354d0da20a6c416e50be794ca4

1024 P = e2662c8df32db56309ccb7f8f419e73263c55c1a89954fa68d85d8b09c720618532bd05dc0994b
e245526367b08888f4ef07bb0977ac6aa3c4653f6d70151027fb73a9d7f99e63a63ea5c89de1b1
5b35ecc0beae18a89ee4aac0f75b2c364026c3b6ef8ad13cdd6886d93f9b86c71cb2537b449643
4412033ab3002de749d963

1024 Q = fd5274d166045c96e5f180ab181ccde55804a9c7

1024 G = 0c8392be4b9c222526fc2160864b117b7c8d9e3bec9faa1f7e4d8cfcecbfbf521a0aca11620aaaf0
f847068e8f6c936438bd482cd2d6ee2bbac519b63f5809c412dbd39664fa4e05567fc9bf01f83e3
f816aa945304f31e832a243e138b7b776bb519411d5669b4c6e38c840c2b9ae195f84f04b8b508
7271613c12d938720cc

GenerateRsaKeyPair

Synopsis

```
CK_RV GenerateRsaKeyPair(
 CK_SESSION_HANDLE hSession,
 char * txt,
 int ftok,
 int priv,
 CK_SIZE modulusBits,
 int expType,
 CK_OBJECT_HANDLE * phPublicKey,
 CK_OBJECT_HANDLE * phPrivateKey);
```

Description

Generate an RSA key pair.

Parameters

hSession	Open session handle
txt	Optional label
ftok	1 for a Token object, 0 for Session object
priv	1 for private object, 0 for public object
modulusBits	Size of modulus to generate
expType	0 for random exponent, 1 for Fermat 4 exponent (\x00010001), 2 for smallest valid exponent (3)
phPublicKey	Reference to object handle to hold created public key
phPrivateKey	Reference to object handle to hold created private key

On Successful Return

***phPublicKey** — handle to newly created public key
***phPrivateKey** — handle to newly created private key

In addition to the Public key attributes set via the parameters, the following are set:

```
CKA_CLASS CKO_PUBLIC_KEY
 CKA_KEY_TYPE CKK_RSA
 CKA_SUBJECT_STR "SUBJECT"
 CKA_ENCRYPT TRUE
 CKA_VERIFY TRUE
 CKA_WRAP FALSE
 CKA_EXTRACTABLE TRUE
```

In addition to the Private key attributes set via the parameters, the following are set:

```
CKA_CLASS CKO_PRIVATE_KEY CKA_KEY_TYPE CKK_RSA CKA_SUBJECT_STR "SUBJECT"
 CKA_ID 123 CKA_SENSITIVE TRUE CKA_DECRYPT TRUE CKA_SIGN TRUE CKA_UNWRAP
 FALSE CKA_EXTRACTABLE TRUE
```

GetAttr

Synopsis

```
CK_RV GetAttr(
  CK_SESSION_HANDLE hSession,
  CK_OBJECT_HANDLE obj,
  CK_ATTRIBUTE_TYPE type,
  CK_VOID_PTR buf,
  CK_SIZE len,
  CK_SIZE_PTR size);
```

Description

Get a single attribute of an object.

Parameters

hSession	Open session on the slot containing the object
obj	Object whose attribute is to be retrieved
type	Attribute to retrieve
buf	Buffer to hold the attribute value
len	Total number of bytes referenced by buf
size	Reference to CK_SIZE to hold the number of bytes copied into buf

On Successful Return

buf — contains the attribute value
***size** — number of bytes copied to buf

If buf is too small to hold the attribute value (that is, len is < attribute value length), then `CKR_ATTRIBUTE_TYPE_INVALID`is returned.

To determine the attribute value length, pass in 0 for len, and check the resulting value of *size.

GetDeviceError

Synopsis

```
CK_RV GetDeviceError( CK_SLOT_ID slotID,CK_NUMERIC *pDeviceError);
```

Description

Returns the device-error value for a given slot ID

Parameters

slotID	Slot to be queried
pDeviceError	Error code

On Successful Return

***pDeviceError** — returned error code

GetObjectCount

Synopsis

```
CK_RV GetObjectCount(
 CK_SLOT_ID slotID,
 unsigned int * pCount);
```

Description

Determine the number of objects on a token.

Parameters

slotID	Slot ID containing objects to count
pCount	Reference to int to hold number of objects

On Successful Return

***pCount** — number of objects

GetSecurityMode

Synopsis

```
CK_RV GetSecurityMode(CK_SLOT_ID inputSlotId,
 CK_SLOT_ID* pAdminSlotId,
 CK_FLAGS* pSecMode);
```

Description

Get the security mode for the slot id given by inputSlotID.

Parameters

inputSlotId	Slot ID to retrieve the security flags from
pAdminSlotId	Location to store the ID of the Admin Slot; Optional - ignored if NULL
pSecMode	Location to store the security mode

On Successful Return

* **pStringVal** — pointer to the returned string value
* **pStringValLen** — length of the string

GetSessionCount

Synopsis

```
CK_RV GetSessionCount(
 CK_SLOT_ID slotID,
 unsigned int * pSessionCount,
 unsigned int *prwSessionCount);
```

Description

Determine the number of sessions on a token

Parameters

slotID	Slot ID containing objects to coun
pSessionCount	Reference to int to hold the number of open session
prwSessionCount	Reference to int to hold the number of open RW session

On Successful Return

***pSessionCount** — number of open session
***prwSessionCount** — number of open RW session

GetTotalSessionCount

Synopsis

```
CK_RV GetTotalSessionCount(
 unsigned int *pSessionCount);
```

Description

Determine the total number of sessions open on all tokens on all adapters.

Parameters

pSessionCount	Reference to int to hold the number of open session

On Successful Return

***pSessionCount** — total number of open sessions

rmTrailSpace

Synopsis

```
void rmTrailSpace(
 char * txt,
 int len);
```

Description

Remove trailing spaces from a string.

Parameters

txt	String to process
len	Length of the string

On Successful Return

txt — string no longer has trailing spaces

SetAttr

Synopsis

```
CK_RV SetAttr(
 CK_SESSION_HANDLE hSession,
 CK_OBJECT_HANDLE obj,
 CK_ATTRIBUTE_TYPE type,
 const CK_VOID_PTR buf,
 CK_SIZE len);
```

Description

Set a single attribute of an object.

Parameters

hSession	Open session on the slot containing the object
obj	Object whose attribute is to be retrieved
type	Attribute to retrieve
buf	Contains the attribute value to set
len	Number of bytes referenced by buf

ShowSlot

Synopsis

```
CK_RV ShowSlot(
  CK_SLOT_ID slotID,
  int verbose);
```

Description

Dump slot details via printf.

Parameters

slotID	Slot to dump
verbose	0 for description and manufacturer, 1 for more details

ShowToken

Synopsis

```
CK_RV ShowToken(
  CK_SLOT_ID slotID,
  int verbose);
```

Description

Dump token details via printf.

Parameters

slotID	Slot containing Token to dump
verbose	0 for brief details, 1 for more details

strAttribute

Synopsis

```
char * strAttribute(
  CK_NUMERIC val);
```

Description

Given the numeric value of an attribute, return the string name.

Parameters

val	Numeric value of an attribute

strError

Synopsis

```
char * strError(
 CK_NUMERIC val);
```

Description

Given the numeric value of an error, return the string name.

Parameters

val	Numeric value of an error

strKeyType

Synopsis

```
char * strKeyType(
 CK_NUMERIC val);
```

Description

Given the numeric value of a key type, return the string name.

Parameters

val	Numeric value of a key type

strMechanism

Synopsis

```
char * strMechanism(
 CK_NUMERIC val);
```

Description

Given the numeric value of a mechanism, return the string name.

Parameters

val	Numeric value of a mechanism

strObjClass

Synopsis

```
char * strObjClass(
 CK_NUMERIC val);
```

Description

Given the numeric value of an object class, return the string name.

Parameters

val	Numeric value of an object class

strSesState

Synopsis

```
char * strSesState(
 CK_NUMERIC val);
```

Description

Given the numeric value of a session state, return the string name.

Parameters

val	Numeric value of a session state

TransferObject

Synopsis

```
CK_RV TransferObject(
 CK_SESSION_HANDLE sTo,
 CK_SESSION_HANDLE sFrom,
 CK_OBJECT_HANDLE hObj,
 CK_OBJECT_HANDLE * phObj,
 CK_ATTRIBUTE_PTR pTemplate,
 CK_COUNT ulCount);
```

Description

Copies an object from one Token to another.

Parameters

sTo	Open session handle on destination Token
sFrom	Open session handle on source Token
hObj	Handle to object to transfer
phObj	Reference to handle to hold new object
pTemplate	Specifies new values for some attributes of the new object
ulCount	Number of attributes in pTemplate

On Successful Return

***phObj** — handle to newly copied object

pTemplate — can only overwrite attributes which are ordinarily writeable.

This function tries the following methods to copy the object, in order:

- Using the `CKM_ENCODE_ATTRIBUTES` vendor defined key wrapping mechanism to transfer keys.
- Reading all the attributes of the existing object and creating a new object with them.

valAttribute

Synopsis

```
CK_NUMERIC valAttribute(
 const char * txt);
```

Description

Given the string name of an attribute, return the numeric value.

Parameters

txt	String name of an attribute

valError

Synopsis

```
CK_NUMERIC valError(
 const char * txt);
```

Description

Given the string name of an error, return the numeric value.

Parameters

txt	String name of an error

valKeyType

Synopsis

```
CK_NUMERIC valKeyType(
 const char * txt);
```

Description

Given the string name of a key type, return the numeric value.

Parameters

txt	String name of a key type

valMechanism

Synopsis

```
CK_NUMERIC valMechanism(
 const char * txt);
```

Description

Given the string name of a mechanism, return the numeric value.

Parameters

txt	String name of a mechanism

valObjClass

Synopsis

```
CK_NUMERIC valObjClass(
 const char * txt);
```

Description

Given the string name of an object class, return the numeric value.

Parameters

txt	String name of the object class

valSesState

Synopsis

```
CK_NUMERIC valSesState(
 const char * txt);
```

Description

Given the string name of a session state, return the numeric value.

Parameters

txt	String name of a session state

CHAPTER 12
CTEXTRA.H LIBRARY REFERENCE

Overview

The ProtectToolkit C Software Development Kit offers a number of extended API libraries with functionality that is extended to that of the standard PKCS#11 function set.

The following additional features do not form part of the standard PKCS#11 functionality, but are provided by SafeNet as part of the ProtectToolkit C API within the CTEXTRA.H library.

AddAttributeSets

Synopsis

```
CK_RV AddAttributeSets(TOK_ATTR_DATA ** pSum,const TOK_ATTR_DATA *
 base,const TOK_ATTR_DATA * user);
```

Description

Add two attribute sets being careful to drop duplicates. The 'base' attributes will override 'user' attributes where duplicates are concerned. Resulting set is located in *pSum.

Parameters

pSum	Reference to addition of base and user sets
base	Attribute set to add to user set
user	Attribute set to add to base set

On Successful Return

***pSum**—reference to a newly allocated attribute set resulting from the addition. This memory needs to be released via a call to *FreeAttributeSet*.

at_assign

Synopsis

```
CK_RV at_assign(
 CK_ATTRIBUTE * tgtNa,
 const CK_ATTRIBUTE * srcNa);
```

Description

Assign one attribute value to another. Attribute types and lengths have to match up.

Parameters

tgtNa	Target attribute
srcNa	Source attribute

To determine the length of tgtNa->pValue required, set tgtNa->pValue to NULL and check tgtNa->valueLen after invocation.

ConcatAttributeSets

Synopsis

```
CK_RV ConcatAttributeSets(
 TOK_ATTR_DATA * base,
 const TOK_ATTR_DATA * user);
```

Description

Append attributes from the user set to the base set. The 'base' attributes will override 'user' attributes where duplicates are concerned.

Parameters

base	Reference to attribute set to append to
user	Reference to attribute set to append

CopyAttribute

Synopsis

```
CK_ATTRIBUTE * CopyAttribute(
 CK_ATTRIBUTE_TYPE at,
 TOK_ATTR_DATA * tgt,
 const TOK_ATTR_DATA * src);
```

Description

Make a copy of an attribute from one attribute set to another. Only copy it if it is in 'src'. Overwrite it if it is in 'tgt'. Returns reference to the copied attribute in tgt attribute set.

Parameters

at	Attribute to copy
tgt	Target attribute set
src	Source attribute set

On Successful Return

tgt — contains value of the specified attribute from src

DupAttributes

Synopsis

```
TOK_ATTR_DATA * DupAttributes(
 const CK_ATTRIBUTE * attr,
 CK_COUNT attrCount);
```

Description

Make a copy of an array of attributes. The returned attribute set is newly allocated. This memory needs to be released via a call to *FreeAttributeSet*.

Parameters

attr	Attribute array to duplicate
attrCount	Number of attributes in attr

DupAttributeSet

Synopsis

```
TOK_ATTR_DATA * DupAttributeSet(
 const TOK_ATTR_DATA * attrData);
```

Description

Make a copy of an attribute set. The returned attribute set is newly allocated. This memory needs to be released via a call to *FreeAttributeSet*.

Parameters

attrData	Attribute set to duplicate

ExtractAllAttributes

Synopsis

```
CK_RV ExtractAllAttributes(
 CK_SESSION_HANDLE hSession,
 CK_OBJECT_HANDLE hobj,
 TOK_ATTR_DATA ** pna);
```

Description

Extract all non-sensitive valid attributes of an object.

Parameters

hSession	Open session handle
hObj	Object to extract from
pna	Reference to a reference to extracted attribute set

On Successful Return

***pna** — newly allocated attribute set of extracted attributes; this memory needs to be freed (see FreeAttributeSet)

FindAttr

Synopsis

```
CK_ATTRIBUTE * FindAttr(CK_ATTRIBUTE_TYPE attrType,const TOK_ATTR_DATA *
 attrData);
```

Description

Find the first attribute of the specified type in an attribute set.

Parameters

attrType	Type of attribute to locate
attrData	Attribute set

On Successful Return

Return a pointer to the attribute of the specified type.

FreeAttributes

Synopsis

```
void FreeAttributes(
  CK_ATTRIBUTE_PTR attr,
  CK_COUNT attrCount);
```

Description

Free all attributes contained in the attribute array, then free the array itself. This function also explicitly writes 0xaa to the memory to be freed before freeing.

Parameters

attr	Attribute array to free
attrCount	Number of attributes in the array

FreeAttributesNoClear

Synopsis

```
void FreeAttributesNoClear(
  CK_ATTRIBUTE_PTR attr,
  CK_COUNT attrCount);
```

Description

Free all attributes contained in the attribute array, then free the array itself. This function does not explicitly write 0xaa to the memory to be freed before freeing.

Parameters

attr	Attribute array to free
attrCount	Number of attributes in the array

FreeAttributeSet

Synopsis

```
void FreeAttributeSet(
  TOK_ATTR_DATA * attr);
```

Description

Free all of the attributes contained in the attribute set, and then free the set itself. This function also explicitly writes 0xaa to the memory to be freed before freeing.

Parameters

attr	Reference to the attribute set to free

FreeMechData

Synopsis

```
void FreeMechData(
 TOK_MECH_DATA * pMech);
```

Description

Free dynamic memory of pMech, including pMech itself.

Parameters

pMech	Mechanism list to free

genkMechanismTabFromMechanismTab

Synopsis

```
CK_MECHANISM_TYPE * genkMechanismTabFromMechanismTab(TOK_MECH_DATA *
 mTab,unsigned int * len);
```

Description

Creates a key generation mechanism table for the list of mechanisms supplied in mTab

Parameters

mTab	Number of mechanisms to look up
len	Number of returned mechanisms

genkpMechanismTabFromMechanismTab

Synopsis

```
CK_MECHANISM_TYPE * genkpMechanismTabFromMechanismTab(TOK_MECH_DATA *
 mTab,unsigned int * len);
```

Description

Creates a key pair generation mechanism table for the list of mechanisms supplied in mTab.

Parameters

mTab	List of mechanisms to look up
len	Number of returned mechanisms

GetCryptokiVersion

Synopsis

```
CK_VOID GetCryptokiVersion(CK_VERSION_PTR pVer);
```

Description

Returns the Cryptoki version information.

Parameters

pVer	Returned Cryptoki version

On Successful Return

pVer — pointer to a value which holds Cryptoki version

GetObjAttrInfo

Synopsis

```
CK_RV GetObjAttrInfo(CK_SESSION_HANDLE hSession,
 CK_OBJECT_HANDLE hObj,
 CK_ATTRIBUTE_PTR* ppAttributes,
 CK_ULONG_PTR pAttrCount);
```

Description

Get the list of attributes (type and size) of the specified object.

This function relies on the SafeNet extension CKA_ENUM_ATTRIBUTES to retrieve the list of attributes. Only the attribute type and size are returned. Attribute values must be retrieved by the caller as required.

Parameters

hSession	Handle to a valid session
hObj	Handle to the object to operate on
ppAttributes	Location to receive the attribute array (on return, *ppAttributes references an array of CK_ATTRIBUTE - the caller must free the memory allocated at *ppAttributes).
pAttrCount	Location to hold the number of CK_ATTRIBUTE entries (on return, *pAttrCount is the number of CK_ATTRIBUTE entries referenced by *ppAttributes).

On Successful Return

***ppAttributes** — handle that points to the returned attributes
pAttrCount — number of returned attributes

GetObjectClassAndKeyType

Synopsis

```
CK_RV GetObjectClassAndKeyType(
 const TOK_ATTR_DATA * attr,
 CK_OBJECT_CLASS * at_class,
 CK_KEY_TYPE * kt);
```

Description

Extract the object class and key type from an attribute set.

Parameters

attr	Attribute set to extract from
at_class	Reference to object class to hold resulting value
kt	Reference to key type to hold resulting value

On Successful Return

at_class — references located object class
kt — references located key type

hashMech

Synopsis

```
CK_MECHANISM_TYPE * hashMech(
 unsigned int * len);
```

Description

Return an array of all related mechanisms.

Parameters

len	Reference to int to hold the number of items returned

intAttr

Synopsis

```
unsigned int intAttr(
 const CK_ATTRIBUTE * at);
```

Description

Return the value of the attribute as an int.

Parameters

at	Reference to attribute whose value is to be returned

intAttrLookup

Synopsis

```
unsigned int intAttrLookup(CK_ATTRIBUTE_TYPE atype,const CK_ATTRIBUTE *
 attr,CK_COUNT attrCount);
```

Description

Extract an int attribute from an attribute template.

Parameters

atype	Type of attribute to extract attr array of attributes to search attrCount number of attributes in attr array

isBooleanAttr

Synopsis

```
int isBooleanAttr(const CK_ATTRIBUTE * na);
```

Description

Return `TRUE` if an attribute is a Boolean.

Parameters

na	Reference to attribute to check

isEnumeratedAttr

Synopsis

```
int isEnumeratedAttr(
 const CK_ATTRIBUTE * na);
```

Description

Return `TRUE` if attribute is enumerated and can have Vendor defined values.

Parameters

na	Reference to attribute to check

isGenMech

Synopsis

```
int isGenMech(
 CK_MECHANISM_TYPE mechType);
```

Description

Return `TRUE` if mechType is a key or key pair generation mechanism.

Parameters

mechType	Mechanism type to check

kgMech

Synopsis

```
CK_MECHANISM_TYPE * kgMech(
 unsigned int * len);
```

Description

Return an array of all key generation related mechanisms.

Parameters

mechType	Reference to int to hold the number of items returned

isNumericAttr

Synopsis

```
int isNumericAttr(const CK_ATTRIBUTE * na);
```

Description

Return TRUE if an attribute is a numeric.

Parameters

na	Reference to attribute to check

isSensitiveAttr

Synopsis

```
int isSensitiveAttr(
 const struct TOK_ATTR_DATA * attrData,
 const CK_ATTRIBUTE * na);
```

Description

Report `TRUE` for potentially sensitive attributes, as per the PKCS#11 spec. Note that the object has to be marked sensitive for this to have any effect.

ProtectToolkit C extension: all objects have the `CKA_VALUE` as sensitive if the object has `CKA_SENSITIVE`set to `TRUE`. This is useful for objects that are used internally only, or just wrapped for transmission elsewhere.

Parameters

na	Reference to attribute to check

KeyFromPin

Synopsis

```
void KeyFromPin(
 unsigned char key[16],
 unsigned int klen,
 CK_USER_TYPE user,
 const unsigned char * pin,
 unsigned int pinLen);
```

Description

Generate a double length key from a PIN, using PKCS#5 password based encryption.

Parameters

key	Buffer to hold generated key
keylen	Number of bytes in key (should be 16)
user	Salt value for key generation
pin	Password used for key generation
pinLen	Number of bytes referenced by pin

On Successful Return

key — contains the generated key

kpgMech

Synopsis

```
CK_MECHANISM_TYPE * kpgMech(
 unsigned int * len);
```

Description

Return an array of all key pair generation related mechanisms.

Parameters

len	Reference to int to hold the number of items returned

ktFromMech

Synopsis

```
CK_KEY_TYPE * ktFromMech(
 CK_MECHANISM_TYPE mt,
 unsigned int * len);
```

Description

Return an array of key types valid for the given mechanism. The returned array does not need to be freed.

Parameters

mt	Mechanism type to get key types for
len	Reference to int to hold the number of items in returned array

On Successful Return

***len** number of items in returned array

LookupMech

Synopsis

```
int LookupMech(
 TOK_MECH_DATA * pMech,
 CK_MECHANISM_TYPE mechType);
```

Description

Return `TRUE` if mechType is in the pMech list.

Parameters

pMech	Reference to mechanism list
mechType	Mechanism to look for in pMech list

MatchAttributeSet

Synopsis

```
int MatchAttributeSet(
 const TOK_ATTR_DATA * match,
 const TOK_ATTR_DATA * ad);
```

Description

Do a comparison of two attribute sets. Every attribute in the 'match' set must be found in the 'ad' set. It is OK if 'ad' is a super set of 'match'. Return `TRUE` if all attributes in 'match' were found in 'ad'.

Parameters

match	Attribute set to look for
ad	Atritute set to compare to

mechDeriveFromKt

Synopsis

```
CK_MECHANISM_TYPE * mechDeriveFromKt(CK_KEY_TYPE kt,unsigned int * len);
```

Description

Return an array of derive mechanisms valid for the given key type. The returned array is newly allocated and needs to be freed.

Parameters

kt	Key type to look up
len	Pointer to integer that receives length of returned array

On Successful Return

Array of CK_MECHANISM_TYPE values or NULL if key type is invalid. Caller should free the array when finished.

mechFromKt

Synopsis

```
CK_MECHANISM_TYPE * mechFromKt(
 CK_KEY_TYPE kt,
 unsigned int * len);
```

Description

Return an array of mechanisms valid for the given key type. The returned array is newly allocated and needs to be freed.

Parameters

kt	Key type to get mechanisms for
len	Reference to int to hold number of items in returned array

On Successful Return

Array of CK_MECHANISM_TYPE values or NULL if key type is invalid. Caller should free the array when finished.

mechSignFromKt

Synopsis

```
CK_MECHANISM_TYPE * mechSignFromKt(CK_KEY_TYPE kt,unsigned int * len);
```

Description

Return an array of signing mechanisms valid for the given key type. The returned array is newly allocated and needs to be freed.

Parameters

kt	Key type to get mechanisms for
len	Reference to int to hold number of items in returned array

On Successful Return

Array of CK_MECHANISM_TYPE values or NULL if key type is invalid. Caller should free the array when finished.

mechSignRecFromKt

Synopsis

```
CK_MECHANISM_TYPE * mechSignRecFromKt(CK_KEY_TYPE kt,unsigned int * len);
```

Description

Return an array of signing mechanisms valid for the given key type. The returned array is newly allocated and needs to be freed.

Parameters

kt	Key type to get mechanisms for
len	Reference to int to hold number of items in returned array

On Successful Return

Array of CK_MECHANISM_TYPE values or NULL if key type is invalid. Caller should free the array when finished.

NewAttributeSet

Synopsis

```
TOK_ATTR_DATA * NewAttributeSet(
 unsigned int count);
```

Description

Allocate memory for an attribute set to hold the specified number of attributes. The returned memory needs to be freed (see FreeAttributeSet)

Parameters

count	Number of attribute place holders to allocate in the set

numAttr

Synopsis

```
CK_NUMERIC numAttr(
const CK_ATTRIBUTE * at);
```

Description

Return the value of the attribute as a numeric.

Parameters

at	Reference to attribute whose value is to be returned

numAttrLookup

Synopsis

```
CK_NUMERIC numAttrLookup(CK_ATTRIBUTE_TYPE atype,const CK_ATTRIBUTE *
 attr,CK_COUNT attrCount);
```

Description

Extract a numeric attribute from an attribute template.

Parameters

atype	Type of attribute to extract attr array of attributes to search
attrCount	Number of attributes in attr array

PvcFromPin

Synopsis

```
void PvcFromPin(unsigned char * key,unsigned int klen,CK_USER_TYPE
 user,const unsigned char * pin,unsigned int pinLen);
```

Description

Create a PVC from a PIN using PKCS#5 password based encryption.

Parameters

key	Resulting pvc
klen	Number of bytes referenced by key
user	Salt value
pin	Password
pinLen	Number of bytes referenced by pin

On Successful Return

key — contains the pvc

ReadAttr

Synopsis

```
int ReadAttr(
 void * buf,
 unsigned int len,
 unsigned int * plen,
 CK_ATTRIBUTE_TYPE attrType,
 const TOK_ATTR_DATA * attr);
```

Description

Read an attribute value from an attribute set. Return `TRUE`if the attribute was present.

Parameters

buf	Buffer to receive attribute value
len	Number of bytes referenced by buf
plen	Reference to int to hold number of bytes copied to buf
attrType	Type of attribute to extract from attr
attr	Attribute set to search

On Successful Return

buf — contains attribute value
plen — references number of bytes copied into buf

TransferAttr

Synopsis

```
CK_RV TransferAttr(
  CK_ATTRIBUTE * pTgtTemplate,
  const CK_ATTRIBUTE * pSrcTemplate,
  CK_COUNT attrCount);
```

Description

Using at_assign, copy attribute values from one array to another. The order of the attributes must be the same in both arrays.

Parameters

pTgtTemplate	Target attribute array
pSrcTemplate	Source attribute array
attrCount	Number of attributes to copy from source to target

On Successful Return

pTgtTemplate — contains copy of attribute values from pSrcTemplate

UnwrapDec

Synopsis

```
int UnwrapDec(
  CK_SESSION_HANDLE hSession,
  CK_OBJECT_HANDLE hUnwrapper,
  CK_OBJECT_HANDLE * hUnwrappedKey,
  unsigned char * buf,
  unsigned int bufLen);
```

Description

Unwrap a key and decode its attributes.

Parameters

hSession	Open session handle
hUnwrapper	Handle to unwrapping key
hUnwrappedKey	Reference to handle to the key unwrapped
buf	Reference to bytes containing the key and attributes
bufLen	Number of bytes referenced by buf

On Successful Return

***hUnwrappedKey** — handle to unwrapped key with attributes

WrapEnc

Synopsis

```
int WrapEnc (
 CK_SESSION_HANDLE hSession,
 CK_OBJECT_HANDLE hWrapper,
 CK_OBJECT_HANDLE hWrappee,
 unsigned char * buf,
 unsigned int bufLen,
 CK_SIZE * bytesWritten);
```

Description

Wrap a key, encode its attributes and write it to a buffer.

Parameters

hSession	Open session handle
hWrapper	Handle to wrapping key
hWrappee	Wrappee handle to the key to wrap
buf	Reference to bytes to hold the result
bufLen	Number of bytes referenced by buf
bytesWritten	Reference to value to hold the number of bytes written to buf

On Successful Return

buf — contains the wrapped key and encoded attributes *bytesWritten number of bytes written to buf

WriteAttr

Synopsis

```
CK_RV WriteAttr(
 const void * buf,
 unsigned int len,
 CK_ATTRIBUTE_TYPE attrType,
 TOK_ATTR_DATA * attr);
```

Description

Add/Replace an attribute to an attribute set. Delete attribute if len is 0.

Parameters

buf	Value to add to attribute set
len	Number of bytes to add from buf
attrType	Type of attribute to add
attr	Attribute set to modify

On Successful Return

attr — modified attribute set

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 13
HEX2BIN.H LIBRARY REFERENCE

Overview

The ProtectToolkit C Software Development Kit offers a number of extended API libraries with functionality that is extended to that of the standard PKCS#11 function set.

The following additional features do not form part of the standard PKCS#11 functionality, but are provided by SafeNet as part of the ProtectToolkit C API within the HEX2BIN.H library.

hex2bin

Synopsis

```
int hex2bin(
 void * bin,
 const char * hex,
 unsigned maxLen );
```

Description

Used to convert ASCII HEX strings to binary data.

The function ignores white space in 'hex' and converts pairs of HEX characters into their equivalent binary representation.

Example:

Input –
hex = "41424300"
maxLen = 4

Output -
bin[4] = "ABC"

Parameters

bin Output	A buffer to receive the binary data
hex Input	A string of ASCII HEX characters to be converted
maxLen Input	The maximum number of characters that 'bin' can hold

bin2hex

Synopsis

```
int bin2hex(
 char * hex,
 const void * bin,
 unsigned maxLen );
```

Description

Converts binary data into an ASCII HEX. This function is the inverse of bin2hex.

Example:

Input -
bin = "abc"
maxLen = 3

Output -
hex[7] = "616263"

Parameters

bin Input	A buffer of binary data
hex Output	A buffer to receive the string of ASCII HEX characters
maxLen Input	The number of characters that 'bin' contains that should be converted (this is not the length of the output buffer 'hex')

bin2hexM

Synopsis

```
int bin2hexM(
 char * hex,
 const void * bin,
 unsigned maxLen,
 unsigned int lineLen);
```

Description

As for bin2hex Converts binary data into an ASCII HEX and then inserts a '\n' after every 'lineLen' characters for display formatting.

Parameters

bin Input	A buffer of binary data
hex Output	A buffer to receive the string of ASCII HEX characters
maxLen Input	The number of characters that 'bin' contains that should be converted (this is not the length of the output buffer 'hex')
lineLen	Number of characters before a new line (\n) is added

memdump

Synopsis

```
void memdump(
 const char * txt,
 const unsigned char * buf,
 unsigned int len);
```

Description

This function prints the contents of the memory as binary data to stdout for debugging purposes.

Parameters

txt Input	Title string (may be NULL)
buf Input	Binary data that is to be hex dumped
len Input	Length of 'buf'

SetOddParity

Synopsis

```
void SetOddParity(
 unsigned char * string,
 unsigned int count);
```

Description

Converts a buffer of binary data to 'odd' parity.

For each byte in 'string' this function will flip the least significant bit, if necessary, to make the number of one bits in the entire byte an odd number (odd parity).

Parameters

string	Input/output, binary data to convert
count	Length of 'string'

isOddParity

Synopsis

```
int isOddParity(
 const unsigned char * string,
 unsigned int count);
```

Description

This function checks the parity of the supplied data and returns 1 if buffer contains bytes that are all of odd parity.

Parameters

string	Input, binary data to check
count	Input, length of 'string'

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 14
HSMADMIN.H LIBRARY REFERENCE

Overview

The ProtectToolkit C Software Development Kit offers a number of extended API libraries with functionality that is extended to that of the standard PKCS#11 function set.

The following additional features do not form part of the standard PKCS#11 functionality, but are provided by SafeNet as part of the ProtectToolkit C API within the HSMAdmin.h library.

The following functions provide an interface to the HSM's Real Time Clock (RTC). This Library is used in conjunction with the CTCONF utility. The CTCONF utility provides the capability to set the access control configuration parameters for the RTC.

This Library cannot be used in software emulation mode.

Return Codes

The return code of all of the functions in the HSMAdmin Library is the enumerated type HSMADM_RV which can have the following values.

Return Code	Meaning
HSMADM_OK	The operation was successful.
HSMADM_BAD_PARAMETER	One or more of the parameters have an invalid value.
HSMADM_ADJ_TIME_LIMIT	The delta value passed to the HSMADM_AdjustTime() is too large, and will not be used.
HSMADM_ADJ_COUNT_LIMIT	The number of calls made to the HSMADM_AdjustTime() that change the time is too large. The adjustment will not be made.
HSMADM_NO_MEMORY	There is not enough memory to complete operation.
HSMADM_SYSERR	There was a system error. The operation was not performed.

HSMADM_GetTimeOfDay

Synopsis

```
#include hsmadmin.h

HSMADM_RV    HSMADM_GetTimeOfDay(unsigned int
 hsmIndex,HSMADM_TimeVal_t * tv );
```

Description

Obtains the current time of day from the HSM RTC.

Parameters

hsmIndex	Zero-based index of the HSM number to be used
tv	Address of the variable which is to be initialized with the current time of day. It indicates the time passed since midnight, 1 Jan 1970. This struct contains a field tv_usec, which is the number of microseconds. The HSM real-time clock only has millisecond resolution; therefore, tv_usec is always rounded up to the nearest millisecond HSMADM_TimeVal_t is defined in hsmadmin.h.

HSMADM_AdjustTime

Synopsis

```
#include hsmadmin.h

HSMADM_RV HSMADM_AdjustTime(unsigned int hsmIndex,const
 HSMADM_TimeVal_t * delta,HSMADM_TimeVal_t * oldDelta );
```

Description

Either adjust the time, or obtain the current adjustment value.

The parameter, delta, indicates the adjustment to be applied to the HSM RTC. The HSM is only capable of performing Slew Adjustment when updating the Real Time Clock (RTC). This prevents large (multiple second) negative or positive steps of the current RTC. The RTC has a Slew Adjustment of 1 second for every 10 seconds of elapsed time, hence if the RTC was out by 1000 seconds, it will take approx 10000 seconds (2.7 hours) to match the external time source.

Because Slew Adjustment is the means by which the RTC is updated, the HSM may not have completed making an adjustment requested by a previous HSMADM_AdjustTime call. If there is an adjustment being performed when this function is called, then this adjustment is discarded, and the new adjustment value is used instead.

This function can alternatively be used to obtain the value of the time adjustment that remains to be completed. If the parameter delta is NULL, and oldDelta is a valid pointer, it will return the pending adjustment.

Parameters

hsmIndex	Zero-based index of the HSM number to be used
delta	Amount of adjustment to be made to the RTC. This parameter must be NULL if *oldDelta* is not NULL. HSMADM_TimeVal_t is defined in hsmadmin.h
oldDelta	Address of the variable that will receive the value of the adjustment that remains to be completed. HSMADM_TimeVal_t is defined in hsmadmin.h. If this parameter is not NULL, *delta* must be NULL

HSMADM_SetRtcStatus

Synopsis

```
#include hsmadmin.h

HSMADM_RV HSMADM_SetRtcStatus(unsigned int
 hsmIndex,HSMADM_RtcStatus_t status );
```

Description

Changes the RTC status.

Parameters

hsmIndex	Zero-based index of the HSM number to be used
status	New status of the RTC. Possible values of the RTC status are defined in hsmadmin.h and are described below.

Value	Meaning
HSMADM_RTC_UNIN ITIALIZED	The RTC is not initialized yet.
HSMADM_RTC_STA ND_ALONE	The RTC is in the stand alone mode. This means that it is completely controlled by the crypto subsystem. In this mode, all cryptographic operations are allowed to use the value of the clock.
HSMADM_RTC_MANAGED_UNTRUSTED	The RTC is being controlled by an external program; but the value is not trusted yet. This means certain cryptographic operations are refused access to the RTC because the value is (possibly) incorrect. When the RTC Status is set to this value, the CTCONF –t command, which normally is used to set the RTC, cannot be used.
HSMADM_RTC_MANAGED_TRUSTED	The RTC is being controlled by an external program, and its value may be trusted. This means that all cryptographic operations are allowed to use the value of the clock. When the RTC Status is set to this value, the CTCONF –t command, which normally is used to set the RTC, cannot be used.

HSMADM_GetRtcStatus

Synopsis

```
#include hsmadmin.h

HSMADM_RV HSMADM_GetRtcStatus(unsigned int
 hsmIndex,HSMADM_RtcStatus_t* status );
```

Description

Obtain the HSM RTC status.

Parameters

hsmIndex	Zero-based index of the HSM number to be used. This parameter is only valid if RTC Access Control is enabled. RTC Access Control can be modified via the CTCONF utility.
status	Address of the variable that will obtain the current status of the RTC. This parameter must not be NULL. Possible values of the RTC status are defined in hsmadmin.hand are described below.

Value	Meaning
HSMADM_RTC_UNIN ITIALIZED	The RTC is not initialized yet.
HSMADM_RTC_STA ND_ALONE	The RTC is in the stand alone mode. This means that it is completely controlled by the crypto subsystem. In this mode, all cryptographic operations are allowed to use the value of the clock.
HSMADM_RTC_MANAGED_UNTRUSTED	The RTC is being controlled by an external program; but the value is not trusted yet. This means certain cryptographic operations are refused access to the RTC because the value is (possibly) incorrect. When the RTC Status is set to this value, the CTCONF –t command, which normally is used to set the RTC, cannot be used.
HSMADM_RTC_MANAGED_TRUSTED	The RTC is being controlled by an external program, and its value may be trusted. This means that all cryptographic operations are allowed to use the value of the clock. When the RTC Status is set to this value, the CTCONF –t command, which normally is used to set the RTC, cannot be used.

HSMADM_GetRtcAdjustAmount

Synopsis

```
#include hsmadmin.h
HSMADM_RV HSMADM_GetRtcAdjustAmount(unsigned intlong*);
hsmIndex,totalMs
```

Description

Get the effective total amount, in milliseconds, of adjustment made to the RTC using the HSMADM_AdjustTime() function.

Parameters

hsmIndex	Zero-based index of the HSM number to be used.
totalMs	Address of the variable that will contain the total amount adjusted. The total amount adjusted is calculated by summing the adjust amounts specified via a valid HSMADM_AdjustTime() call. For instance if two adjustments are made of 20ms and -3ms this parameter should return 17ms. This parameter must not be NULL. This parameter is only valid if RTC Access Control is enabled. RTC

	Access Control can be modified via the CTCONF utility.

HSMADM_GetRtcAdjustCount

Synopsis

```
#include hsmadmin.h

HSMADM_RV HSMADM_GetRtcAdjustCount(unsigned intunsigned long*);

hsmIndex,totalCount
```

Description

Get the effective count of adjustments made to the RTC using the HSMADM_AdjustTime() function.

Parameters

hsmIndex	Zero-based index of the HSM number to be used.
totalCount	Address of the variable that will obtain the total count of adjustments. The total count of adjustments indicates the a count of the number of valid adjustments made via HSMADM_AdjustTime() call. This parameter must not be NULL. This parameter is only valid if RTC Access Control is enabled. RTC Access Control can be modified via the CTCONF utility.

HSMADM_GetHsmUsageLevel

Synopsis

```
#include hsmadmin.h

HSMADM_RV HSMADM_GetHsmUsageLevel (unsigned int hsmIndex,
unsigned long* value
);
```

Description

Get the usage level of the hsm as a percentage i.e. the load on the HSM.

Parameters

hsmIndex	Zero-based index of the HSM number to be used.
totalCount	Address of the variable that will obtain the value. This parameter must not be NULL

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 15
KMLIB.H LIBRARY REFERENCE

Overview

The ProtectToolkit C Software Development Kit offers a number of extended API libraries with functionality that is extended to that of the standard PKCS#11 function set.

The following functions provide an interface to the key management library used by the KMUTIL utility. Not all functions are documented – refer to kmlib.h for more details.

KM_EncodeECParamsP

```
#include    "kmlib.h"
```

	Windows	**UNIX**
Library	Kmlib.lib	Libkmlib.a

```
CK_RV KM_EncodeECParamsP(
                        CK_BYTE_PTR prime,    CK_SIZE primeLen,
                        CK_BYTE_PTR curveA,   CK_SIZE
curveALen,
                        CK_BYTE_PTR curveB,   CK_SIZE
curveBLen,
                        CK_BYTE_PTR curveSeed,CK_SIZE
curveSeedLen,
                        CK_BYTE_PTR baseX,    CK_SIZE baseXLen,
                        CK_BYTE_PTR baseY,    CK_SIZE baseYLen,
                        CK_BYTE_PTR bpOrder,  CK_SIZE
bpOrderLen,
                        CK_BYTE_PTR cofactor, CK_SIZE
cofactorLen,
                        CK_BYTE_PTR result,   CK_SIZE *
resultLen
                        );
```

Do DER enc of ECC Domain Parameters Prime

All integer values are variable length big endian numbers with optional leading zeros. Integer lengths are all in bytes.

Parameters

prime	Prime modulus
primeLen	Prime modulus len
curveA	Elliptic Curve coefficient a
curveALen	Elliptic Curve coefficient a length
curveB	Elliptic Curve coefficient b

curveBLen	Elliptic Curve coefficient b length
curveSeed	Seed (optional may be NULL)
curveSeedLen	Seed length
baseX	Elliptic Curve point X coord
baseXLen	Elliptic Curve point X coord length
baseY	Elliptic Curve point Y coord
baseYLen	Elliptic Curve point Y coord length
bpOrder	Order n of the Base Point
bpOrderLen	Order n of the Base Point length
cofactor	The integer h = #E(Fq)/n (optional)
cofactorLen	h length
result	Resultant Encoding (length prediction supported if NULL)
resultLen	Buffer len/Length of resultant encoding
Return	Status of operation. CKR_OK if ok

KM_EncodeECParams2M

```
#include    "kmlib.h"
```

	Windows	UNIX
Library	Kmlib.lib	Libkmlib.a

```
typedef enum {
    ECBT_GnBasis,   /* Gaussian Normal Basis - parameters = 0,  0,  0
*/
    ECBT_TpBasis,   /* Trinomial Basis - parameters = k,  0,  0 */
    ECBT_PpBasis    /* Pentanomial Basis - parameters = k1, k2, k3 */
} ECBasisType;

CK_RV KM_EncodeECParams2M(
        CK_SIZE m,
        ECBasisType basis,
        CK_SIZE  parameters[3],
        CK_BYTE_PTR curveA,   CK_SIZE curveALen,
        CK_BYTE_PTR curveB,   CK_SIZE curveBLen,
        CK_BYTE_PTR curveSeed,CK_SIZE curveSeedLen,
        CK_BYTE_PTR baseX,    CK_SIZE baseXLen,
        CK_BYTE_PTR baseY,    CK_SIZE baseYLen,
        CK_BYTE_PTR bpOrder,  CK_SIZE bpOrderLen,
        CK_BYTE_PTR cofactor, CK_SIZE cofactorLen,
        CK_BYTE_PTR result,   CK_SIZE * resultLen
);
```

Do DER enc of ECC Domain Parameters 2^M

All long integer values are variable length big endian numbers with optional leading zeros, lengths are all in bytes.

Parameters

M	Degree of field
basis	Should be ECBT_GnBasis or ECBT_TpBasis or ECBT_PpBasis
parameters	Array of three integers - values depend on basis ECBT_GnBasis - parameters = 0. 0. 0 ECBT_TpBasis - parameters = k. 0. 0 ECBT_PpBasis - parameters = k1.k2.k3
curveA	Elliptic Curve coefficient a
curveALen	Elliptic Curve coefficient a length
curveB	Elliptic Curve coefficient b
curveBLen	Elliptic Curve coefficient b length
curveSeed	Seed (optional may be NULL)
curveSeedLen	Seed length
baseX	Elliptic Curve point X coord
baseXLen	Elliptic Curve point X coord length
baseY	Elliptic Curve point Y coord
baseYLen	Elliptic Curve point Y coord length
bpOrder	Order n of the Base Point
bpOrderLen	Order n of the Base Point length
cofactor	The integer h = #E(Fq)/n (optional)
cofactorLen	h length
result	Resultant Encoding (length prediction supported if NULL)
resultLen	Buffer len/Length of resultant encoding
Return	Status of operation. CKR_OK if ok

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 16
CTAUTH.H LIBRARY REFERENCE

Overview

The ctauthlib library provides a single function used by a remote agent attempting to authenticate to the HSM using the challenge Response system.

CT_Gen_AUTH_Response

Creates the response to a challenge.

```
CK_RV CT_Gen_AUTH_Response(CK_BYTE_PTR pPin,
                CK_ULONG ulPinLen, CK_BYTE_PTR pChallenge,
                CK_ULONG ulChallengeLen, CK_USER_TYPE userType,
                CK_BYTE_PTR pResponse, CK_ULONG_PTR
pulResponse);
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX
ATTRIBUTE CERTIFICATE

The Set Attribute Ticket, which is used to authorise updates to key usage limits, has the format of an Attribute Certificate defined by PKIX (RFC 3281).

```
AttributeCertificate ::= SEQUENCE {
   acinfo               AttributeCertificateInfo,
   signatureAlgorithm   AlgorithmIdentifier,
   signatureValue       BIT STRING
}

AttributeCertificateInfo ::= SEQUENCE {
      version              AttCertVersion -- version is v2,
      holder               Holder,
      issuer               AttCertIssuer,
      signature            AlgorithmIdentifier,
      serialNumber         CertificateSerialNumber,
      attrCertValidityPeriod   AttCertValidityPeriod,
      attributes           SEQUENCE OF Attribute,
      issuerUniqueID       UniqueIdentifier OPTIONAL,
      extensions           Extensions OPTIONAL
}

AttCertVersion ::= INTEGER { v2(1) }

Holder ::= SEQUENCE {
       baseCertificateID   [0] IssuerSerial OPTIONAL,
    -- the issuer and serial number of
    -- the holder's Public Key Certificate

       entityName          [1] GeneralNames OPTIONAL,
       objectDigestInfo    [2] ObjectDigestInfo OPTIONAL
           -- used to directly authenticate the target key,
           -- see further description below
}

ObjectDigestInfo ::= SEQUENCE {
       digestedObjectType  ENUMERATED {
       publicKey            (0),
       publicKeyCert        (1),
       otherObjectTypes     (2) },
       -- otherObjectTypes only to be used
       otherObjectTypeID   OBJECT IDENTIFIER OPTIONAL,
              -- must be OID_X509_ATTR_KEY_DIGEST
       digestAlgorithm     AlgorithmIdentifier,
       objectDigest        BIT STRING
            }
```

The algorithm `OID_X509_ATTR_KEY_DIGEST` is:

```
objectDigest  = Digest(Token_Serial_Number | Token_Label |
ObjectID)
```

Where ObjectID is the concatenation of the CKA_LABEL and CKA_ID attributes of the target Object.

```
AttCertIssuer ::= CHOICE {
      v1Form   GeneralNames,  -- MUST NOT be used in this
                              -- profile
     v2Form   [0] V2Form     -- v2 only
}

V2Form ::= SEQUENCE {
     issuerName            GeneralNames  OPTIONAL,
     baseCertificateID     [0] IssuerSerial  OPTIONAL,
     objectDigestInfo      [1] ObjectDigestInfo  OPTIONAL
     -- issuerName MUST be present in this profile
     -- baseCertificateID and objectDigestInfo MUST NOT
     -- be present in this profile
}

IssuerSerial  ::=  SEQUENCE {
     issuer         GeneralNames,
     serial         CertificateSerialNumber,
     issuerUID      UniqueIdentifier OPTIONAL
}

AttCertValidityPeriod  ::= SEQUENCE {
     notBeforeTime  GeneralizedTime,
     notAfterTime   GeneralizedTime
}


Attribute ::= SEQUENCE {
     type      AttributeType,
     values    SET OF AttributeValue
     -- at least one value is required
}

AttributeType ::= OBJECT IDENTIFIER
-- there is a different OID for each type of Cryptoki Attribute
-- see below for a list

AttributeValue ::= ANY DEFINED BY AttributeType
      -- the data type depends on the type field but it
      -- represents the value part of the Cryptoki attribute.
```

OID Used to Indicate Key Digest Algorithm

OID	OID-type
{ iso(1) identified-organization(3) dod(6) internet(1) private(4) enterprises(1) safeNetInc(23629) safenetRoot(1) safenetHSM(4) ptkc(2) objDigests(2) key(1) }	`OID_X509_ATTR_KEY_DIGEST`

OID Value	OID-type	Cryptoki Attribute Type	DER Encoded Value
{ iso(1) identified-organization(3) dod(6) internet(1) private(4) enterprises(1) safeNetInc(23629) safenetRoot(1) safenetHSM(4) ptkc(2) p11Attrs(1) usage_limit(1) }	`OID_X509_ATTR_USAGE_LIMIT`	CKA_USAGE_LIMIT	INTEGER
{ iso(1) identified-organization(3) dod(6) internet(1) private(4) enterprises(1) safeNetInc(23629) safenetRoot(1) safenetHSM(4) ptkc(2) p11Attrs(1) end_date(2) }	`OID_X509_ATTR_END_DATE`	CKA_END_DATE	PrintableString
{ iso(1) identified-organization(3) dod(6) internet(1) private(4) enterprises(1) safeNetInc(23629) safenetRoot(1) safenetHSM(4) ptkc(2) p11Attrs(1) start_date(3) }	`OID_X509_ATTR_START_DATE`	CKA_START_DATE	PrintableString
{ iso(1) identified-organization(3) dod(6) internet(1) private(4) enterprises(1) safeNetInc(23629) safenetRoot(1) safenetHSM(4) ptkc(2) p11Attrs(1) admin_cert(4) }	`OID_X509_ATTR_ADMIN_CERT`	CKA_ADMIN_CERT	

THIS PAGE INTENTIONALLY LEFT BLANK

GLOSSARY
COMMON TERMS AND PHRASEOLOGY

Software Development Kits (SDKs)

Other documentation may refer to the SafeNet Cprov and Protect Toolkit J SDKs. These SDKs have been renamed ProtectToolkit C and ProtectToolkit J respectively.

- The names Cprov and ProtectToolkit C refer to the same device in the context of this or previous manuals.
- The names Protect Toolkit J and ProtectToolkit J refer to the same device in the context of this or previous manuals.

END OF DOCUMENT