This is a Non-Standards Track Work Product. The patent provisions of the OASIS IPR Policy do not apply.

OASIS

PKCS #11 Cryptographic Token Interface Usage Guide Version 2.40

Committee Note Draft 02 / Public Review Draft 02

23 April 2014

Specification URIs

This version:

http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/cnprd02/pkcs11-ug-v2.40-cnprd02.doc (Authoritative)
http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/cnprd02/pkcs11-ug-v2.40-cnprd02.html
http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/cnprd02/pkcs11-ug-v2.40-cnprd02.pdf

Previous version:

http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/cnprd01/pkcs11-ug-v2.40-cnprd01.doc (Authoritative)
http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/cnprd01/pkcs11-ug-v2.40-cnprd01.html
http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/cnprd01/pkcs11-ug-v2.40-cnprd01.pdf

Latest version:

http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/pkcs11-ug-v2.40.doc (Authoritative)
http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/pkcs11-ug-v2.40.html
http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/pkcs11-ug-v2.40.pdf

Technical Committee:

OASIS PKCS 11 TC

Chairs:

Robert Griffin (robert.griffin@rsa.com), EMC Corporation
Valerie Fenwick (valerie.fenwick@oracle.com), Oracle

Editors:

John Leiseboer (jl@quintessencelabs.com), QuintessenceLabs Pty Ltd.
Robert Griffin (robert.griffin@rsa.com), EMC Corporation

Related work:

This document is related to:

- *PKCS #11 Cryptographic Token Interface Base Specification Version 2.40*. Edited by Susan Gleeson and Chris Zimman. Latest version. http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/pkcs11-base-v2.40.html.
- *PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 2.40*. Edited by Susan Gleeson and Chris Zimman. Latest version. http://docs.oasis-open.org/pkcs11/pkcs11-curr/v2.40/pkcs11-curr-v2.40.html.
- *PKCS #11 Cryptographic Token Interface Historical Mechanisms Specification Version 2.40*. Edited by Susan Gleeson and Chris Zimman. Latest version. http://docs.oasis-open.org/pkcs11/pkcs11-hist/v2.40/pkcs11-hist-v2.40.html.
- *PKCS #11 Cryptographic Token Interface Profiles Version 2.40*. Edited by Tim Hudson. Latest version. http://docs.oasis-open.org/pkcs11/pkcs11-profiles/v2.40/pkcs11-profiles-v2.40.html.

Abstract:

This document provides guidance on using PKCS #11 v2.40.

Status:

This document was last revised or approved by the OASIS PKCS 11 TC on the above date. The level of approval is also listed above. Check the “Latest version” location noted above for possible later revisions of this document.

Technical Committee members should send comments on this document to the Technical Committee’s email list. Others should send comments to the Technical Committee by using the “Send A Comment” button on the Technical Committee’s web page at https://www.oasis-open.org/committees/pkcs11/.

Citation format:

When referencing this document the following citation format should be used:

[PKCS11-UG-v2.40]

PKCS #11 Cryptographic Token Interface Usage Guide Version 2.40. Edited by John Leiseboer and Robert Griffin. 23 April 2014. OASIS Committee Note Draft 02 / Public Review Draft 02. http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/cnprd02/pkcs11-ug-v2.40-cnprd02.html. Latest version: http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/pkcs11-ug-v2.40.html.

Copyright © OASIS Open 2014. All Rights Reserved.

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full Policy may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Table of Contents

 Copyright © OASIS Open 2014. All Rights Reserved.

 Copyright © OASIS Open 2014. All Rights Reserved.

1 Introduction

1.1 Description of this Document

This PKCS #11 Cryptographic Token Interface Usage Guide Version 2.40 is intended to complement **[PKCS11-Base]**, **[PKCS11-Curr]**, **[PKCS11-Hist** and **[PKCS11-Prof]** by providing guidance on how to implement the PKCS #11 interface most effectively. In particular, it includes the following guidance:

- General overview information and clarification of assumptions and requirements that drive or influence the design of PKCS #11 and the implementation of PKCS #11-compliant solutions.
- Specific recommendations for implementation of particular PKCS #11 functionality.
- Functionality considered for inclusion in PKCS #11 V2.40, but deferred to subsequent versions of the standard.

Guidance regarding conformant PKCS #11 implementations is provided in [**PKCS11-Prof**].

1.2 Terminology

For a list of terminologies refer to **[PKCS11-Spec]**.

1.3 References (normative)

This is a non-standars track document and does not contain normative references.

1.4 References (non-normative)

[FORTEZZA! FORTEZZA CIPG Application Programming Interface http://cryptome.org/jya/fortezza.htm

[GCS-API] Generic Cryptographic Service API Base (GCS-API). X/Open Preliminary Specification. June 1996. http://archive.opengroup.org/publications/archive/CDROM/p442.pdf

[PKCS11-Base] *PKCS #11 Cryptographic Token Interface Base Specification Version 2.40*. Latest version. http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/pkcs11-base-v2.40.html.

[PKCS11-Curr] *PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 2.40*. Latest version. http://docs.oasis-open.org/pkcs11/pkcs11-curr/v2.40/pkcs11-curr-v2.40.html.

[PKCS11-Hist] *PKCS #11 Cryptographic Token Interface Historical Mechanisms Specification Version 2.40*. Latest version. http://docs.oasis-open.org/pkcs11/pkcs11-hist/v2.40/pkcs11-hist-v2.40.html.

 Copyright © OASIS Open 2014. All Rights Reserved.

[PKCS11-Prof] *PKCS #11 Cryptographic Token Interface Profiles Version 2.40*. Latest version. http://docs.oasis-open.org/pkcs11/pkcs11-profiles/v2.40/pkcs11-profiles-v2.40.html.

[PKCS11_V2.11] **PKCS #11 Cryptographic Token Interface Standard Version 2.11, Revision 1. November 2001. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v211/pkcs-11v2-11r1.doc**

Copyright © OASIS Open 2014. All Rights Reserved.

2 General overview

2.1 Introduction

Portable computing devices such as smart cards, PCMCIA cards, and smart diskettes are ideal tools for implementing public-key cryptography, as they provide a way to store the private-key component of a public-key/private-key pair securely, under the control of a single user. With such a device, a cryptographic application, rather than performing cryptographic operations itself, utilizes the device to perform the operations, with sensitive information such as private keys never being revealed. As more applications are developed for public-key cryptography, a standard programming interface for these devices becomes increasingly valuable. This standard addresses this and other needs.

2.2 General model

Cryptoki's general model is illustrated in the following figure. The model begins with one or more applications that need to perform certain cryptographic operations, and ends with one or more cryptographic devices, on which some or all of the operations are actually performed. A user may or may not be associated with an application.

 Copyright © OASIS Open 2014. All Rights Reserved.

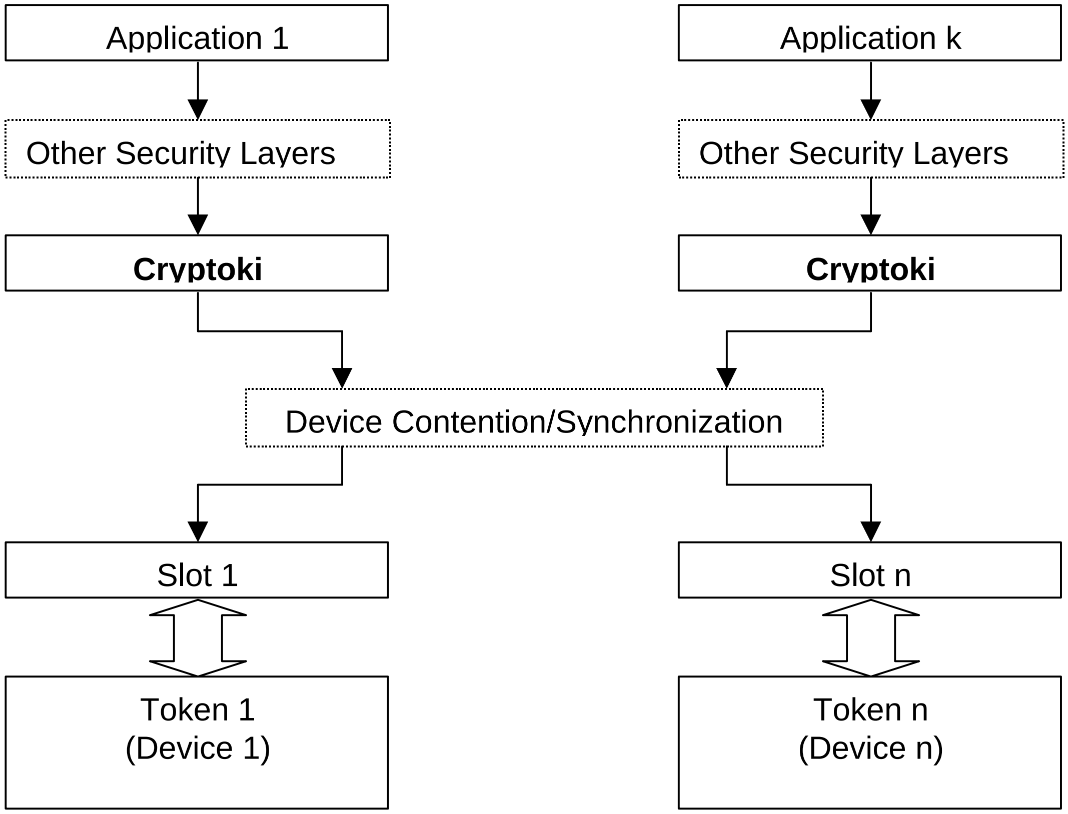

FIGURE 1: GENERAL CRYPTOKI MODEL

Cryptoki provides an interface to one or more cryptographic devices that are active in the system through a number of “slots”. Each slot, which corresponds to a physical reader or other device interface, may contain a token. A token is typically “present in the slot” when a cryptographic device is present in the reader. Of course, since Cryptoki provides a logical view of slots and tokens, there may be other physical interpretations. It is possible that multiple slots may share the same physical reader. The point is that a system has some number of slots, and applications can connect to tokens in any or all of those slots.

A cryptographic device can perform some cryptographic operations, following a certain command set; these commands are typically passed through standard device drivers, for instance PCMCIA card services or socket services. Cryptoki makes each cryptographic device look logically like every other device, regardless of the implementation technology. Thus the application need not interface directly to the device drivers (or even know which ones are involved); Cryptoki hides these details. Indeed, the underlying “device” may be implemented entirely in software (for instance, as a process running on a server)—no special hardware is necessary.

Cryptoki is likely to be implemented as a library supporting the functions in the interface, and applications will be linked to the library. An application may be linked to Cryptoki directly; alternatively, Cryptoki can be a so-called “shared” library (or dynamic link library), in which case the application would link the library dynamically. Shared libraries are fairly straightforward to

 Copyright © OASIS Open 2014. All Rights Reserved.

produce in operating systems such as Microsoft Windows and OS/2, and can be achieved without too much difficulty in UNIX and DOS systems.

The dynamic approach certainly has advantages as new libraries are made available, but from a security perspective, there are some drawbacks. In particular, if a library is easily replaced, then there is the possibility that an attacker can substitute a rogue library that intercepts a user's PIN. From a security perspective, therefore, direct linking is generally preferable, although code-signing techniques can prevent many of the security risks of dynamic linking. In any case, whether the linking is direct or dynamic, the programming interface between the application and a Cryptoki library remains the same.

The kinds of devices and capabilities supported will depend on the particular Cryptoki library. This standard specifies only the interface to the library, not its features. In particular, not all libraries will support all the mechanisms (algorithms) defined in this interface (since not all tokens are expected to support all the mechanisms), and libraries will likely support only a subset of all the kinds of cryptographic devices that are available. (The more kinds, the better, of course, and it is anticipated that libraries will be developed supporting multiple kinds of token, rather than just those from a single vendor.) It is expected that as applications are developed that interface to Cryptoki, standard library and token "profiles" will emerge.

2.3 Logical view of a token

Cryptoki's logical view of a token is a device that stores objects and can perform cryptographic functions. Cryptoki defines three classes of object: data, certificates, and keys. A data object is defined by an application. A certificate object stores a certificate. A key object stores a cryptographic key. The key may be a public key, a private key, or a secret key; each of these types of keys has subtypes for use in specific mechanisms. This view is illustrated in the following figure:

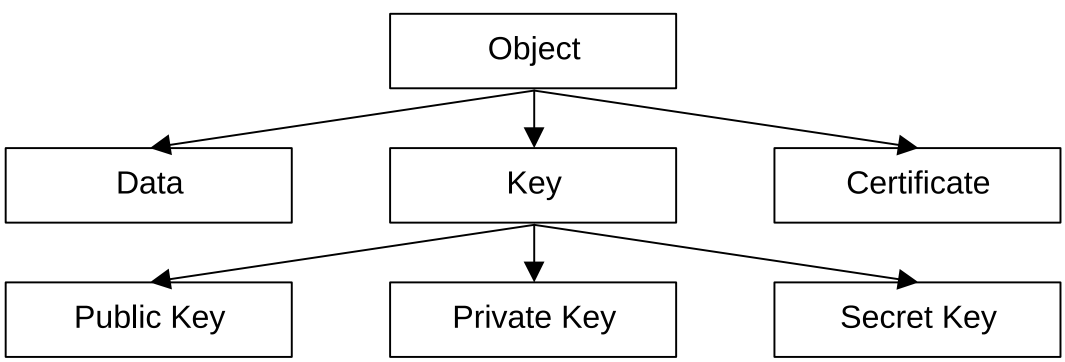

FIGURE 2: OBJECT HIERARCHY

Objects are also classified according to their lifetime and visibility. "Token objects" are visible to all applications connected to the token that have sufficient permission, and remain on the token even after the "sessions" (connections between an application and the token) are closed and the token is removed from its slot. "Session objects" are more temporary: whenever a session is

 Copyright © OASIS Open 2014. All Rights Reserved.

closed by any means, all session objects created by that session are automatically destroyed. In addition, session objects are only visible to the application which created them.

Further classification defines access requirements. Applications are not required to log into the token to view "public objects"; however, to view "private objects", a user must be authenticated to the token by a PIN or some other token-dependent method (for example, a biometric device)..A token can create and destroy objects, manipulate them, and search for them. It can also perform cryptographic functions with objects. A token may have an internal random number generator.

It is important to distinguish between the logical view of a token and the actual implementation, because not all cryptographic devices will have this concept of "objects," or be able to perform every kind of cryptographic function. Many devices will simply have fixed storage places for keys of a fixed algorithm, and be able to do a limited set of operations. Cryptoki's role is to translate this into the logical view, mapping attributes to fixed storage elements and so on. Not all Cryptoki libraries and tokens need to support every object type. It is expected that standard "profiles" will be developed, specifying sets of algorithms to be supported.

"Attributes" are characteristics that distinguish an instance of an object. In Cryptoki, there are general attributes, such as whether the object is private or public. There are also attributes that are specific to a particular type of object, such as a modulus or exponent for RSA keys.

2.4 Users

This version of Cryptoki recognizes two token user types. One type is a Security Officer (SO). The other type is the normal user. Only the normal user is allowed access to private objects on the token, and that access is granted only after the normal user has been authenticated. Some tokens may also require that a user be authenticated before any cryptographic function can be performed on the token, whether or not it involves private objects. The role of the SO is to initialize a token and to set the normal user's PIN (or otherwise define, by some method outside the scope of this version of Cryptoki, how the normal user may be authenticated), and possibly to manipulate some public objects. The normal user cannot log in until the SO has set the normal user's PIN.

Other than the support for two types of user, Cryptoki does not address the relationship between the SO and a community of users. In particular, the SO and the normal user may be the same person or may be different, but such matters are outside the scope of this standard.

With respect to PINs that are entered through an application, Cryptoki assumes only that they are variable-length strings of characters from the set in [PKCS11-BASE] Table 3. Any translation to the device's requirements is left to the Cryptoki library. The following issues are beyond the scope of Cryptoki:

- Any padding of PINs.
- How the PINs are generated (by the user, by the application, or by some other means).

 Copyright © OASIS Open 2014. All Rights Reserved.

PINs that are supplied by some means other than through an application (*e.g.*, PINs entered via a PIN pad on the token) are even more abstract. Cryptoki knows how to wait (if need be) for such a PIN to be supplied and used, and little more.

2.5 Applications and their use of Cryptoki

2.5.1 General Guidance

To Cryptoki, an application consists of a single address space and all the threads of control running in it. An application becomes a "Cryptoki application" by calling the Cryptoki function **C_Initialize** from one of its threads; after this call is made, the application can call other Cryptoki functions. When the application has finished using Cryptoki, it calls the Cryptoki function **C_Finalize** and ceases to be a Cryptoki application.

2.5.2 Applications and processes

In general, on most platforms, the previous paragraph means that an application consists of a single process.

Consider a UNIX process **P** which becomes a Cryptoki application by calling **C_Initialize**, and then uses the fork() system call to create a child process **C**. Since **P** and **C** have separate address spaces (or will when one of them performs a write operation, if the operating system follows the copy-on-write paradigm), they are not part of the same application. Therefore, if **C** needs to use Cryptoki, it needs to perform its own **C_Initialize** call. Furthermore, if **C** needs to be logged into the token(s) that it will access via Cryptoki, it needs to log into them *even if **P** is already logged in*, since **P** and **C** are completely separate applications.

In this particular case (when **C** is the child of a process which is a Cryptoki application), the behavior of Cryptoki is undefined if **C** tries to use it without its own **C_Initialize** call. Ideally, such an attempt would return the value CKR_CRYPTOKI_NOT_INITIALIZED; however, because of the way fork() works, insisting on this return value might have a bad impact on the performance of libraries. Therefore, the behavior of Cryptoki in this situation is left undefined. Applications should definitely *not* attempt to take advantage of any potential "shortcuts" which might (or might not!) be available because of this.

In the scenario specified above, **C** should actually call **C_Initialize** whether or not it needs to use Cryptoki; if it has no need to use Cryptoki, it should then call **C_Finalize** immediately thereafter. This (having the child immediately call **C_Initialize** and then call **C_Finalize** if the parent is using Cryptoki) is considered to be good Cryptoki programming practice, since it can prevent the existence of dangling duplicate resources that were created at the time of the fork() call; however, it is not required by Cryptoki.

2.5.3 Applications and threads

Some applications will access a Cryptoki library in a multi-threaded fashion. Cryptoki enables applications to provide information to libraries so that they can give appropriate support for

 Copyright © OASIS Open 2014. All Rights Reserved.

multi-threading. In particular, when an application initializes a Cryptoki library with a call to **C_Initialize**, it can specify one of four possible multi-threading behaviors for the library:

1. The application can specify that it will not be accessing the library concurrently from multiple threads, and so the library need not worry about performing any type of locking for the sake of thread-safety.
2. The application can specify that it *will* be accessing the library concurrently from multiple threads, and the library must be able to use native operation system synchronization primitives to ensure proper thread-safe behavior.
3. The application can specify that it *will* be accessing the library concurrently from multiple threads, and the library must use a set of application-supplied synchronization primitives to ensure proper thread-safe behavior.
4. The application can specify that it *will* be accessing the library concurrently from multiple threads, and the library must use either the native operating system synchronization primitives or a set of application-supplied synchronization primitives to ensure proper thread-safe behavior.

The 3rd and 4th types of behavior listed above are appropriate for multi-threaded applications that are not using the native operating system thread model. The application-supplied synchronization primitives consist of four functions for handling mutex (*mut*ual *ex*clusion) objects in the application's threading model. Mutex objects are simple objects that can be in either of two states at any given time: unlocked or locked. If a call is made by a thread to lock a mutex that is already locked, that thread blocks (waits) until the mutex is unlocked; then it locks it and the call returns. If more than one thread is blocking on a particular mutex, and that mutex becomes unlocked, then exactly one of those threads will get the lock on the mutex and return control to the caller (the other blocking threads will continue to block and wait for their turn).

See [PKCS11-BASE] Section 5.1.5 for more information on Cryptoki's view of mutex objects.

In addition to providing the above thread-handling information to a Cryptoki library at initialization time, an application can also specify whether or not application threads executing library calls may use native operating system calls to spawn new threads.

2.6 Sessions

2.6.1 General Guidance

Cryptoki requires that an application open one or more sessions with a token to gain access to the token's objects and functions. A session provides a logical connection between the application and the token. A session can be a read/write (R/W) session or a read-only (R/O) session. Read/write and read-only refer to the access to token objects, not to session objects. In both session types, an application can create, read, write and destroy session objects, and read token objects. However, only in a read/write session can an application create, modify, and destroy token objects.

 Copyright © OASIS Open 2014. All Rights Reserved.

After it opens a session, an application has access to the token's public objects. All threads of a given application have access to exactly the same sessions and the same session objects. To gain access to the token's private objects, the normal user must log in and be authenticated.

When a session is closed, any session objects which were created in that session are destroyed. This holds even for session objects that are "being used" by other sessions. That is, if a single application has multiple sessions open with a token, and it uses one of them to create a session object, then that session object is visible through any of that application's sessions. However, as soon as the session that was used to create the object is closed, that object is destroyed.

Cryptoki supports multiple sessions on multiple tokens. An application may have one or more sessions with one or more tokens. In general, a token may have multiple sessions with one or more applications. A particular token may allow an application to have only a limited number of sessions—or only a limited number of read/write sessions-- however.

An open session can be in one of several states. The session state determines allowable access to objects and functions that can be performed on them. The session states are described in Section 2 and Section 3.

2.6.2 Read-only session states

A read-only session can be in one of two states, as illustrated in the following figure. When the session is initially opened, it is in either the "R/O Public Session" state (if the application has no previously open sessions that are logged in) or the "R/O User Functions" state (if the application already has an open session that is logged in). Note that read-only SO sessions do not exist. Read-only sessions that are open while the SO is logged in behave identically to the "R/O Public Session" state.

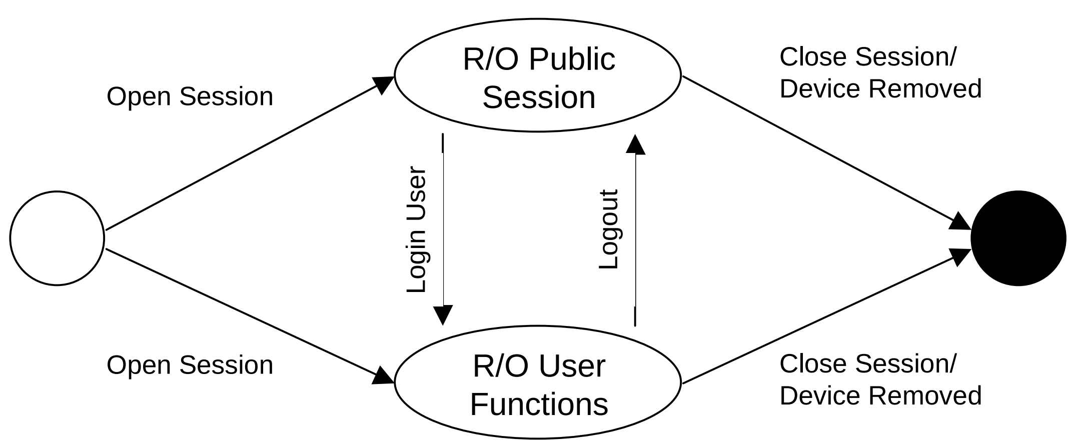

FIGURE 3: READ-ONLY SESSION STATES

The following table describes the session states:

Table 1: Read-Only Session States

 Copyright © OASIS Open 2014. All Rights Reserved.

State	Description
R/O Public Session	The application has opened a read-only session. The application has read-only access to public token objects and read/write access to public session objects.
R/O User Functions	The normal user has been authenticated to the token. The application has read-only access to all token objects (public or private) and read/write access to all session objects (public or private).

2.6.3 Read/write session states

A read/write session can be in one of three states, as illustrated in the following figure. When the session is opened, it is in either the "R/W Public Session" state (if the application has no previously open sessions that are logged in), the "R/W User Functions" state (if the application already has an open session that the normal user is logged into), or the "R/W SO Functions" state (if the application already has an open session that the SO is logged into).

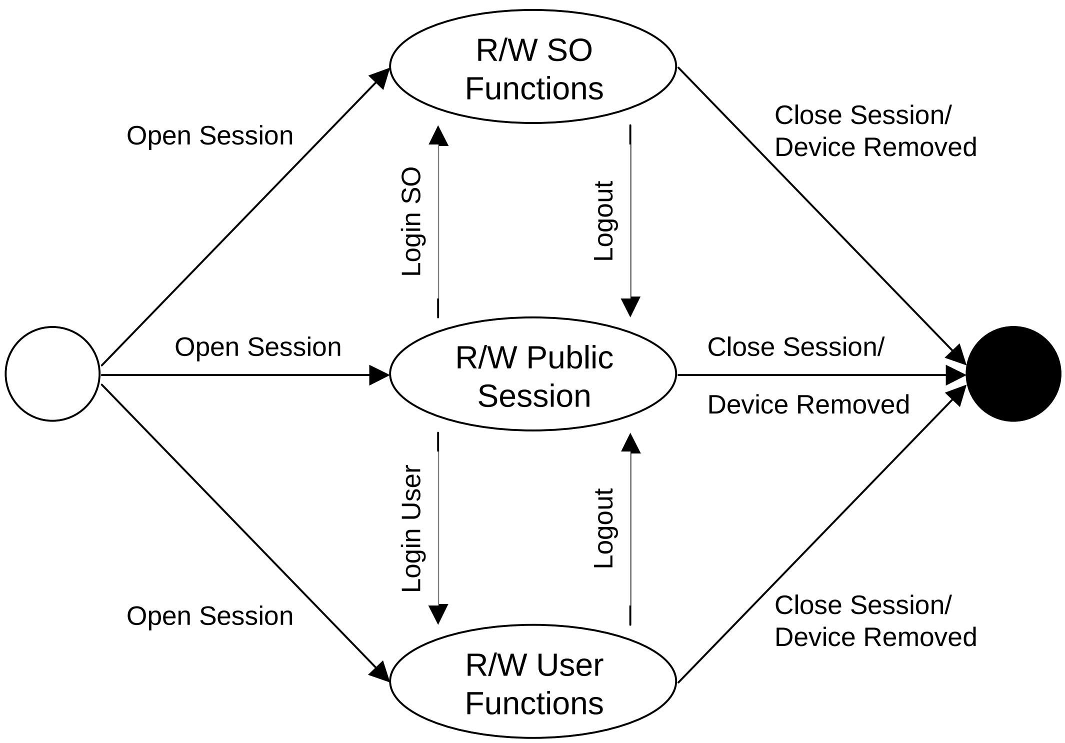

FIGURE 4: READ/WRITE SESSION STATES

The following table describes the session states:

 Copyright © OASIS Open 2014. All Rights Reserved.

Table 2: Read/Write Session States

State	Description
R/W Public Session	The application has opened a read/write session. The application has read/write access to all public objects.
R/W SO Functions	The Security Officer has been authenticated to the token. The application has read/write access only to public objects on the token, not to private objects. The SO can set the normal user's PIN.
R/W User Functions	The normal user has been authenticated to the token. The application has read/write access to all objects.

2.6.4 Permitted object accesses by sessions

The following table summarizes the kind of access each type of session has to each type of object. A given type of session has either read-only access, read/write access, or no access whatsoever to a given type of object.

Note that creating or deleting an object requires read/write access to it, *e.g.*, a "R/O User Functions" session cannot create or delete a token object.

Table 3: Access to Different Types Objects by Different Types of Sessions

	Type of session				
Type of object	**R/O Public**	**R/W Public**	**R/O User**	**R/W User**	**R/W SO**
Public session object	R/W	R/W	R/W	R/W	R/W
Private session object			R/W	R/W	
Public token object	R/O	R/W	R/O	R/W	R/W
Private token object			R/O	R/W	

As previously indicated, the access to a given session object which is shown in Table is limited to sessions belonging to the application which owns that object (*i.e.*, which created that object).

2.6.5 Session events

Session events cause the session state to change. The following table describes the events:

Table 4: Session Events

Event	Occurs when...
Log In SO	the SO is authenticated to the token.
Log In User	the normal user is authenticated to the token.
Log Out	the application logs out the current user (SO or normal user).
Close Session	the application closes the session or closes all sessions.

 Copyright © OASIS Open 2014. All Rights Reserved.

Event	Occurs when...
Device Removed	the device underlying the token has been removed from its slot.

When the device is removed, all sessions of all applications are automatically logged out. Furthermore, all sessions any applications have with the device are closed (this latter behavior was not present in Version 1.0 of Cryptoki)—an application cannot have a session with a token that is not present. Realistically, Cryptoki may not be constantly monitoring whether or not the token is present, and so the token's absence could conceivably not be noticed until a Cryptoki function is executed. If the token is re-inserted into the slot before that, Cryptoki might never know that it was missing.

In Cryptoki, all sessions that an application has with a token must have the same login/logout status (*i.e.*, for a given application and token, one of the following holds: all sessions are public sessions; all sessions are SO sessions; or all sessions are user sessions). When an application's session logs into a token, *all* of that application's sessions with that token become logged in, and when an application's session logs out of a token, *all* of that application's sessions with that token become logged out. Similarly, for example, if an application already has a R/O user session open with a token, and then opens a R/W session with that token, the R/W session is automatically logged in.

This implies that a given application may not simultaneously have SO sessions and user sessions open with a given token.

2.6.6 Session handles and object handles

A session handle is a Cryptoki-assigned value that identifies a session. It is in many ways akin to a file handle, and is specified to functions to indicate which session the function should act on. All threads of an application have equal access to all session handles. That is, anything that can be accomplished with a given file handle by one thread can also be accomplished with that file handle by any other thread of the same application.

Cryptoki also has object handles, which are identifiers used to manipulate Cryptoki objects. Object handles are similar to session handles in the sense that visibility of a given object through an object handle is the same among all threads of a given application. R/O sessions, of course, only have read-only access to token objects, whereas R/W sessions have read/write access to token objects.

Valid session handles and object handles in Cryptoki always have nonzero values. For developers' convenience, Cryptoki defines the following symbolic value:

```
CK_INVALID_HANDLE
```

2.6.7 Capabilities of sessions

Very roughly speaking, there are three broad types of operations an open session can be used to perform: administrative operations (such as logging in); object management operations (such as

 Copyright © OASIS Open 2014. All Rights Reserved.

creating or destroying an object on the token); and cryptographic operations (such as computing a message digest). Cryptographic operations sometimes require more than one function call to the Cryptoki API to complete. In general, a single session can perform only one operation at a time; for this reason, it may be desirable for a single application to open multiple sessions with a single token. For efficiency's sake, however, a single session on some tokens can perform the following pairs of operation types simultaneously: message digesting and encryption; decryption and message digesting; signature or MACing and encryption; and decryption and verifying signatures or MACs. Details on performing simultaneous cryptographic operations in one session are provided in [PKCS11-Base] Section 5.12.

A consequence of the fact that a single session can, in general, perform only one operation at a time is that *an application should never make multiple simultaneous function calls to Cryptoki which use a common session*. If multiple threads of an application attempt to use a common session concurrently in this fashion, Cryptoki does not define what happens. This means that if multiple threads of an application all need to use Cryptoki to access a particular token, it might be appropriate for each thread to have its own session with the token, unless the application can ensure by some other means (*e.g.*, by some locking mechanism) that no sessions are ever used by multiple threads simultaneously. This is true regardless of whether or not the Cryptoki library was initialized in a fashion which permits safe multi-threaded access to it. Even if it is safe to access the library from multiple threads simultaneously, it is still not necessarily safe to use *a particular session* from multiple threads simultaneously.

2.6.8 Example of use of sessions

We give here a detailed and lengthy example of how multiple applications can make use of sessions in a Cryptoki library. Despite the somewhat painful level of detail, we highly recommend reading through this example carefully to understand session handles and object handles.

We caution that our example is decidedly *not* meant to indicate how multiple applications *should* use Cryptoki simultaneously; rather, it is meant to clarify what uses of Cryptoki's sessions and objects and handles are permissible. In other words, instead of demonstrating good technique here, we demonstrate "pushing the envelope".

For our example, we suppose that two applications, **A** and **B**, are using a Cryptoki library to access a single token **T**. Each application has two threads running: **A** has threads **A1** and **A2**, and **B** has threads **B1** and **B2**. We assume in what follows that there are no instances where multiple threads of a single application simultaneously use the same session, and that the events of our example occur in the order specified, without overlapping each other in time.

1. **A1** and **B1** each initialize the Cryptoki library by calling **C_Initialize** (the specifics of Cryptoki functions will be explained in Section **Error! Reference source not found.**). Note that exactly one call to **C_Initialize** should be made for each application (as opposed to one call for every thread, for example).
2. **A1** opens a R/W session and receives the session handle 7 for the session. Since this is the first session to be opened for **A**, it is a public session.

 Copyright © OASIS Open 2014. All Rights Reserved.

3. **A2** opens a R/O session and receives the session handle 4. Since all of **A**'s existing sessions are public sessions, session 4 is also a public session.
4. **A1** attempts to log the SO into session 7. The attempt fails, because read-only sessions cannot be used to log in the SO.
5. **A2** logs the normal user into session 7. This turns session 7 into a R/W user session, and turns session 4 into a R/O user session. Note that because **A1** and **A2** belong to the same application, they have equal access to all sessions, and therefore, **A2** is able to perform this action.
6. **A2** opens a R/W session and receives the session handle 9. Since all of **A**'s existing sessions are user sessions, session 9 is also a user session.
7. **A1** closes session 9.
8. **B1** attempts to log out session 4. The attempt fails, because **A** and **B** have no access rights to each other's sessions or objects. **B1** receives an error message which indicates that there is no such session handle (CKR_SESSION_HANDLE_INVALID).
9. **B2** attempts to close session 4. The attempt fails in precisely the same way as **B1**'s attempt to log out session 4 failed (*i.e.*, **B2** receives a CKR_SESSION_HANDLE_INVALID error code).
10. **B1** opens a R/W session and receives the session handle 7. Note that, as far as **B** is concerned, this is the first occurrence of session handle 7. **A**'s session 7 and **B**'s session 7 are completely different sessions.
11. **B1** logs the SO into [**B**'s] session 7. This turns **B**'s session 7 into a R/W SO session, and has no effect on either of **A**'s sessions.
12. **A1** uses [**A**'s] session 7 to create a session object **O1** of some sort and receives the object handle 7. Note that a Cryptoki implementation may or may not support separate spaces of handles for sessions and objects.
13. **B1** uses [**B**'s] session 7 to create a token object **O2** of some sort and receives the object handle 7. As with session handles, different applications have no access rights to each other's object handles, and so **B**'s object handle 7 is entirely different from **A**'s object handle 7. Of course, since **B1** is an SO session, it cannot create private objects, and so **O2** must be a public object (if **B1** attempted to create a private object, the attempt would fail with error code CKR_USER_NOT_LOGGED_IN or CKR_TEMPLATE_INCONSISTENT).
14. **B2** uses [**B**'s] session 7 to perform some operation to modify the object associated with [**B**'s] object handle 7. This modifies **O2**.
15. **A1** uses [**A**'s] session 4 to perform an object search operation to get a handle for **O2**. The search returns object handle 1. Note that **A**'s object handle 1 and **B**'s object handle 7 now point to the same object.
16. **A1** attempts to use [**A**'s] session 4 to modify the object associated with [**A**'s] object handle 1. The attempt fails, because **A**'s session 4 is a R/O session, and is therefore incapable of modifying **O2**, which is a token object. **A1** receives an error message indicating that the session is a R/O session (CKR_SESSION_READ_ONLY).
17. **A1** uses [**A**'s] session 7 to modify the object associated with [**A**'s] object handle 1. This time, since **A**'s session 7 is a R/W session, the attempt succeeds in modifying **O2**.
18. **B1** uses [**B**'s] session 7 to perform an object search operation to find **O1**. Since **O1** is a session object belonging to **A**, however, the search does not succeed.

 Copyright © OASIS Open 2014. All Rights Reserved.

19. **A2** uses [**A**'s] session 4 to perform some operation to modify the object associated with [**A**'s] object handle 7. This operation modifies **O1**.
20. **A2** uses [**A**'s] session 7 to destroy the object associated with [**A**'s] object handle 1. This destroys **O2**.
21. **B1** attempts to perform some operation with the object associated with [**B**'s] object handle 7. The attempt fails, since there is no longer any such object. **B1** receives an error message indicating that its object handle is invalid (CKR_OBJECT_HANDLE_INVALID).
22. **A1** logs out [**A**'s] session 4. This turns **A**'s session 4 into a R/O public session, and turns **A**'s session 7 into a R/W public session.
23. **A1** closes [**A**'s] session 7. This destroys the session object **O1**, which was created by **A**'s session 7.
24. **A2** attempt to use [**A**'s] session 4 to perform some operation with the object associated with [**A**'s] object handle 7. The attempt fails, since there is no longer any such object. It returns a CKR_OBJECT_HANDLE_INVALID.
25. **A2** executes a call to **C_CloseAllSessions**. This closes [**A**'s] session 4. At this point, if **A** were to open a new session, the session would not be logged in (*i.e.*, it would be a public session).
26. **B2** closes [**B**'s] session 7. At this point, if **B** were to open a new session, the session would not be logged in.
27. **A** and **B** each call **C_Finalize** to indicate that they are done with the Cryptoki library.

Modules implementing previous versions of PKCS #11 may return the CKR_SESSION_READ_ONLY_EXISTS and CKR_SESSION_READ_WRITE_SO_EXISTS error codes.

 Copyright © OASIS Open 2014. All Rights Reserved.

3 Security considerations

3.1 General Guidance

As an interface to cryptographic devices, Cryptoki provides a basis for security in a computer or communications system. Two of the particular features of the interface that facilitate such security are the following:

1. Access to private objects on the token, and possibly to cryptographic functions and/or certificates on the token as well, requires a PIN. Thus, possessing the cryptographic device that implements the token may not be sufficient to use it; the PIN may also be needed.
2. Additional protection can be given to private keys and secret keys by marking them as "sensitive" or "unextractable". Sensitive keys cannot be revealed in plaintext off the token, and unextractable keys cannot be revealed off the token even when encrypted (though they can still be used as keys).

It is expected that access to private, sensitive, or unextractable objects by means other than Cryptoki (*e.g.*, other programming interfaces, or reverse engineering of the device) would be difficult.

If a device does not have a tamper-proof environment or protected memory in which to store private and sensitive objects, the device may encrypt the objects with a master key which is perhaps derived from the user's PIN. The particular mechanism for protecting private objects is left to the device implementation, however.

Based on these features it should be possible to design applications in such a way that the token can provide adequate security for the objects the applications manage.

Of course, cryptography is only one element of security, and the token is only one component in a system. While the token itself may be secure, one must also consider the security of the operating system by which the application interfaces to it, especially since the PIN may be passed through the operating system. This can make it easy for a rogue application on the operating system to obtain the PIN; it is also possible that other devices monitoring communication lines to the cryptographic device can obtain the PIN. Rogue applications and devices may also change the commands sent to the cryptographic device to obtain services other than what the application requested.

It is important to be sure that the system is secure against such attack. Cryptoki may well play a role here; for instance, a token may be involved in the "booting up" of the system.

We note that none of the attacks just described can compromise keys marked "sensitive," since a key that is sensitive will always remain sensitive. Similarly, a key that is unextractable cannot be modified to be extractable.

An application may also want to be sure that the token is "legitimate" in some sense (for a variety of reasons, including export restrictions and basic security). This is outside the scope of

 Copyright © OASIS Open 2014. All Rights Reserved.

the present standard, but it can be achieved by distributing the token with a built-in, certified public/private-key pair, by which the token can prove its identity. The certificate would be signed by an authority (presumably the one indicating that the token is "legitimate") whose public key is known to the application. The application would verify the certificate and challenge the token to prove its identity by signing a time-varying message with its built-in private key.

Once a normal user has been authenticated to the token, Cryptoki does not restrict which cryptographic operations the user may perform; the user may perform any operation supported by the token. Some tokens may not even require any type of authentication to make use of its cryptographic functions.

3.2 Padded Oracle Attacks

To protect against chosen ciphertext attacks, like the Bleichenbacher attack, use PKCS #1 Version 2, with OAEP, and disable support for PKCS #1, Version 1.5..

Furthermore, more specifically to smart card implementations, the requirement of the PIN and a long open connection to the device is required to execute the attack. For smartcard implementations, execution of these attacks requires private key operations and a sufficiently long open connection. It is strongly recommended that any applets exposing private key operations are protected using an encrypted PIN (a PIN not submitted in the clear), and the session is closed when not in use.

 Copyright © OASIS Open 2014. All Rights Reserved.

4 Cryptoki tips and reminders

4.1 Operations, sessions, and threads

In Cryptoki, there are several different types of operations which can be "active" in a session. An active operation is essentially one which takes more than one Cryptoki function call to perform. The types of active operations are object searching; encryption; decryption; message-digesting; signature with appendix; signature with recovery; verification with appendix; and verification with recovery.

A given session can have 0, 1, or 2 operations active at a time. It can only have 2 operations active simultaneously if the token supports this; moreover, those two operations must be one of the four following pairs of operations: digesting and encryption; decryption and digesting; signing and encryption; decryption and verification.

If an application attempts to initialize an operation (make it active) in a session, but this cannot be accomplished because of some other active operation(s), the application receives the error value CKR_OPERATION_ACTIVE. This error value can also be received if a session has an active operation and the application attempts to use that session to perform any of various operations which do not become "active", but which require cryptographic processing, such as using the token's random number generator, or generating/wrapping/unwrapping/deriving a key.

To abandon an active operation an application may have to complete the operation and discard the result. Closing the session will also have this effect. Alternatively, the library may allow active operations to be abandoned by the application, simply by allowing initialization for some other operation. In this case CKR_OPERATION_ACTIVE will not be returned but the previous active operation will be unusable.

Different threads of an application should never share sessions, unless they are extremely careful not to make function calls at the same time. This is true even if the Cryptoki library was initialized with locking enabled for thread-safety.

4.2 Multiple Application Access Behavior

When multiple applications, or multiple threads within an application, are accessing a set of common objects the issue of object protection becomes important. This is especially the case when application A activates an operation using object O, and application B attempts to delete O before application A has finished the operation. Unfortunately, variation in device capabilities makes an absolute behavior specification impractical. General guidelines are presented here for object protection behavior.

Whenever possible, deleting an object in one application should not cause that object to become unavailable to another application or thread that is using the object in an active operation until that operation is complete. For instance, application A has begun a signature

 Copyright © OASIS Open 2014. All Rights Reserved.

operation with private key P and application B attempts to delete P while the signature is in progress. In this case, one of two things should happen. The object is deleted from the device but the operation is allow to complete because the operation uses a temporary copy of the object, or the delete operation blocks until the signature operation has completed. If neither of these actions can be supported by an implementation, then the error code CKR_OBJECT_HANDLE_INVALID may be returned to application A to indicate that the key being used to perform its active operation has been deleted.

Whenever possible, changing the value of an object attribute should impact the behavior of active operations in other applications or threads. If this cannot be supported by an implementation, then the appropriate error code indicating the reason for the failure should be returned to the application with the active operation.

4.3 Objects, attributes, and templates

In general, a Cryptoki function which requires a template for an object needs the template to specify—either explicitly or implicitly—any attributes that are not specified elsewhere. If a template specifies a particular attribute more than once, the function can return CKR_TEMPLATE_INVALID or it can choose a particular value of the attribute from among those specified and use that value. In any event, object attributes are always single-valued.

4.4 Signing with recovery

Signing with recovery is a general alternative to ordinary digital signatures ("signing with appendix") which is supported by certain mechanisms. Recall that for ordinary digital signatures, a signature of a message is computed as some function of the message and the signer's private key; this signature can then be used (together with the message and the signer's public key) as input to the verification process, which yields a simple "signature valid/signature invalid" decision.

Signing with recovery also creates a signature from a message and the signer's private key. However, to verify this signature, no message is required as input. Only the signature and the signer's public key are input to the verification process, and the verification process outputs either "signature invalid" or—if the signature is valid—the original message.

Consider a simple example with the **CKM_RSA_X_509** mechanism. Here, a message is a byte string which we will consider to be a number modulo *n* (the signer's RSA modulus). When this mechanism is used for ordinary digital signatures (signatures with appendix), a signature is computed by raising the message to the signer's private exponent modulo *n*. To verify this signature, a verifier raises the signature to the signer's public exponent modulo *n*, and accepts the signature as valid if and only if the result matches the original message.

If **CKM_RSA_X_509** is used to create signatures with recovery, the signatures are produced in exactly the same fashion. For this particular mechanism, *any* number modulo *n* is a valid signature. To recover the message from a signature, the signature is raised to the signer's public exponent modulo *n*.

 Copyright © OASIS Open 2014. All Rights Reserved.

5 Comparison of Cryptoki and other APIs

5.1 FORTEZZA CIPG

The following table lists the FORTEZZA CIPG functions, together with the equivalent Cryptoki functions,. See [FORTEZZA] for more information on the FORTEZZA API.

TABLE 5: FORTEZZA CIPG VS. CRYPTOKI

FORTEZZA CIPG	Equivalent Cryptoki
CI_ChangePIN	C_InitPIN, C_SetPIN
CI_CheckPIN	C_Login
CI_Close	C_CloseSession
CI_Decrypt	C_DecryptInit, C_Decrypt, C_DecryptUpdate, C_DecryptFinal
CI_DeleteCertificate	C_DestroyObject
CI_DeleteKey	C_DestroyObject
CI_Encrypt	C_EncryptInit, C_Encrypt, C_EncryptUpdate, C_EncryptFinal
CI_ExtractX	C_WrapKey
CI_GenerateIV	C_GenerateRandom
CI_GenerateMEK	C_GenerateKey
CI_GenerateRa	C_GenerateRandom
CI_GenerateRandom	C_GenerateRandom
CI_GenerateTEK	C_GenerateKey
CI_GenerateX	C_GenerateKeyPair
CI_GetCertificate	C_FindObjects
CI_Configuration	C_GetTokenInfo
CI_GetHash	C_DigestInit, C_Digest, C_DigestUpdate, and C_DigestFinal
CI_GetIV	No equivalent
CI_GetPersonalityList	C_FindObjects
CI_GetState	C_GetSessionInfo
CI_GetStatus	C_GetTokenInfo
CI_GetTime	C_GetTokenInfo or C_GetAttributeValue(clock object) ***[preferred]***
CI_Hash	C_DigestInit, C_Digest, C_DigestUpdate, and C_DigestFinal
CI_Initialize	C_Initialize
CI_InitializeHash	C_DigestInit
CI_InstallX	C_UnwrapKey
CI_LoadCertificate	C_CreateObject
CI_LoadDSAParameters	C_CreateObject
CI_LoadInitValues	C_SeedRandom
CI_LoadIV	C_EncryptInit, C_DecryptInit
CI_LoadK	C_SignInit
CI_LoadPublicKeyParameters	C_CreateObject

 Copyright © OASIS Open 2014. All Rights Reserved.

FORTEZZA CIPG	Equivalent Cryptoki
CI_LoadPIN	C_SetPIN
CI_LoadX	C_CreateObject
CI_Lock	Implicit in session management
CI_Open	C_OpenSession
CI_RelayX	C_WrapKey
CI_Reset	C_CloseAllSessions
CI_Restore	Implicit in session management
CI_Save	Implicit in session management
CI_Select	C_OpenSession
CI_SetConfiguration	No equivalent
CI_SetKey	C_EncryptInit, C_DecryptInit
CI_SetMode	C_EncryptInit, C_DecryptInit
CI_SetPersonality	C_CreateObject
CI_SetTime	No equivalent
CI_Sign	C_SignInit, C_Sign
CI_Terminate	C_CloseAllSessions
CI_Timestamp	C_SignInit, C_Sign
CI_Unlock	Implicit in session management
CI_UnwrapKey	C_UnwrapKey
CI_VerifySignature	C_VerifyInit, C_Verify
CI_VerifyTimestamp	C_VerifyInit, C_Verify
CI_WrapKey	C_WrapKey
CI_Zeroize	C_InitToken

5.2 GCS-API

This proposed standard defines an API to high-level security services such as authentication of identities and data-origin, non-repudiation, and separation and protection. It is at a higher level than Cryptoki. The following table lists the GCS-API functions with the Cryptoki functions used to implement the functions. Note that full support of GCS-API is left for future versions of Cryptoki. See [GCS-API] for more information on the API.

TABLE 6: GCS-API VS. CRYPTOKI

GCS-API	Cryptoki implementation
retrieve_CC	
release_CC	
generate_hash	C_DigestInit, C_Digest
generate_random_number	C_GenerateRandom
generate_checkvalue	C_SignInit, C_Sign, C_SignUpdate, C_SignFinal
verify_checkvalue	C_VerifyInit, C_Verify, C_VerifyUpdate, C_VerifyFinal
data_encipher	C_EncryptInit, C_Encrypt, C_EncryptUpdate, C_EncryptFinal
data_decipher	C_DecryptInit, C_Decrypt, C_DecryptUpdate, C_DecryptFinal

 Copyright © OASIS Open 2014. All Rights Reserved.

GCS-API	Cryptoki implementation
create_CC	
derive_key	C_DeriveKey
generate_key	C_GenerateKey
store_CC	
delete_CC	
replicate_CC	
export_key	C_WrapKey
import_key	C_UnwrapKey
archive_CC	C_WrapKey
restore_CC	C_UnwrapKey
set_key_state	
generate_key_pattern	
verify_key_pattern	
derive_clear_key	C_DeriveKey
generate_clear_key	C_GenerateKey
load_key_parts	
clear_key_encipher	C_WrapKey
clear_key_decipher	C_UnwrapKey
change_key_context	
load_initial_key	
generate_initial_key	
set_current_master_key	
protnder_new_master_key	
proect_under_current_master_key	
initialise_random_number_generator	C_SeedRandom
install_algorithm	
de_install_algorithm	
disable_algorithm	
enable_algorithm	
set_defaults	

 Copyright © OASIS Open 2014. All Rights Reserved.

6 Deprecated PKCS #11 Functionality

6.1 Secondary authentication (Deprecated)

Note:	This support may be present for backwards compatibility. Refer to [PKCS#11- 2.1]1 for details.

6.2 Method for Exposing Multiple-PINs on a Token Through Cryptoki (deprecated)

Note:	This support may be present for backwards compatibility. Refer to [PKCS11-V 2.11] for details.

6.3 Non-Normative Token Profiles

6.3.1 Description of this Section

This section describes sample "profiles," *i.e.*, sets of mechanisms, which a token should support for various common types of application that were defined in PKCS #11 V2.30. It is expected that these sets would be standardized as parts of the various applications, for instance within a list of requirements on the module that provides cryptographic services to the application (which may be a Cryptoki token in some cases). Thus, these profiles are intended for reference only at this point, and are not part of this standard.

The following table summarizes the mechanisms relevant to two common types of applications:

 Copyright © OASIS Open 2014. All Rights Reserved.

Table 7: Mechanisms and profiles

Mechanism	Application	
	Government Authentication-only	Cellular Digital Packet Data
CKM_DSA_KEY_PAIR_GEN	✓	
CKM_DSA	✓	
CKM_DH_PKCS_KEY_PAIR_GEN		✓
CKM_DH_PKCS_DERIVE		✓
CKM_RC4_KEY_GEN		✓
CKM_RC4		✓
CKM_SHA_1	✓	

6.3.2 Government authentication-only

The U.S. government has standardized on the Digital Signature Algorithm as defined in FIPS PUB 186-2 for signatures and the Secure Hash Algorithm as defined in FIPS PUB 180-2 for message digesting. The relevant mechanisms include the following:

DSA key generation (512-1024 bits)

DSA (512-1024 bits)

SHA-1

6.3.3 Cellular Digital Packet Data

Cellular Digital Packet Data (CDPD) is a set of protocols for wireless communication. The basic set of mechanisms to support CDPD applications includes the following:

Diffie-Hellman key generation (256-1024 bits)

Diffie-Hellman key derivation (256-1024 bits)

RC4 key generation (40-128 bits)

RC4 (40-128 bits)

(The initial CDPD security specification limits the size of the Diffie-Hellman key to 256 bits, but it has been recommended that the size be increased to at least 512 bits.)

6.3.4 Other profiles

The reader is also informed of the presence of other sample profiles defined prior to PKCS #11 v2.40. See the documentation for previous versions of PKCS #11.

 Copyright © OASIS Open 2014. All Rights Reserved.

Appendix A. Acknowledgments

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:

Gil Abel, Athena Smartcard Solutions, Inc.
Warren Armstrong, QuintessenceLabs
Peter Bartok, Venafi, Inc.
Anthony Berglas, Cryptsoft
Kelley Burgin, National Security Agency
Robert Burns, Thales e-Security
Wan-Teh Chang, Google Inc.
Hai-May Chao, Oracle
Janice Cheng, Vormetric, Inc.
Sangrae Cho, Electronics and Telecommunications Research Institute (ETRI)
Doron Cohen, SafeNet, Inc.
Fadi Cotran, Futurex
Tony Cox, Cryptsoft
Christopher Duane, EMC
Chris Dunn, SafeNet, Inc.
Valerie Fenwick, Oracle
Terry Fletcher, SafeNet, Inc.
Susan Gleeson, Oracle
Sven Gossel, Charismathics
Robert Griffin, EMC
Paul Grojean, Individual
Peter Gutmann, Individual
Dennis E. Hamilton, Individual
Thomas Hardjono, M.I.T.
Tim Hudson, Cryptsoft
Gershon Janssen, Individual
Seunghun Jin, Electronics and Telecommunications Research Institute (ETRI)
Andrey Jivsov, Symantec Corp.
Greg Kazmierczak, Wave Systems Corp.
Mark Knight, Thales e-Security
Darren Krahn, Google Inc.
Alex Krasnov, Infineon Technologies AG
Dina Kurktchi-Nimeh, Oracle
Mark Lambiase, SecureAuth Corporation
Lawrence Lee, GoTrust Technology Inc.
John Leiseboer, QuintessenceLabs
Hal Lockhart, Oracle
Robert Lockhart, Thales e-Security
Dale Moberg, Axway Software
Darren Moffat, Oracle

 Copyright © OASIS Open 2014. All Rights Reserved.

Valery Osheter, SafeNet, Inc.
Sean Parkinson, EMC
Rob Philpott, EMC
Mark Powers, Oracle
Ajai Puri, SafeNet, Inc.
Robert Relyea, Red Hat
Saikat Saha, Oracle
Subhash Sankuratripati, NetApp
Johann Schoetz, Infineon Technologies AG
Rayees Shamsuddin, Wave Systems Corp.
Radhika Siravara, Oracle
Brian Smith, Mozilla Corporation
David Smith, Venafi, Inc.
Ryan Smith, Futurex
Jerry Smith, US Department of Defense (DoD)
Oscar So, Oracle
Michael Stevens, QuintessenceLabs
Michael StJohns, Individual
Sander Temme, Thales e-Security
Kiran Thota, VMware, Inc.
Walter-John Turnes, Gemini Security Solutions, Inc.
Stef Walter, Red Hat
Jeff Webb, Dell
Magda Zdunkiewicz, Cryptsoft
Chris Zimman, Bloomberg Finance L.P.

 Copyright © OASIS Open 2014. All Rights Reserved.

Appendix B. Revision History

Revision	Date	Editor	Changes Made
wd01	18 March 2013	John Leiseboer	Initial version
wd02	10 June 2013	John Leiseboer	Incorporated usage information from PCKS #11 Base Specification V2.30
wd03	10 September 2013	Robert Griffin	Incorporated new sections from approved ballots
wd04	16 September 2013	Robert Griffin	Updated participants list
wd05	1 October 2013	Robert Griffin	Removed design goals section as discussed in face-to-face meeting.
wd06	27 October 2013	John Leiseboer / Robert Griffin	Final participant list and other editorial changes for Committee Note Draft
wd07	15 February 2014	Robert Griffin	Corrections and changes from public review

Copyright © OASIS Open 2014. All Rights Reserved.