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

PKCS #11 Cryptographic Token Interface Usage Guide Version 2.40

Working Draft 02

10 June 2013

Technical Committee:
OASIS PKCS 11 TC

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

Editor:
John Leiseboer (jl@quintessencelabs.com), QuintessenceLabs Pty Ltd.

Related work:
This document is related to:

- *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.
- *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.
- *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.
- *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.
- *PKCS #11 Cryptographic Token Interface Test Cases Version 2.40*. Latest version. http://docs.oasis-open.org/pkcs11/pkcs11-testcases/v2.40/pkcs11-testcases-v2.40.html.

Abstract:
Summary of the purpose of the document

Status:
This Working Draft (WD) has been produced by one or more TC Members; it has not yet been voted on by the TC or approved as a Committee Note Draft. The OASIS document Approval Process begins officially with a TC vote to approve a WD as a Committee Note Draft. A TC may approve a

Working Draft, revise it, and re-approve it any number of times as a Committee Note Draft.

Initial URI pattern:
http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/cnd01/pkcs11-ug-v2.40-cnd01.doc

(Managed by OASIS TC Administration; please don't modify.)

Table of Contents

1 Introduction

This PKCS #11 Cryptographic Token Interface Usage Guide Version 2.40 is intended to complement **[PKCS11-Base], PKCS11-Curr]** and **PKCS11-Hist]** 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**]. Further assistance for implementing PKCS #11 is provided by [**PKCS11-TC**].

1.1 Terminology

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

1.2 References (normative)

[PKCS11-Base] PKCS #11 Cryptographic Token Interface Base Specification Version 2.40. wd01 30 April 2013 *Working Draft* https://www.oasis-open.org/apps/org/workgroup/pkcs11/download.php/49020/pkcs11-base-v2%2040-wd01.pdf

[PKCS11-Curr] PKCS #11 Cryptographic Token Interface Current Mechanisms Version 2.40. wd01 30 April 2013 *Working Draft* https://www.oasis-open.org/apps/org/workgroup/pkcs11/download.php/49021/pkcs11-curr-v2.40-wd01.pdf

[PKCS11-Hist] PKCS #11 Cryptographic Token Interface Current Mechanisms Version 2.40. wd02 16 May 2013 *Working Draft* https://www.oasis-open.org/apps/org/workgroup/pkcs11/download.php/49216/pkcs11-hist-v2_40-wd01.pdf

[PKCS11-Prof] PKCS #11 Cryptographic Token Interface Profiles Version 2.40. wd02 03 April 2013 *Working Draft* https://www.oasis-open.org/apps/org/workgroup/pkcs11/download.php/48726/pkcs11-profiles-v2.40-wd02.doc

See [PKCS-Spec] for additional normative references.

1.3 References (non-normative)

[PKCS-TC] PKCS #11 Cryptographic Token Interface Test Cases Version 2.40. tbd

See [PKCS-Spec] for additional non-normative references.

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 need.

2.2 Design goals

Cryptoki was intended from the beginning to be an interface between applications and all kinds of portable cryptographic devices, such as those based on smart cards, PCMCIA cards, and smart diskettes. There are already standards (de facto or official) for interfacing to these devices at some level. For instance, the mechanical characteristics and electrical connections are well-defined, as are the methods for supplying commands and receiving results. (See, for example, ISO 7816, or the PCMCIA specifications.)

What remained to be defined were particular commands for performing cryptography. It would not be enough simply to define command sets for each kind of device, as that would not solve the general problem of an *application* interface independent of the device. To do so is still a long-term goal, and would certainly contribute to interoperability. The primary goal of Cryptoki was a lower-level programming interface that abstracts the details of the devices, and presents to the application a common model of the cryptographic device, called a “cryptographic token” (or simply “token”).

A secondary goal was resource-sharing. As desktop multi-tasking operating systems become more popular, a single device should be shared between more than one application. In addition, an application should be able to interface to more than one device at a given time.

It is not the goal of Cryptoki to be a generic interface to cryptographic operations or security services, although one certainly could build such operations and services with the functions that Cryptoki provides. Cryptoki is intended to complement, not compete with, such emerging and evolving interfaces as “Generic Security Services Application Programming Interface” (RFC 2743 and RFC 2744) and “Generic Cryptographic Service API” (GCS-API) from X/Open.

2.3 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.

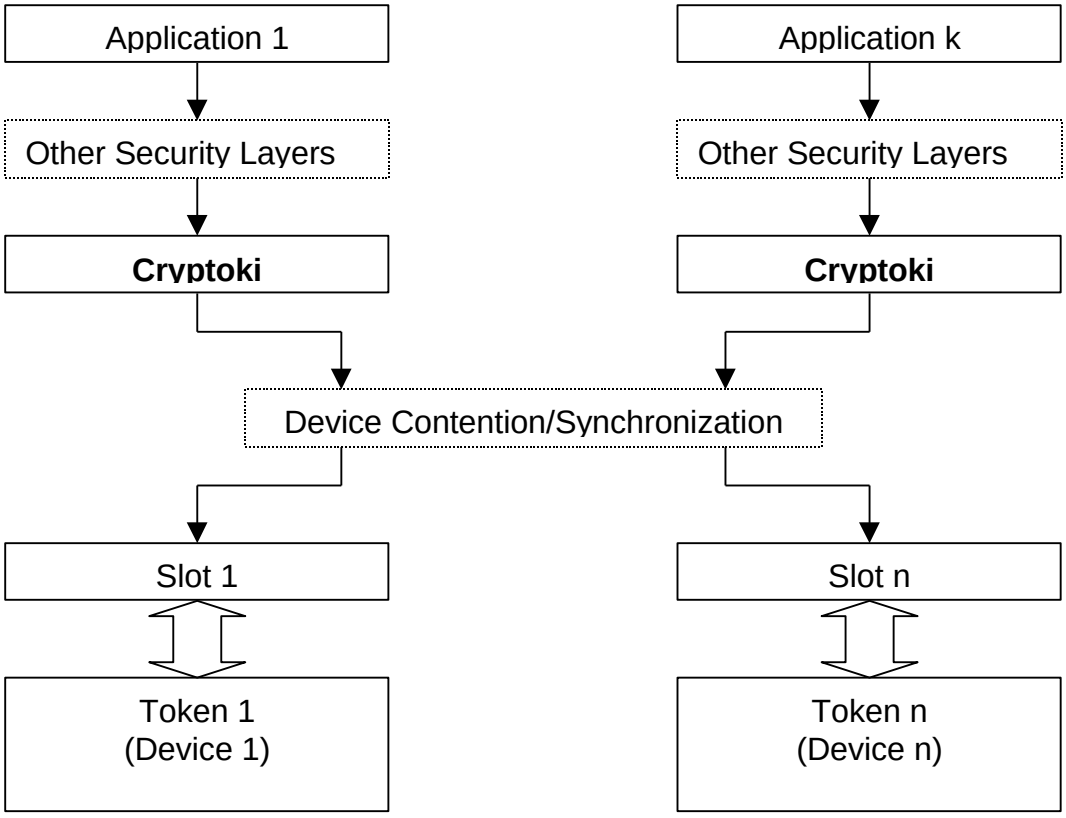

FIGURE 1: GENERAL CRYPTOKI MODEL

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 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.

2.4 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:

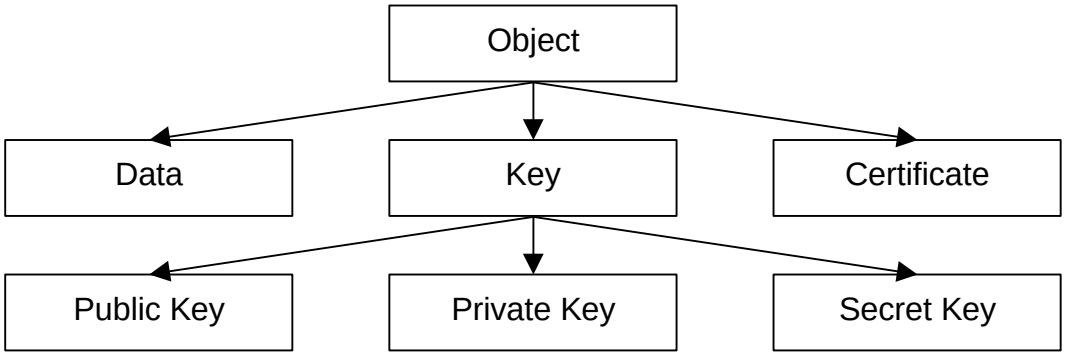

FIGURE 2: OBJECT HIERARCHY

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 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).

See Table on page 15 for further clarification on access to objects.

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.

2.5 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 **Error! Reference source not found.**. 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).

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.

2.6 Applications and their use of Cryptoki

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** (see Section **Error! Reference source not found.**) from one of its threads; after this call is made, the application can call other Cryptoki functions. When the application is done using Cryptoki, it calls the Cryptoki function **C_Finalize** (see Section **Error! Reference source not found.**) and ceases to be a Cryptoki application.

2.6.1 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** 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.

2.6.2 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 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 operation 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 which 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 which 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 which 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 **Error! Reference source not found.** 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.

2.7 Sessions

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.

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 which 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 2.7.1 and Section 2.7.2.

2.7.1 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.

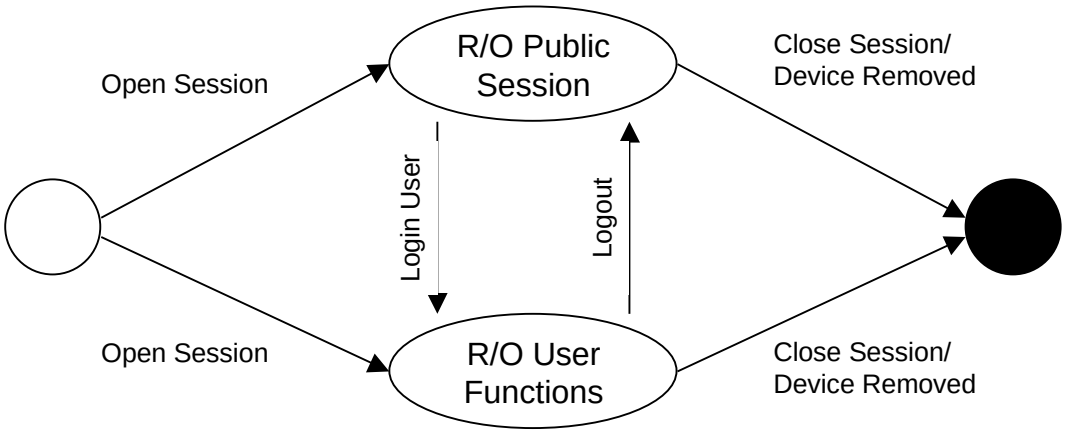

FIGURE 3: READ-ONLY SESSION STATES

The following table describes the session states:

Table 1: Read-Only Session States

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).

2.7.2 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).

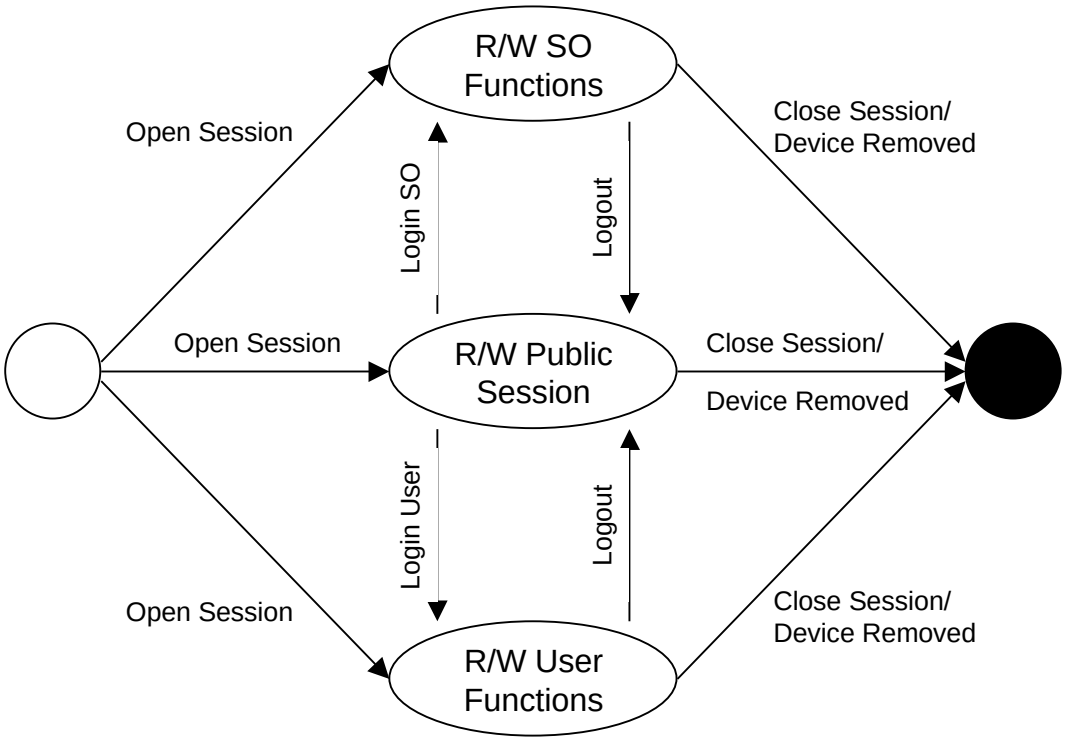

FIGURE 4: READ/WRITE SESSION STATES

The following table describes the session states:

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.

2.7.3 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).

2.7.4 Session events

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

TABLE 3: 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.
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. It also implies that if an application has a R/W SO session with a token, then it may not open a R/O session with that token, since R/O SO sessions do not exist. For the same reason, if an application has a R/O session open, then it may not log any other session into the token as the SO.

2.7.5 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
```

2.7.6 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

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 Section **Error! Reference source not found.**.

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.

2.7.7 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.

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 if session 7 becomes an SO session, then session 4 does, as well, and R/O SO sessions do not exist. **A1** receives an error code indicating that the existence of a R/O session has blocked this attempt to log in (CKR_SESSION_READ_ONLY_EXISTS).
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. **B2** attempts to open a R/O session. The attempt fails, since **B** already has an SO session open, and R/O SO sessions do not exist. **B1** receives an error message indicating that the existence of an SO session has blocked this attempt to open a R/O session (CKR_SESSION_READ_WRITE_SO_EXISTS).
13. **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.
14. **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).
15. **B2** uses [**B**'s] session 7 to perform some operation to modify the object associated with [**B**'s] object handle 7. This modifies **O2**.
16. **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.
17. **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).

18. **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**.
19. **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.
20. **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**.
21. **A2** uses [**A**'s] session 7 to destroy the object associated with [**A**'s] object handle 1. This destroys **O2**.
22. **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).
23. **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.
24. **A1** closes [**A**'s] session 7. This destroys the session object **O1**, which was created by **A**'s session 7.
25. **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.
26. **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).
27. **B2** closes [**B**'s] session 7. At this point, if **B** were to open a new session, the session would not be logged in.
28. **A** and **B** each call **C_Finalize** to indicate that they are done with the Cryptoki library.

3 Security considerations

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 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.

4 Cryptoki tips and reminders

In this section, we clarify, review, and/or emphasize a few odds and ends about how Cryptoki works.

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 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*.

5 OTP Example code

5.1 Disclaimer concerning sample code

For the sake of brevity, sample code presented herein is somewhat incomplete. In particular, initial steps needed to create a session with a cryptographic token are not shown, and the error handling is simplified.

5.2 OTP retrieval

The following sample code snippet illustrates the retrieval of an OTP value from an OTP token using the **C_Sign** function. The sample demonstrates the generality of the approach described herein and does not include any OTP mechanism-specific knowledge.

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_RV rv;
CK_SLOT_ID slotId;
CK_OBJECT_CLASS class = CKO_OTP_KEY;
CK_ATTRIBUTE template[] = {
 {CKA_CLASS, &class, sizeof(class)} };
CK_UTF8CHAR time[] = {...};
/* UTC time value for OTP, or NULL */
CK_UTF8CHAR pin[] = {...};
/* User PIN, or NULL */
CK_BYTE counter[] = {...};
/* Counter value, or NULL */
CK_BYTE challenge[] = {...};
/* Challenge, or NULL */
CK_MECHANISM_TYPE_PTR allowedMechanisms = NULL_PTR;
CK_MECHANISM_INFO mechanismInfo;
CK_MECHANISM mechanism;
CK_ULONG i, ulOTPLen, ulKeyCount, ulChalReq, ulPINReq, ulTimeReq,
          ulCounterReq;
CK_ATTRIBUTE mechanisms[] = { {CKA_ALLOWED_MECHANISMS, NULL_PTR, 0} };
CK_ATTRIBUTE attributes[] = {
 {CKA_OTP_CHALLENGE_REQUIREMENT, &ulChalReq, sizeof(ulChalReq)},
 {CKA_OTP_PIN_REQUIREMENT, &ulPINReq, sizeof(ulPINReq)},
 {CKA_OTP_COUNTER_REQUIREMENT, &ulCounterReq, sizeof(ulCounterReq)},
 {CKA_OTP_TIME_REQUIREMENT, &ulTimeReq, sizeof(ulTimeReq)} };

CK_OTP_PARAM param[4];
CK_OTP_PARAMS params;
```



```
CK_BYTE *pOTP;/* Storage for OTP result */

do {

 /* N.B.: Minimal error and memory handling in this
    sample code. */

 /* Find first OTP key on the token. */
 if ((rv = C_FindObjectsInit(hSession, template, 1)) != CKR_OK) {
  break;
 };
 if ((rv = C_FindObjects(hSession, &hKey, 1, &ulKeyCount)) != CKR_OK) {
  break;
 };
 if (ulKeyCount == 0) {
  /* No OTP key found */
  break;
 }
 rv = C_FindObjectsFinal(hSession);

 /* Find a suitable OTP mechanism. */
 if ((rv = C_GetAttributeValue(hSession, hKey, mechanisms, 1)) != CKR_OK) {
  break;
 };

 if ((allowedMechanisms = (CK_MECHANISM_TYPE_PTR)
              malloc(mechanisms[0].ulValueLen)) == 0) {
  break;
 };

 mechanisms[0].pValue = allowedMechanisms;
 if ((rv = C_GetAttributeValue(hSession, hKey, mechanisms, 1)) != CKR_OK) {
  break;
 };

 for (i = 0; i < mechanisms[0].ulValueLen/ sizeof(CK_MECHANISM_TYPE); ++i) {
  if ((rv = C_GetMechanismInfo(slotId, allowedMechanisms[i], &mechanismInfo))
              == CKR_OK) {
   if (mechanismInfo.flags & CKF_SIGN) {
    break;
   }
  }
 }
```



```
if (i == mechanisms[0].ulValueLen) {
  break;
}

mechanism.mechanism = allowedMechanisms[i];
free(allowedMechanisms);

/* Set required mechanism parameters based on
   the key attributes. */
if ((rv = C_GetAttributeValue(hSession, hKey,
    attributes, sizeof(attributes) /
    sizeof(attributes[0]))) != CKR_OK) {
  break;
}

i = 0;
if (ulPINReq == CK_OTP_PARAM_MANDATORY) {
  /* PIN value needed. */
  param[i].type = CK_OTP_PIN;
  param[i].pValue = pin;
  param[i++].ulValueLen = sizeof(pin) - 1;
}
if (ulChalReq == CK_OTP_PARAM_MANDATORY) {
  /* Challenge neded. */
  param[i].type = CK_OTP_CHALLENGE;
  param[i].pValue = challenge;
  param[i++].ulValueLen = sizeof(challenge);
}
if (ulTimeReq == CK_OTP_PARAM_MANDATORY) {
  /* Time needed (would not normally be
     the case if token has its own clock). */
  param[i].type = CK_OTP_TIME;
  param[i].pValue = time;
  param[i++].ulValueLen = sizeof(time) -1;
}
if (ulCounterReq == CK_OTP_PARAM_MANDATORY) {
  /* Counter value needed (would not normally
     be the case if token has its own counter.*/
  param[i].type = CK_OTP_COUNTER;
  param[i].pValue = counter;
  param[i++].ulValueLen = sizeof(counter);
}

params.pParams = param;
```



```
params.ulCount = i;

mechanism.pParameter = &params;
mechanism.ulParameterLen = sizeof(params);

/* Sign to get the OTP value. */
if ((rv = C_SignInit(hSession, &mechanism, hKey))
  != CKR_OK) {
  break;
}

/* Get the buffer length needed for the OTP Value
   and any associated data. */
if ((rv = C_Sign(hSession, NULL_PTR, 0, NULL_PTR, &ulOTPLen)) != CKR_OK) {
  break;
};

if ((pOTP = malloc(ulOTPLen)) == NULL_PTR) {
  break;
};

/* Get the actual OTP value and any
   associated data. */
if ((rv = C_Sign(hSession, NULL_PTR, 0, pOTP,
        &ulOTPLen)) != CKR_OK) {
  break;
}

/* Traverse the returned pOTP here. The actual
   OTP value is in CK_OTP_VALUE in pOTP. */

} while (0);
```

5.3 User-friendly mode OTP token

This sample demonstrates an application retrieving a user-friendly OTP value. The code is the same as in 5.1 except for the following:

```
/* Add these variable declarations */

CK_FLAGS flags = CKF_USER_FRIENDLY_OTP;
CK_BBOOL bUserFriendlyMode;
CK_ULONG ulFormat;
```



```
/* Replace the declaration of the "attributes" and the
   "param" variables with: */

  CK_ATTRIBUTE attributes[] = {
   {CKA_OTP_CHALLENGE_REQUIREMENT, &ulChalReq,
   sizeof(ulChalReq)},
   {CKA_OTP_PIN_REQUIREMENT, &ulPINReq,
   sizeof(ulPINReq)},
   {CKA_OTP_COUNTER_REQUIREMENT, &ulCounterReq,
   sizeof(ulCounterReq)},
   {CKA_OTP_TIME_REQUIREMENT, &ulTimeReq,
   sizeof(ulTimeReq)},
   {CKA_OTP_USER_FRIENDLY_MODE, &bUserFriendlyMode,
   sizeof(bUserFriendlyMode)},
   {CKA_OTP_FORMAT, &ulFormat,
   sizeof(ulFormat)}
  };

    CK_OTP_PARAM param[5];

/* Replace the assignment of the "pParam" component
   of the "params" variable with: */

    if (bUserFriendlyMode == CK_TRUE) {
      /* Token supports user-friendly OTPs */
      param[i].type = CK_OTP_FLAGS;
      param[i].pValue = &flags;
      param[i++].ulValueLen = sizeof(CK_FLAGS);
    } else if (ulFormat == CK_OTP_FORMAT_BINARY) {
      /* Some kind of error since a user-friendly
         OTP cannot be returned to an application
         that needs it. */
      break;
    };

    params.pParams = param;

/* Further processing is as in B.1. */
```


5.4 OTP verification

The following sample code snippet illustrates the verification of an OTP value from an RSA SecurID token, using the **C_Verify** function. The desired UTC time, if a time is specified, is supplied in the CK_OTP_PARAMS structure, as is the user's PIN.

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_UTF8CHAR time[] = {...};
/* UTC time value for OTP, or NULL */
CK_UTF8CHAR pin[] = {...};
/* User PIN or NULL (if collected by library) */
CK_OTP_PARAM param[] = {
 {CK_OTP_TIME, time, sizeof(time)-1},
 {CK_OTP_PIN, pin, sizeof(pin)-1}
};
CK_OTP_PARAMS params = {param, 2};
CK_MECHANISM mechanism = {CKM_SECURID, &params, sizeof(params)};
CK_ULONG ulKeyCount;
CK_RV rv;
CK_BYTE OTP[] = {...};      /* Supplied OTP value. */
CK_ULONG ulOTPLen = strlen((CK_CHAR_PTR)OTP);
CK_OBJECT_CLASS class = CKO_OTP_KEY;
CK_KEY_TYPE keyType = CKK_SECURID;

CK_ATTRIBUTE template[] = {
 {CKA_CLASS, &class, sizeof(class)},
 {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
};

/* Find the RSA SecurID key on the token. */
rv = C_FindObjectsInit(hSession, template, 2);
if (rv == CKR_OK) {
      rv = C_FindObjects(hSession, &hKey, 1, &ulKeyCount);
      rv = C_FindObjectsFinal(hSession);
}

if ((rv != CKR_OK) || (ulKeyCount == 0)) {
      printf(" \nError: unable to find RSA SecurID key on token.\n");
      return(rv);
}

rv = C_VerifyInit(hSession, &mechanism, hKey);
if (rv == CKR_OK) {
 ulOTPLen = sizeof(OTP);
```



```
    rv = C_Verify(hSession, NULL_PTR, 0, OTP, ulOTPLen);
  }

switch(rv) {
    case CKR_OK:
        printf("\nSupplied OTP value verified.\n");
        break;

    case CKR_SIGNATURE_INVALID:
        printf("\nSupplied OTP value not verified.\n");
        break;

    default:
        printf("\nError:Unable to verify OTP value.\n");
        break;
  }

  return(rv);
```


6 Using PKCS #11 with CT-KIP

A suggested procedure to perform CT-KIP with a cryptographic token through the PKCS #11 interface using the mechanisms defined herein is as follows:

a. On the client side,

   I. The client selects a suitable slot and token (e.g. through use of the **<TokenID>** or the **<PlatformInfo>** element of the CT-KIP trigger message).

   II. Optionally, a nonce *R* is generated, e.g. by calling **C_SeedRandom** and **C_GenerateRandom**.

   III. The client sends its first message to the server, potentially including the nonce *R*.

b. On the server side,

   I. A nonce R_S is generated, e.g. by calling **C_SeedRandom** and **C_GenerateRandom**.

   II. If the server needs to authenticate its first CT-KIP message, and use of **CKM_KIP_MAC** has been negotiated, it calls **C_SignInit** with **CKM_KIP_MAC** as the mechanism followed by a call to **C_Sign**. In the call to **C_SignInit**, K_{AUTH} (see **Error! Reference source not found.**) shall be the signature key, the *hKey* parameter in the **CK_KIP_PARAMS** structure shall be set to NULL_PTR, the *pSeed* parameter of the **CT_KIP_PARAMS** structure shall also be set to NULL_PTR and the *ulSeedLen* parameter shall be set to zero. In the call to **C_Sign**, the *pData* parameter shall be set to point to (the concatenation of the nonce *R*, if received, and) the nonce R_S (see **Error! Reference source not found.** for a definition of the variables), and the *ulDataLen* parameter shall hold the length of the (concatenated) string. The desired length of the MAC shall be specified through the *pulSignatureLen* parameter as usual.

   III. The server sends its first message to the client, including R_S, the server's public key *K* (or an identifier for a shared secret key *K*), and optionally the MAC.

c. On the client side,

   I. If a MAC was received, it is verified. If the MAC does not verify, or was required but not received, the protocol session ends with a failure.

   II. If the MAC verified, or was not required and not present, a generic secret key, R_C, is generated by calling **C_GenerateKey** with the **CKM_GENERIC_SECRET_KEY_GEN** mechanism. The *pTemplate* attribute shall have **CKA_EXTRACTABLE** and **CKA_SENSITIVE** set to **CK_TRUE**, and should have **CKA_ALLOWED_MECHANISMS** set to **CKM_KIP_DERIVE** only.

   III. The generic secret key R_C is wrapped by calling **C_WrapKey**. If the server's public key is used to wrap R_C, and that key is temporary only, then the **CKA_EXTRACTABLE** attribute of R_C shall be set to CK_FALSE once R_C has been wrapped and the server's public key is to be destroyed. If a shared secret key is used to wrap R_C, and use of the CT-KIP key wrapping algorithm was negotiated, then the **CKM_KIP_WRAP** mechanism shall be used. The *hKey* handle in the

CK_KIP_PARAMS structure shall be set to NULL_PTR. The *pSeed* parameter in the **CK_KIP_PARAMS** structure shall point to the nonce R_S provided by the CT-KIP server, and the *ulSeedLen* parameter shall indicate the length of R_S. The *hWrappingKey* parameter in the call to **C_WrapKey** shall be set to refer to the wrapping key.

IV. The client sends its second message to the server, including the wrapped generic secret key R_C.

d. On the server side,

I. Once the wrapped generic secret key R_C has been received, the server calls **C_UnwrapKey**. If use of the CT-KIP key wrapping algorithm was negotiated, then **CKM_KIP_WRAP** shall be used to unwrap R_C. When calling **C_UnwrapKey**, the **CK_KIP_PARAMS** structure shall be set as described in c.III above. The *hUnwrappingKey* function parameter shall refer to the shared secret key and the *pTemplate* function parameter shall have **CKA_SENSITIVE** set to **CK_TRUE**, **CKA_KEY_TYPE** set to **CKK_GENERIC_SECRET** and should have **CKA_ALLOWED_MECHANISMS** set to **CKM_KIP_DERIVE** only. This will return a handle to the generic secret key R_C.

II. A token key, K_{TOKEN}, is derived from R_C by calling **C_DeriveKey** with the **CKM_KIP_DERIVE** mechanism, using R_C as *hBaseKey*. The *hKey* handle in the **CK_KIP_PARAMS** structure shall refer either to the public key supplied by the CT-KIP server, or alternatively, the shared secret key indicated by the server. The *pSeed* parameter shall point to the nonce R_S provided by the CT-KIP server, and the *ulSeedLen* parameter shall indicate the length of R_S. The *pTemplate* attribute shall be set in accordance with local policy and as negotiated in the protocol. This will return a handle to the token key, K_{TOKEN}.

III. For the server's last CT-KIP message to the client, if use of the CT-KIP MAC algorithm has been negotiated, then the MAC is calculated by calling **C_SignInit** with the **CKM_KIP_MAC** mechanism followed by a call to **C_Sign**. In the call to **C_SignInit**, K_{AUTH} (see **Error! Reference source not found.**) shall be the signature key, the *hKey* parameter in the **CK_KIP_PARAMS** structure shall be a handle to the generic secret key R_C, the *pSeed* parameter of the **CT_KIP_PARAMS** structure shall be set to NULL_PTR, and the *ulSeedLen* parameter shall be set to zero. In the call to **C_Sign**, the *pData* parameter shall be set to NULL_PTR and the *ulDataLen* parameter shall be set to 0. The desired length of the MAC shall be specified through the *pulSignatureLen* parameter as usual.

IV. The server sends its second message to the client, including the MAC.

e. On the client side,

I. The MAC is verified in a reciprocal fashion as it was generated by the server. If use of the **CKM_KIP_MAC** mechanism was negotiated, then in the call to **C_VerifyInit,** the *hKey* parameter in the **CK_KIP_PARAMS** structure shall refer to R_C, the *pSeed* parameter shall be set to NULL_PTR, and *ulSeedLen* shall be set to 0. The *hKey* parameter of **C_VerifyInit** shall refer to K_{AUTH}. In the call to **C_Verify**, *pData* shall be set to NULL_PTR, *ulDataLen* to 0, *pSignature* to the MAC value received from

the server, and *ulSignatureLen* to the length of the MAC. If the MAC does not verify the protocol session ends with a failure.

II. A token key, K_{TOKEN}, is derived from R_C by calling **C_DeriveKey** with the **CKM_KIP_DERIVE** mechanism, using R_C as *hBaseKey*. The *hKey* handle in the **CK_KIP_PARAMS** structure shall be set to NULL_PTR as token policy must dictate use of the same key as was used to wrap R_C. The *pSeed* parameter shall point to the nonce R_S provided by the CT-KIP server, and the *ulSeedLen* parameter shall indicate the length of R_S. The *pTemplate* attribute shall be set in accordance with local policy and as negotiated and expressed in the protocol. In particular, the value of the **<KeyID>** element in the server's response message may be used as **CKA_ID.** The call to **C_DeriveKey** will, if successful, return a handle to K_{TOKEN}. [1]

[1] When K_{AUTH} is the newly generated K_{TOKEN}, the client will need to call **C_DeriveKey** before calling **C_VerifyInit** and **C_Verify** (since the *hKey* parameter of **C_VerifyInit** shall refer to K_{TOKEN}). In this case, the token should not allow K_{TOKEN} to be used for any other operation than the verification of the MAC value until the MAC has successfully been verified.

7 Deprecated PKCS #11 Functionality

7.1 Secondary authentication (Deprecated)

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

7.2 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.

8 Deferred Work

Text.

9 Implementation Conformance

This document is intended to be informational only and as such has no conformance clauses. The conformance requirements for the PKCS11 Specification can be found in the "PKCS11 Specification" document itself, at the URL noted in the “Normative References” section of this document.

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.
Peter Bartok, Venafi, Inc.
Anthony Berglas, Cryptsoft Pty Ltd.
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.
Tony Cox, Cryptsoft Pty Ltd.
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 Pty Ltd.
Gershon Janssen, Individual
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, Oracle
Mark Lambiase, SecureAuth Corporation
Lawrence Lee, GoTrust Technology Inc.
John Leiseboer, QuintessenceLabs Pty Ltd.
Hal Lockhart, Oracle
Robert Lockhart, Thales e-Security
Dale Moberg, Axway Software
Darren Moffat, Oracle
Valery Osheter, SafeNet, Inc.
Sean Parkinson, EMC
Rob Philpott, EMC

Mark Powers, Oracle
Ajai Puri, SafeNet, Inc.
Robert Relyea, Red Hat
Saikat Saha, SafeNet, Inc.
Subhash Sankuratripati, NetApp
Johann Schoetz, Infineon Technologies AG
Rayees Shamsuddin, Wave Systems Corp.
Radhika Siravara, Oracle
Ryan Smith, Futurex
David Smith, Venafi, Inc.
Brian Smith, Mozilla Corporation
Jerry Smith, US Department of Defense (DoD)
Oscar So, Oracle
Michael Stevens, QuintessenceLabs Pty Ltd.
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
Wang Xuelin, Beijing WatchSmart Technologies Co., Ltd
Magda Zdunkiewicz, Cryptsoft Pty Ltd.
Chris Zimman, Bloomberg Finance

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