nShield Security World

PKCS 11 Reference Guide for nShield Security World v13.5.1

16 February 2024

Table of Contents

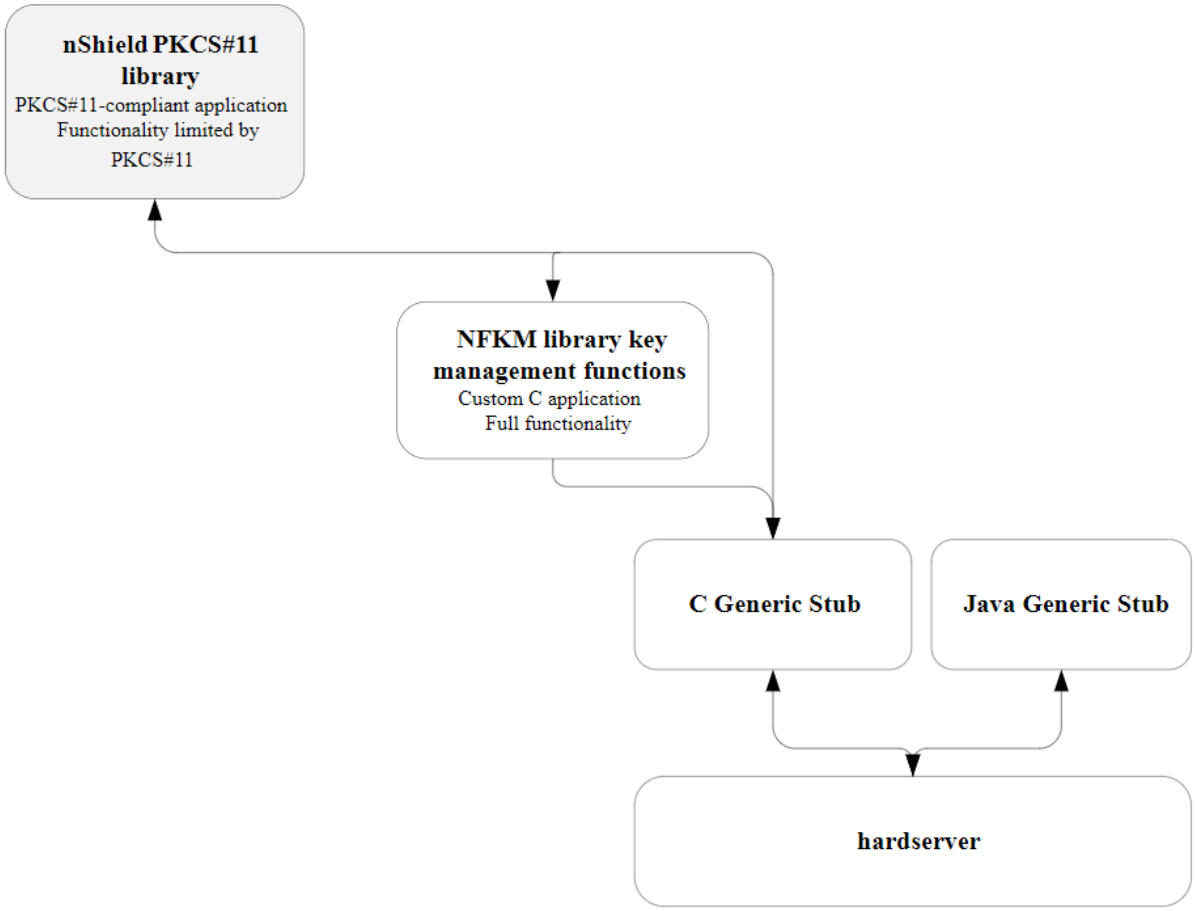
1. Introduction

This guide is for application developers who are writing PKCS #11 applications.

For an introduction to the PKCS #11 user library, including information about the environment variables and utilities available, see the *User Guide* for your HSM.

Before using the nShield PKCS #11 libraries, we recommend that you read http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/pkcs11-base-v2.40.html.

The following diagram illustrates the way that an nShield PKCS #11 library works with the nShield APIs.



This guide does not address how the nShield PKCS #11 libraries map PKCS #11 functions to nCore API calls within the library.

This guide describes the nShield PKCS #11 library supplied by Entrust Security to help developers write applications that use nShield modules.

This toolkit, like the application plug-ins supplied by Entrust, uses the Security World paradigm for key storage. For an introduction to Security Worlds, see the *User Guide*.

1.1. Read this guide if...

Read this guide if you want to build an application that uses an nShield key-management module to accelerate cryptographic operations and protect cryptographic keys through a standard interface rather than the full nCore API.

This guide assumes that you are familiar with the concept of the Security World, described in the User Guide. It is intended for experienced programmers and assumes that you are familiar with the following documentation:

- The *nCore Developer Tutorial*, which describes how to write applications using an nShield module.
- The *nCore API Documentation* (supplied as HTML), which describes the nCore API.

1.2. Model numbers

Model numbering conventions are used to distinguish different nShield hardware security devices. In the following table, *n* represents any single-digit integer.

Model number	Used for
NH2047	nShield Connect 6000
NH2040	nShield Connect 1500
NH2033	nShield Connect 500
NH2068	nShield Connect 6000+
NH2061	nShield Connect 1500+
NH2054	nShield Connect 500+
NH2075-B	nShield Connect XC Base
NH2075-M	nShield Connect XC Medium
NH2075-H	nShield Connect XC High
NH2075-B	nShield 5c Base
NH2075-M	nShield 5c Medium
NH2075-H	nShield 5c High
NH2082	nShield Connect XC SCAP
NH2089-B	nShield Connect XC Base - Serial Console
NH2089-M	nShield Connect XC Mid - Serial Console

Model number	Used for
NH2089-H	nShield Connect XC High - Serial Console
NH3003-B	nShield Connect CLX Base - Serial Console
NH3003-M	nShield Connect CLX Mid - Serial Console
NH3003-H	nShield Connect CLX High - Serial Console
nC2021E-000, NCE2023E-000	nToken PCIe
nC3*nnn*E-*nnn*, nC4*nnn*E-*nnn*	nShield Solo PCIe
nC30n5E-*nnn*, nC40n5E-*nnn*	nShield Solo XC PCIe
nC30*nn*U-10, nC40*nn*U-10	nShield Edge
NC5536E-B	nShield 5s Base
NC5536E-M	nShield 5s Medium
NC5536E-H	nShield 5s High

1.3. Security World Software default directories

The default locations for Security World Software and program data directories on English-language systems are summarized in the following table:

Directory Name	Environment Variable	Windows Server 2016	Linux
nShield Installation	`NFAST_HOME`	`C:\Program Files\nCipher\nfast`	`/opt/nfast/`
Key Management Data	`NFAST_KMDATA`	`C:\ProgramData\nCipher\Key Management Data`	`/opt/nfast/kmdata/`
Dynamic Feature Certificates	`NFAST_CERTDIR`	`C:\ProgramData\nCipher\Feature Certificates`	`/opt/nfast/femcerts/`
Static Feature Certificates		`C:\ProgramData\nCipher\Features`	`/opt/nfast/kmdata/features/`
Log Files	`NFAST_LOGDIR`	`C:\ProgramData\nCipher\Log Files`	`/opt/nfast/log/`

By default, the Windows `%NFAST_KMDATA%` directories are hidden directories. To see these directories and their contents, you must enable the display of hidden files and directories in the **View** settings of the **Folder Options**.

Dynamic feature certificates must be stored in the directory stated in

> the default directories table.
>
> The directory shown for static feature certificates is an example location. You can store those certificates in any directory and provide the appropriate path when using the Feature Enable Tool. However, you must not store static feature certificates in the dynamic features certificates directory. For more information about feature certificates, see the *User Guide* for your HSM.

The absolute paths to the Security World Software installation directory and program data directories on Windows platforms are stored in the indicated nShield environment variables at the time of installation. If you are unsure of the location of any of these directories, check the path set in the environment variable.

The instructions in this guide refer to the locations of the software installation and program data directories by their names (for example, Key Management Data) or:

- For Windows, nShield environment variable names enclosed in percent signs (for example, `%NFAST_KMDATA%`).
- For Linux, absolute paths (for example, `/opt/nfast/kmdata/`).

`NFAST_KMDATA` cannot be a symbolic link.

If the software has been installed into a non-default location:

- For Windows, ensure that the associated nShield environment variables are re-set with the correct paths for your installation.
- For Linux, you must create a symbolic link from `/opt/nfast/` to the directory where the software is actually installed. For more information about creating symbolic links, see your operating system's documentation.

1.4. Utility help options

Unless noted, all the executable utilities provided in the `bin` subdirectory of your nShield installation have the following standard help options:

`-h|--help` displays help for the utility

`-v|--version` displays the version number of the utility

`-u|--usage` displays a brief usage summary for the utility.

1.5. Further information

This guide forms one part of the information and support provided by Entrust.

The *nCore API Documentation* is supplied as HTML files installed in the following locations:

- Windows:
  - API reference for host: `%NFAST_HOME%\document\ncore\html\index.html`
  - API reference for SEE: `%NFAST_HOME%\document\csddoc\html\index.html`
- Linux:
  - API reference for host: `/opt/nfast/document/ncore/html/index.html`
  - API reference for SEE: `/opt/nfast/document/csddoc/html/index.html`

The Java Generic Stub classes, nCipherKM JCA/JCE provider classes, and Java Key Management classes are supplied with HTML documentation in standard Javadoc format, which is installed in the appropriate `nfast\java` or `nfast/java` directory when you install these classes.

1.6. Security advisories

If Entrust becomes aware of a security issue affecting nShield HSMs, Entrust will publish a security advisory to customers. The security advisory will describe the issue and provide recommended actions. In some circumstances the advisory may recommend you upgrade the nShield firmware and or image file. In this situation you will need to re-present a quorum of administrator smart cards to the HSM to reload a Security World. Because of this, you should consider the procedures and actions required to upgrade devices in the field when deploying and maintaining your HSMs.

The Remote Administration feature supports remote firmware upgrade of nShield HSMs, and remote ACS card presentation.

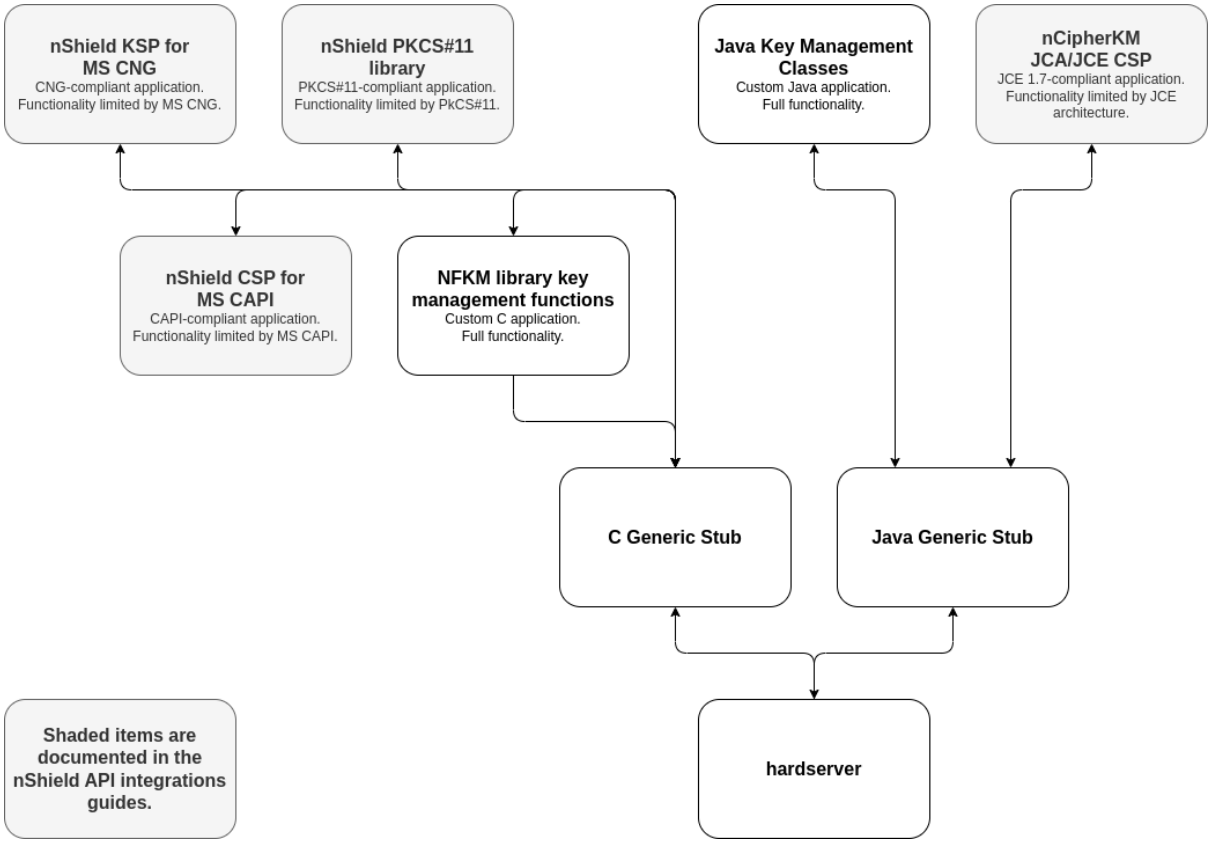
We recommend that you monitor the Announcements & Security Notices section on Entrust nShield, https://nshieldsupport.entrust.com, where any announcement of nShield Security Advisories will be made.

1.7. Contacting Entrust nShield Support

To obtain support for your product, contact Entrust nShield Support, https://nshieldsupport.entrust.com.

2. nShield Architecture

This chapter provides a brief overview of the Security World Software architecture. The following diagram provides a visual representation of nShield architecture and the documentation that relates to it.



2.1. Security World Software modules

nShield modules provide a secure environment to perform cryptographic functions. Key-management modules are fitted with a smart card interface that enables keys to be stored on removable tokens for extra security. nShield modules are available for PCI buses and also as network-attached Ethernet modules (nShield Connect).

2.2. Security World Software server

The Security World Software server, often referred to as the `hardserver`, accepts requests by means of an interprocess communication facility (for example, a domain socket on Linux or named pipes or TCP/IP sockets on Windows).

The Security World Software server receives requests from applications and passes these

to the nShield module(s). The module handles these requests and returns them to the server. The server ensures that the results are returned to the correct calling program.

You only need a single Security World Software server running on your host computer. This server can communicate with multiple applications and multiple nShield modules.

2.3. Stubs and interface libraries

An application can either handle its own cryptographic functions or it can use a cryptographic library:

- If the application uses a cryptographic library that is already able to communicate with the Security World Software server, then no further modification is necessary. The application can automatically make use of the nShield module.
- If the application uses a cryptographic library that has not been modified to be able to communicate with the Security World Software server, then either Entrust or the cryptographic library supplier need to create adaption function(s) and compile them into the cryptographic library. The application users then must relink their applications using the updated cryptographic library.

If the application performs its own cryptographic functions, you must create adaption function(s) that pass the cryptographic functions to the Security World Software server. You must identify each cryptographic function within the application and change it to call the nShield adaption function, which in turn calls the generic stub. If the cryptographic functions are provided by means of a DLL or shared library, the library file can be changed. Otherwise, the application itself must be recompiled.

2.4. Using an interface library

Entrust supplies the following interface libraries:

- Microsoft Cryptography API: Next Generation (CNG)
- Microsoft CryptoAPI (CAPI)
- PKCS #11
- nCipherKM JCA/JCE CSP

Third-party vendors may supply nShield-aware versions of their cryptographic libraries.

The functionality provided by these libraries is the intersection of the functionality provided by the nCore API and the functionality provided by the standard for that library.

Most standard libraries offer fewer key-management options than are available in the nCore API. However, the nShield libraries do not include any extensions to their standards. If you want to make use of features of the nCore API that are not offered in the standard, you should convert your application to work directly with the generic stub.

On the other hand, many standard libraries include functions that are not supported on the nShield module, such as support for IDEA or Skipjack. If you require a feature that is not supported on the nShield module, contact Support because it may be possible to add the feature in a future release. However, in many cases, features are not present on the module for licensing reasons, as opposed to technical reasons, and Entrust cannot offer them in the interface library.

2.5. Writing a custom application

If you choose not to use one of the interface libraries, you must write a custom application. This gives you access to all the features of the nCore API. For this purpose, Entrust provides generic stub libraries for C and Java. If you want to use a language other than C orJava, you must write your own wrapper functions in your chosen programming language that call the C generic stub functions.

Entrust supplies several utility functions to help you write your application.

2.6. Acceleration-only or key management

You must also decide whether you want to use key management or whether you are writing an acceleration-only application.

Acceleration-only applications are much simpler to write but do not offer any security benefits.

The Microsoft CryptoAPI, Java JCE, PKCS #11, as well as the application plug-ins, use the Security World paradigm for key storage.

If you are writing a custom application, you have the option of using the Security World mechanisms, in which case your users can use either KeySafe or the command-line utilities supplied with the module for many key-management operations. This means you do not have to write these functions yourself.

The NFKM library gives you access to all the Security World functionality.

3. PKCS #11 Developer libraries

The nShield PKCS #11 libraries, `libcknfast.so` and `libcknfast.a` (nShield tools only) on Linux, and `cknfast.lib` and `cknfast.dll` on Windows are provided so that you can integrate your PKCS #11 applications with the nShield hardware security modules.

The nShield PKCS #11 libraries:

- Provide the PKCS #11 mechanisms listed in Mechanisms
- Help you to identify potential security weaknesses, enabling you to create secure PKCS #11 applications more easily.

3.1. PKCS #11 security assurance mechanism

It is possible for an application to use the PKCS #11 API in ways that can introduce potential security weaknesses. For example, it is a requirement of the PKCS #11 standard that the nShield PKCS #11 libraries are able to generate keys that are explicitly exportable in plain text. An application could use this ability in error when a secure key would be more appropriate.

The nShield PKCS #11 libraries are provided with a configurable security assurance mechanism (SAM). SAM helps prevent PKCS #11 applications from performing operations through the PKCS #11 API that may compromise the security of cryptographic keys. Operations that reveal questionable behavior by the application fail by default with an explanation of the cause of failure.

If you decide that some operations that carry a higher security risk are acceptable to you, then you can reconfigure the nShield PKCS #11 library to permit these operations by means of the environment variable `CKNFAST_OVERRIDE_SECURITY_ASSURANCES`. You must think carefully, however, before permitting operations that could compromise the security of cryptographic keys. For more information about the environment variable and its parameters, see the *User Guide* for your HSM.

It is your responsibility as a security developer to familiarize yourself with the PKCS #11 standard and to ensure that all cryptographic operations used by your application are implemented in a secure manner.

If no parameters are supplied to the environment variable, the nShield PKCS #11 library fails and issues a warning, with an explanation, when the following operations are detected:

- Short term session keys created as long term objects

- Keys that can be exported as plain text are created
- Keys are imported from external sources
- Wrapping keys are created or imported
- Unwrapping keys are created or imported
- Keys with weak algorithms (for example, DES) are created
- Keys with short key length are created.

4. PKCS #11 with load sharing mode

The behavior of the nShield PKCS #11 library varies depending on which of load-sharing mode, HSM Pool mode or neither or these is enabled. If you have enabled load-sharing mode, the nShield PKCS #11 library creates one virtual slot for each OCS and, optionally, also creates one slot for the HSM or HSMs. Softcards appear as additional virtual slots once enabled.

Load-sharing mode must be enabled in PKCS #11 in order to use softcards.

Whether or not load-sharing mode is enabled is determined by the state of the `CKNFAST_LOADSHARING` environment variable.

Load-sharing mode enables you to load a single PKCS #11 token onto several nShield HSMs to improve performance. To enable successful load-sharing with an OCS protected key:

- You must have an Operator Card from the OCS inserted into every slot from the same 1/*N* card set
- All the Operator Cards must have the same passphrase.

The nShield-specific API calls, `C_LoginBegin`, `C_LoginNext`, and `C_LoginEnd` do not function in load-sharing mode. *K*/*N* support for card sets in load-sharing mode is only available if you first use `preload` to load the logical token.

4.1. Logging in

If you call `C_Login` without a token present, it fails (as expected) unless you are using a persistent token with `preload` or using only module-protected keys. Therefore, your application should prompt users to insert tokens before logging in.

The nShield PKCS #11 library removes the nShield logical token when you call `C_Logout`, whether or not there is a smart card in the reader.

If there are any cards from the OCS present when you call `C_Logout`, the PKCS #11 token remains present but not logged-in until all cards in the set are removed. If there are no cards present, the PKCS #11 token becomes not present.

The `CKNFAST_NONREMOVABLE` environment variable is only available for persistent tokens. When the variable is set, the rules for recognizing new cards are overridden, and the only way to invoke a new token is to call `C_Finalize` or `C_Initialize`.

4.2. Session objects

Session objects are loaded on all modules.

4.3. Module failure

If a subset of the modules fails, the nShield PKCS #11 library handles commands using the remaining modules. If a module fails, the single cryptographic function that was running on that module will fail, and the nShield PKCS #11 library will return a PKCS #11 error. Subsequent cryptographic commands will be run on other modules.

4.4. Compatibility

Before the implementation of load-sharing, the nShield PKCS #11 library puts the electronic serial number in both the `slotinfo.slotDescription` and `tokeninfo.serialNumber` fields. If you have enabled load-sharing, the `tokeninfo.serialNumber` field displays the hash of the OCS.

4.5. Restrictions on function calls in load-sharing mode

The following function calls are not supported in load-sharing mode:

- `C_LoginBegin` (nShield-specific call to support *K*/*N* card sets)
- `C_LoginNext` (nShield-specific call to support *K*/*N* card sets)
- `C_LoginEnd` (nShield-specific call to support *K*/*N* card sets).

The following function calls are supported in load-sharing mode *only* when using softcards:

- `C_InitToken`
- `C_InitPIN`
- `C_SetPIN`.

To use `C_InitToken`, `C_InitPIN`, or `C_SetPIN` in load-sharing mode, you must have created a softcard with the command `ppmk -n` before selecting the corresponding slot.

The `C_InitToken` function is *not* supported for use in non-load-sharing FIPS 140 Level 3 Security Worlds.

5. PKCS #11 with HSM Pool mode

If HSM Pool mode is enabled, the nShield PKCS #11 library exposes a single pool of HSMs and a single virtual slot for a fixed token with the label `accelerator`. This accelerator slot can be used to create module protected keys and to support session objects.

HSM Pool mode supports module protected keys but does not support token-protected keys. If your application only uses module protected keys, you can use HSM Pool mode as an alternative to using load-sharing mode. HSM Pool mode supports returning or adding a hardware security module to the pool without restarting the system.

Whether or not HSM Pool mode is enabled is determined by the state of the `CKNFAST_HSM_POOL` environment variable.

In FIPS 140 Level 3 Security Worlds, keys cannot be created in HSM Pool mode, however keys created outside HSM Pool mode can be used in HSM Pool mode.

5.1. Module failure

If a subset of the modules in the HSM pool fail, the nShield PKCS #11 library handles commands using the remaining modules. When a module fails, any cryptographic functions that were running on that module are restarted on one of the remaining modules. If all of the modules in the HSM pool fail, the nShield PKCS #11 library will return a PKCS #11 error.

5.2. Module recovery

If a failed module recovers and remains part of the Security World, it is automatically returned to the HSM Pool and the nShield PKCS #11 library can use it for new commands. If a new module is added to the system that is accessible to the host running the PKCS #11 application, then once the Security World has been loaded onto this HSM, then it is automatically added to the HSM Pool and the nShield PKCS #11 library can use it for new commands.

5.3. Restrictions on function calls in HSM Pool mode

The following function calls are not supported in HSM Pool mode:

- `C_LoginBegin`
- `C_LoginNext`

- C_LoginEnd
- C_InitToken
- C_InitPIN
- C_SetPIN

6. Generating and deleting NVRAM-stored keys with PKCS #11

You can use the nShield PKCS #11 library to generate keys stored in nonvolatile memory (up to a maximum of 12 keys) if you have set the `CKNFAST_NVRAM_KEY_STORAGE` environment variable.

6.1. Generating NVRAM-stored keys

To generate NVRAM-stored keys with the nShield PKCS #11 library:

1. Load (or reload) the ACS using the `preload` command-line utility. Open a command-line window and give the command:

   ```
   preload --admin=NV pause
   ```

2. After loading the ACS, remove the Administrator Cards from the module.
3. Ensure that the `CKNFAST_NVRAM_KEY_STORAGE` environment variable is set. If this variable is not set, the keys generated are not stored in NVRAM.
4. Open a second command-line window, and give the command:

   ```
   preload --cardset-name=<name> <pkcs11app>
   ```

   where `<name>` is the cardset name and `<pkcs11app>` is the name of your PKCS #11 application.

5. Generate the NVRAM-stored keys that you need (up to a maximum of 12 keys) as normal.
6. Stop or close `<pkcs11app>`.
7. Return to the command-line window you opened in step 1 and terminate the `preload --admin=NV pause` process.

   Do not allow the `preload --admin=NV pause` process to run continuously. Run this process only when generating or deleting NVRAM-stored keys. As usual, remove the Administrator Cards when they are not in use and store them safely.

8. Unset the `CKNFAST_NVRAM_KEY_STORAGE` environment variable.
9. Restart `<pkcs11app>`.

You can use the newly generated NVRAM-stored keys in the same way as other PKCS #11 keys. You can also generate any number of standard keys (not stored in NVRAM) in the usual way.

6.2. Deleting NVRAM-stored keys

To delete NVRAM-stored keys with the nShield PKCS #11 library:

1. Load (or reload) the ACS using the `preload` command-line utility. Open a command-line window and give the command:

   ```
   preload --admin=NV pause
   ```

2. After loading the ACS, remove the Administrator Cards from the module. Ensure that the `CKNFAST_NVRAM_KEY_STORAGE` environment variable is set.

   If you attempt to delete NVRAM-stored keys without the `CKNFAST_NVRAM_KEY_STORAGE` environment variable set, only the key blob stored on hard disk is deleted. The keys remain in NVRAM on the module. Use the `nvram-sw` command-line utility to fully remove the NVRAM-stored keys. For more information, see the *User Guide*.

3. Open a second command-line window, and give the command:

   ```
   preload --cardset-name=<name> -M <pkcs11app>
   ```

   where `<name>` is the cardset name and `<pkcs11app>` is the name of the PKCS #11 application that you use to delete the keys.

4. Delete the NVRAM-stored keys as you would delete normal keys.
5. Stop or close `<pkcs11app>`.
6. Return to the command-line window you opened in step 1 and terminate the `preload --admin=NV pause` process.

   Do not allow the `preload --admin=NV pause` to run continuously. Run this process only when generating or deleting NVRAM-stored keys. As usual, remove the Administrator Cards when they are not in use and store them safely.

7. Unset the `CKNFAST_NVRAM_KEY_STORAGE` environment variable.

7. PKCS #11 with key reloading

The nShield PKCS #11 library is capable of reloading keys to nShield HSMs after a PKCS #11 application has started. The PKCS #11 library will attempt to reload the keys to all HSMs from which keys have been unloaded after the application was started, for example, if the HSM was cleared. This also means that if an application uses HSMs that became unusable during runtime, the PKCS #11 library will re-add these HSMs into the group of HSMs in a single Security World when they become usable again. The PKCS #11 library will also attempt to reload the keys on new HSMs that become usable after the application has started, for example if you enroll a new HSM into the Security World. The application can then use the HSM for key operations.

The default behavior without PKCS #11 key reloading is that when an HSM is removed from the group of HSMs in a Security World, it is not re-added for PKCS #11 until the user's application is restarted.

The `CKNFAST_RELOAD_KEYS` environment variable determines whether key reloading mode is enabled.

Load-sharing mode must be enabled in PKCS #11 to use key reloading mode. If load-sharing is not enabled, it is enabled automatically if `CKNFAST_RELOAD_KEYS` is enabled.

Key reloading is not supported for session keys.

7.1. Usage under preload

PKCS #11 key reloading only reloads keys. It must also operate under a preload session during which preload is reloading tokens that protect the keys used by PKCS #11, in high availability mode. When the PKCS #11 application is using a token-protected key, `preload` should first be run to reload the token while PKCS #11 is reloading the key. For information on running `preload` for PKCS #11 key reloading, see section *PKCS #11* and JCE in the *User Guide* for your HSM.

PKCS #11 key reloading is also supported for module-protected keys, but the PKCS #11 application must still be run under a preload application which is reloading tokens for another key.

Either run the PKCS #11 application as a subprocess of preload, or in a separate command window ensuring the preload file set for preload matches the one set for PKCS #11. See section *nShield PKCS #11 library with the preload utility* in the *User Guide* for your HSM.

The application will attempt to reload keys when supported functions are called, see Supported function calls.

7.1.1. Persistent preload files

The preload file persists on disk after the preload process has terminated. Therefore, a PKCS #11 application in key reloading mode should not be run with an `NFAST_NFKM_TOKENSFILE` that points to a preload file from an old (non-running) preload process.

7.2. Supported function calls

Key reloading is attempted whenever a key is used for a cryptographic operation. For signing, verifying, encrypting, and decrypting, the functions are as follows:

- `C_SignInit`
- `C_VerifyInit`
- `C_EncryptInit`
- `C_DecryptInit`

On a call to any of these functions, the PKCS #11 library will do the following:

1. Checks if preload has reloaded any token objects on any HSMs since the last time one of the above functions was called. This is done by checking if the preload file has been modified. If not, there is nothing to reload.
2. If reload is required, reloads any keys that are protected by the newly-loaded tokens on all usable HSMs in the group.

7.3. Retrying key reloads

PKCS #11 can fail to reload a key due to transient or genuine errors. An example for a transient error is when an HSM has not finished reinitializing in time for a key to be reloaded. An example for a genuine error is when the key is invalid. In case of a failure, PKCS #11 will attempt to reload the key every time one of the functions in Supported function calls is called for a further 5 minutes before abandoning the key reload on that HSM.

7.4. Adding new HSMs

With key reloading enabled using the `CKNFAST_RELOAD_KEYS` environment variable, the PKCS

#11 library can add new HSMs to its internal list of usable modules. HSMs are new if they were not present when PKCS #11 applications were initialized. When key reloading is not enabled, PKCS #11 applications must be restarted before the new HSMs can be used.

The PKCS #11 library supports a maximum of 32 HSMs. If you have already reached 32 HSMs and you add a new HSM, then the PKCS #11 library will not be able to add this module. If an HSM is removed from the Security World or otherwise becomes unusable, it is still counted towards this limit. The application must be restarted to remove the removed or unusable HSM from the list.

8. PKCS #11 without load-sharing or HSM Pool modes

The nShield PKCS #11 library makes each nShield module appear to your PKCS #11 application as two or more PKCS #11 slots.

The first slot represents the module itself. This token:

- Appears as a non-removable hardware token and has the flag `CKF_REMOVABLE` not set
- Has the flag `CKF_LOGIN_REQUIRED` not set (`C_Login` always fails on this flag).

Applications can ignore this slot, but you can use the slot to store public session objects or for functions that do not use objects (such as `C_GenerateRandom`) even when the smart-card is not present.

The second slot represents the smart-card reader. This token:

- appears as a PKCS #11 slot, potentially containing a removable hardware token that has the flag `CKF_REMOVABLE` set
- is marked as removed if the smart card is removed from the physical slot
- has the flag `CKF_LOGIN_REQUIRED`
- allows the creation of token objects.

To use softcards with PKCS #11, load-sharing mode must be enabled.

A PKCS #11 token can support multiple concurrent sessions on multiple applications. However, by default, only one token may be logged in to a given slot at a given time (see K/N support for PKCS #11). By default, when you insert a new card into a slot, the nShield PKCS #11 library automatically logs out any token that had been logged in to the slot previously.

The `C_InitToken` function is *not* supported for use in non-load-sharing FIPS 140 Level 3 Security Worlds.

8.1. K/N support for PKCS #11

If you use the nShield PKCS #11 library without load-sharing mode or HSM Pool mode, you can implement *K*/*N* card set support in two ways:

- By using the nShield-specific API calls, `C_LoginBegin`, `C_LoginNext`, and `C_LoginEnd`
- By using the `preload` command-line utility to load the logical token first.

9. PKCS #11 Security Officer

The PKCS #11 Security Officer is a role that is created and managed by the `cksotool` utility. The utility creates a softcard and key, which are used to perform operations within the nShield PKCS #11 library as the Security Officer. The `ident`s of the generated softcard and key are `ncipher-pkcs11-so-softcard` and `ncipher-pkcs11-so-key`, respectively. They are used during Security Officer operations to provide the cryptographic security.

`ncipher-pkcs11-so-softcard` does not appear in the result of `C_GetSlotList` and therefore cannot be used to create PKCS #11 keys, or have its PIN changed using `C_SetPIN`.

To act as the Security Officer within the nShield PKCS #11 library, the Security Officer token and key must be preloaded using the `preload` utility:

```
preload -s ncipher-pkcs11-so-softcard pause
```

The PKCS #11 session must also be logged in as the user `CKU_SO`. `preload` is used so that virtual-slots in load-sharing can be logged into using the usual PKCS #11 API. This allows Security Officer operations to be performed on keys protected by any token.

It is strongly advised that operations that require loading the PKCS #11 Security Officer token are performed by a dedicated tool, and not integrated into a main application.

10. nShield-specific PKCS #11 API extensions

nShield *K*/*N* card sets use nShield-specific API calls. These calls can be used by the application in place of the standard `C_Login` to provide log-in to a card set with a K parameter greater than 1. The API calls include three functions, `C_LoginBegin`, `C_LoginNext` and `C_LoginEnd`.

The login sequence must occur in the same session.

You cannot use the API calls in load-sharing mode. To use *K*/*N* card sets in load-sharing mode, use `preload` to load the logical token first. The API calls also work in a non-load-sharing FIPS 140 Level 3 Security Worlds.

10.1. C_LoginBegin

Similar to `C_Login`, this function initiates the log-in process, ensures that the session is valid, and ensures that the user is not in load-sharing mode.

The `pulK` and `pulN` return values provide the caller with the number of card requests required. An example of the use of `C_LoginBegin` is shown here:

```
C_LoginBegin (CK_SESSION_HANDLE hSession, /* the session's handle */
              CK_USER_TYPE userType, /* the user type */
              CK_ULONG_PTR pulK, /* cards required to load logical token*/
              CK_ULONG_PTR pulN /* Number of cards in set */)
```

10.2. C_LoginNext

`C_LoginNext` is called *K* times until the required number of cards (for the given card set) have been presented. This function checks the Security World info to ensure that the card has changed each time. It also checks for the correct passphrase before loading the card share. `pulSharesLeft` allows the user application to assess the number of cards loaded to the number of cards required.

`CK_RV` gives various values that allow the user to access the application state using standard PKCS #11 return values (such as `CKR_TOKEN_NOT_RECOGNIZED`). These values reveal such information as whether the card is the same, whether the card is foreign or blank, and whether the passphrase was incorrect.

An example of the use of `C_LoginNext` is shown here:

```
C_LoginNext (CK_SESSION_HANDLE hSession, /* the session's handle */
             CK_USER_TYPE userType, /* the user type*/
```

```
            CK_CHAR_PTR pPin, /* the user's PIN*/
            CK_ULONG ulPinLen, /* the length of the PIN */
            CK_ULONG_PTR pulSharesLeft /* Number of shares still needed */)
```

10.3. C_LoginEnd

`C_LoginEnd` is called after all the shares are loaded. It constructs the logical token from the presented shares and then loads the private objects protected by the card set that are available to it:

```
C_LoginEnd (CK_SESSION_HANDLE hSession, /* the session's handle */
            CK_USER_TYPE userType /* the user type*/)
```

There must be no other calls between the functions, in that or any other session on the slot. In particular, a call that updates the Security World while using a card that has been removed at the time (for example, because a second card from the set is about to be inserted) returns `CKR_DEVICE_REMOVED` in the same way that it would for a single card. All sessions are then closed and the log-in process is aborted.

If other functions are accidentally called during the log-in cycle, then `slot.loadcardsetstate` is checked before updating the Security World. If the log-in process has not been completed, other functions return `CKR_FUNCTION_FAILED` and allow you to continue with the log-in process.

11. Compiling and linking

The following options are available if you want to integrate the nShield PKCS #11 library with your application. Depending on how your application integrates with PKCS #11 libraries, you can:

- statically link the nShield PKCS #11 library directly into your application
- dynamically link the nShield PKCS #11 library into your application
- create a plug-in shared library that contains the nShield position-independent code object files together with your own adaptation facilities.

You may freely supply your users with the compiled library files linked into your application or into a plug-in library used for your application.

The nShield PKCS #11 library includes the PKCS #11 header files `pkcs11.h`, `pkcs11t.h`, and `pkcs11f.h` from the RSA Data Security, Inc. Cryptoki Cryptographic Token Interface. Any work based on this interface is bound by the following terms of RSA Data Security, Inc. Licence, which states:

License is also granted to make and use derivative works provided that such works are identified as derived from the RSA Data Security, Inc. Cryptoki Cryptographic Token Interface in all material mentioning or referencing the derived work.

For more information about using the available libraries, see the `Include Paths and Linking` section in the *nCore API Documentation* on the Security World Software installation media.

11.1. Windows

All versions are built with Visual Studio 2017. Entrust supplies the following files:

- `%NFAST_HOME%\bin\cknfast.dll` and `%NFAST_HOME%\toolkits\pkcs11\cknfast.dll`: a dynamically linked library

  Both files are identical.

- `%NFAST_HOME%\c\ctd\lib\cknfast.lib`: a stub for applications that link to `cknfast.dll`
- `%NFAST_HOME%\c\ctd\lib\libcknfast.lib`: a static library with position-independent code

11.2. Linux

Entrust supplies the following libraries:

- `libcknfast.so`, `libcknfast.so.a`, or `libcknfast.so`: a standard, dynamically linked, shared library that can be used to create applications that must be dynamically linked with the nShield libraries at run time. On platforms where thread safety requires programs to be compiled differently from non-threaded programs, these libraries are compiled thread-safe.
- `libcknfast.a`: a standard, non-shared library used to statically link an application.
- `libcknfast_thrpic.a`: a non-shared library, compiled as threadsafe position-independent code.

On the Developer installation media, each library is provided with a corresponding set of header files. All the header files for each version are very similar, but some header files (particularly those that contain information about compiler and configuration options) differ by version.

These types of library are provided compiled with the following C compilers for Linux `libc6.11`:

Library Type	Build Notes
`/opt/nfast/c/ctd/gcc/lib`	This type of library is built with gcc 4.9.2 in 32-bit mode.
`/opt/nfast/c/csd/gcc/lib`	This type of library is built with gcc 4.9.2 in 64-bit mode.

12. Objects

Token objects are not stored in the nShield module. Instead, they are stored in an encrypted and integrity-protected form on the hard disk of the host computer. The key used for this encryption is created by combining information stored on the smart card with information stored in the nShield module and with the card passphrase.

Session keys are stored on the nShield module, while other session objects are stored in host memory. Token objects on the host are created in the `kmdata` directory. In order to access token objects, the user must have:

- the smart card
- the passphrase for the smart card
- an nShield module containing the module key used to create the token
- the host file containing the nShield key blob protecting the token object.

The nShield PKCS #11 library can be used to manipulate Data Objects, Certificate Objects, and Key Objects.

12.1. Certificate Objects and Data Objects

The nShield PKCS #11 library does not parse Certificate Objects or Data Objects.

The size of Data Objects is limited by what can be fitted into a single command (under most circumstances, this limit is 8192 bytes).

12.2. Key Objects

The following restrictions apply to keys:

Key types	Restrictions
RSA	Modulus greater than or equal to 1024. The nShield PKCS #11 library requires all of the attributes for an RSA key object to be supplied, as listed in Table 26: "RSA Private Key Object Attributes" of PKCS #11 Cryptographic Token Interface Standard version 2.40.
DSA	Modulus greater than or equal to 1024 in multiples of 8 bits.
Diffie-Hellman	Modulus greater than or equal to 1024.

12.3. Card passphrases

All passphrases are hashed using the SHA-1 hash mechanism and then combined with a module key to produce the key used to encrypt data on the nShield physical or software token. The passphrase supplied can be of any length.

The `ckinittoken` program imposes a 512-byte limit on the passphrase.

`C_GetTokenInfo` reports `_MaxPinLen` as 256 because some applications may have problems with a larger value.

When `C_Login` is called, the passphrase is used to load private objects protected by that card set on to all modules with cards from that set. Public objects belonging to that set are loaded on to all the modules. `C_Login` fails if any logical token fails to load. All cards in a card set must have the same passphrase.

The functions `C_SetPIN`, `C_InitPIN`, and `C_InitToken` are supported in load-sharing mode only when using softcards. To use these functions in load-sharing mode, you must have created a softcard with the command `ppmk -n` before selecting the corresponding slot.

The `C_InitToken` function is *not* supported for use in non-load-sharing FIPS 140 Level 3 Security Worlds.

13. Mechanisms

The following table lists the mechanisms currently supported by the nShield PKCS #11 library and the functions available to each one. Entrust also provides vendor-supplied mechanisms, described in Vendor-defined mechanisms.

Some mechanisms may be restricted from use in Security Worlds conforming to FIPS 140 Level 3. See the *User Guide* for your HSM for more information.

Mechanism	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive Key
CKM_AES_CBC_ENCRYPT_DATA	—	—	—	—	—	—	Y
CKM_AES_CBC_PAD	Y	—	—	—	—	Y	—
CKM_AES_CBC	Y	—	—	—	—	Y[1]	—
CKM_AES_CMAC_GENERAL	—	Y	—	—	—	—	—
CKM_AES_CMAC	—	Y	—	—	—	—	—
CKM_AES_CTR	Y	—	—	—	—	X	—
CKM_AES_ECB_ENCRYPT_DATA	—	—	—	—	—	—	Y
CKM_AES_ECB	Y	—	—	—	—	Y[1]	—
CKM_AES_GCM	Y	—	—	—	—	Y[13]	—
CKM_AES_KEY_GEN	—	—	—	—	Y	—	—
CKM_AES_KEY_WRAP	—	—	—	—	—	Y	—
CKM_AES_KEY_WRAP_PAD[2]	Y	—	—	—	—	Y	—
CKM_AES_KEY_WRAP_KWP	Y	—	—	—	—	Y	—
CKM_AES_MAC_GENERAL	—	Y	—	—	—	—	—
CKM_AES_MAC	—	Y	—	—	—	—	—
CKM_ARIA_CBC[16]	Y	—	—	—	—	Y[17]	—
CKM_ARIA_ECB[16]	Y	—	—	—	—	Y[17]	—
CKM_ARIA_KEY_GEN[16]	—	—	—	—	Y	—	—
CKM_CONCATENATE_BASE_AND_KEY	—	—	—	—	—	—	Y[3]

Mechanism	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive Key
CKM_DES_CBC_ENCRYPT_DATA	—	—	—	—	—	—	Y
CKM_DES_CBC_PAD	Y	—	—	—	—	Y	—
CKM_DES_CBC	Y	—	—	—	—	Y	—
CKM_DES_ECB_ENCRYPT_DATA	—	—	—	—	—	—	Y
CKM_DES_ECB	Y	—	—	—	—	Y	—
CKM_DES_KEY_GEN	—	—	—	—	Y	—	—
CKM_DES_MAC_GENERAL	—	Y	—	—	—	—	—
CKM_DES_MAC	—	Y	—	—	—	—	—
CKM_DES2_KEY_GEN	—	—	—	—	Y	—	—
CKM_DES3_CBC_ENCRYPT_DATA	—	—	—	—	—	—	Y
CKM_DES3_CBC_PAD	Y	—	—	—	—	Y	—
CKM_DES3_CBC	Y	—	—	—	—	Y[1]	—
CKM_DES3_ECB_ENCRYPT_DATA	—	—	—	—	—	—	Y
CKM_DES3_ECB	Y	—	—	—	—	Y[1]	—
CKM_DES3_KEY_GEN	—	—	—	—	Y	—	—
CKM_DES3_MAC_GENERAL	—	Y	—	—	—	—	—
CKM_DES3_MAC	—	Y	—	—	—	—	—
CKM_DH_PKCS_DERIVE	—	—	—	—	—	—	Y
CKM_DH_PKCS_KEY_PAIR_GEN	—	—	—	—	Y	—	—
CKM_DSA_KEY_PAIR_GEN	—	—	—	—	Y	—	—
CKM_DSA_PARAMETER_GEN	—	—	—	—	Y	—	—
CKM_DSA_SHA1	—	Y	—	—	—	—	—
CKM_DSA	—	Y[4]	—	—	—	—	—
CKM_EC_EDWARDS_KEY_PAIR_GEN	—	—	—	—	Y[5]	—	—

Mechanism	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive Key
CKM_EC_KEY_PAIR_GEN	—	—	—	—	Y[6]	—	—
CKM_EC_MONTGOMERY_KEY_PAIR_GEN	—	—	—	—	Y[5]	—	—
CKM_ECDH1_DERIVE	—	—	—	—	—	—	Y[7]
CKM_ECDSA_SHA1	—	Y	—	—	—	—	—
CKM_ECDSA_SHA224	—	Y	—	—	—	—	—
CKM_ECDSA_SHA256	—	Y	—	—	—	—	—
CKM_ECDSA_SHA384	—	Y	—	—	—	—	—
CKM_ECDSA_SHA512	—	Y	—	—	—	—	—
CKM_ECDSA_SHA3_224	—	Y	—	—	—	—	—
CKM_ECDSA_SHA3_256	—	Y	—	—	—	—	—
CKM_ECDSA_SHA3_384	—	Y	—	—	—	—	—
CKM_ECDSA_SHA3_512	—	Y	—	—	—	—	—
CKM_EDDSA	—	Y[4,8]	—	—	—	—	—
CKM_ECDSA	—	Y[4]	—	—	—	—	—
CKM_GENERIC_SECRET_KEY_GEN	—	—	—	—	Y	—	—
CKM_MD5_HMAC_GENERAL	—	Y	—	—	—	—	—
CKM_MD5_HMAC	—	Y	—	—	—	—	—
CKM_MD5	—	—	—	Y	—	—	—
CKM_NC_ECIES	—	—	—	—	—	Y[9]	—
CKM_NC_MD5_HMAC_KEY_GEN	—	—	—	—	Y	—	—
CKM_NC_MILENAGE	—	Y[4,15]	—	—	—	—	—
CKM_NC_MILENAGE_AUTS	—	Y[4,15]	—	—	—	—	—
CKM_NC_MILENAGE_RESYNC	—	Y[4,15]	—	—	—	—	—
CKM_NC_MILENAGE_OPC	—	—	—	—	—	—	Y
CKM_NC_MILENAGEOP_KEY_GEN	—	—	—	—	Y	—	—

Mechanism	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive Key
CKM_NC_MILENAGERC_KEY_GEN	—	—	—	—	Y	—	—
CKM_NC_MILENAGESUBSCRIBER_KEY_GEN	—	—	—	—	Y	—	—
CKM_NC_TUAK	—	Y[4,15]	—	—	—	—	—
CKM_NC_TUAK_AUTS	—	Y[4,15]	—	—	—	—	—
CKM_NC_TUAK_RESYNC	—	Y[4,15]	—	—	—	—	—
CKM_NC_TUAK_TOPC	—	—	—	—	—	—	Y
CKM_NC_TUAKSUBSCRIBER_KEY_GEN	—	—	—	—	Y	—	—
CKM_NC_TUAKTOP_KEY_GEN	—	—	—	—	Y	—	—
CKM_PBE_MD5_DES_CBC	—	—	—	—	Y	—	—
CKM_RIPEMD160	—	—	—	Y	—	—	—
CKM_RSA_9796	—	Y[4]	Y[4]	—	—	—	—
CKM_RSA_AES_KEY_WRAP	—	—	—	—	—	Y[14]	—
CKM_RSA_PKCS_KEY_PAIR_GEN	—	—	—	—	Y	—	—
CKM_RSA_PKCS_OAEP	Y	—	—	—	—	Y	—
CKM_RSA_PKCS_PSS[11]	Y	Y	—	—	—	—	—
CKM_RSA_PKCS	Y[4]	Y[4]	Y[4]	—	—	Y	—
CKM_RSA_X_509	Y[4]	Y[4]	Y[4]	—	—	X	—
CKM_RSA_X9_31_KEY_PAIR_GEN	—	—	—	—	Y	—	—
CKM_SHA_1_HMAC_GENERAL	—	Y[10]	—	—	—	—	—
CKM_SHA_1_HMAC	—	Y[10]	—	—	—	—	—
CKM_SHA_1	—	—	—	Y	—	—	—
CKM_SHA1_RSA_PKCS_PSS[11]	—	Y	—	—	—	—	—
CKM_SHA1_RSA_PKCS	—	Y	—	—	—	—	—

Mechanism	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive Key
CKM_SHA224_HMAC_GENERAL	—	Y[10]	—	—	—	—	—
CKM_SHA224_HMAC	—	Y[10]	—	—	—	—	—
CKM_SHA224_RSA_PKCS_PSS[11]	—	Y	—	—	—	—	—
CKM_SHA224_RSA_PKCS	—	Y	—	—	—	—	—
CKM_SHA224	—	—	—	Y	—	—	—
CKM_SHA256_HMAC_GENERAL	—	Y[10]	—	—	—	—	—
CKM_SHA256_HMAC	—	Y[10]	—	—	—	—	—
CKM_SHA256_RSA_PKCS_PSS[11]	—	Y	—	—	—	—	—
CKM_SHA256_RSA_PKCS	—	Y	—	—	—	—	—
CKM_SHA256	—	—	—	Y	—	—	—
CKM_SHA384_HMAC_GENERAL	—	Y[10]	—	—	—	—	—
CKM_SHA384_HMAC	—	Y[10]	—	—	—	—	—
CKM_SHA384_RSA_PKCS_PSS[11]	—	Y	—	—	—	—	—
CKM_SHA384_RSA_PKCS	—	Y	—	—	—	—	—
CKM_SHA384	—	—	—	Y	—	—	—
CKM_SHA512_HMAC_GENERAL	—	Y[10]	—	—	—	—	—
CKM_SHA512_HMAC	—	Y[10]	—	—	—	—	—
CKM_SHA512_RSA_PKCS_PSS[11]	—	Y	—	—	—	—	—
CKM_SHA512_RSA_PKCS	—	Y	—	—	—	—	—
CKM_SHA512	—	—	—	Y	—	—	—
CKM_SHA3_224	—	—	—	Y	—	—	—
CKM_SHA3_224_RSA_PKCS_PSS[11]	—	Y	—	—	—	—	—
CKM_SHA3_224_RSA_PKCS	—	Y	—	—	—	—	—

Mechanism	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive Key
CKM_SHA3_224_HMAC_GENERAL	—	Y	—	—	—	—	—
CKM_SHA3_224_HMAC	—	Y	—	—	—	—	—
CKM_SHA3_224_KEY_GEN	—	—	—	—	Y	—	—
CKM_SHA3_256	—	—	—	Y	—	—	—
CKM_SHA3_256_RSA_PKCS_PSS[11]	—	Y	—	—	—	—	—
CKM_SHA3_256_RSA_PKCS	—	Y	—	—	—	—	—
CKM_SHA3_256_HMAC_GENERAL	—	Y	—	—	—	—	—
CKM_SHA3_256_HMAC	—	Y	—	—	—	—	—
CKM_SHA3_256_KEY_GEN	—	—	—	—	Y	—	—
CKM_SHA3_384	—	—	—	Y	—	—	—
CKM_SHA3_384_RSA_PKCS_PSS[11]	—	Y	—	—	—	—	—
CKM_SHA3_384_RSA_PKCS	—	Y	—	—	—	—	—
CKM_SHA3_384_HMAC_GENERAL	—	Y	—	—	—	—	—
CKM_SHA3_384_HMAC	—	Y	—	—	—	—	—
CKM_SHA3_384_KEY_GEN	—	—	—	—	Y	—	—
CKM_SHA3_512	—	—	—	Y	—	—	—
CKM_SHA3_512_RSA_PKCS_PSS[11]	—	Y	—	—	—	—	—
CKM_SHA3_512_RSA_PKCS	—	Y	—	—	—	—	—
CKM_SHA3_512_HMAC_GENERAL	—	Y	—	—	—	—	—
CKM_SHA3_512_HMAC	—	Y	—	—	—	—	—
CKM_SHA3_512_KEY_GEN	—	—	—	—	Y	—	—
CKM_XOR_BASE_AND_DATA	—	—	—	—	—	—	Y[12]

The nShield library supports some mechanisms that are defined in versions of the PKCS #11 standard later than 2.01, although the nShield library does not fully support versions of the

PKCS #11 standard later than 2.01.

In the table above:

- Empty cells indicate mechanisms that are not supported by the PKCS #11 standard.
- The entry **Y** indicates that a mechanism is supported by the nShield PKCS #11 library.
- The entry **X** indicates that a mechanism is not supported by the nShield PKCS #11 library.

In the table above, annotations with the following numbers indicate:

13.1. Footnote 1

Wrap secret keys only (private key wrapping must use `CBC_PAD`).

13.2. Footnote 2

`CKM_AES_KEY_WRAP_PAD` has been deprecated and replaced by `CKM_AES_KEY_WRAP_KWP`.

13.3. Footnote 3

Before you can create a key for use with the derive mechanism `CKM_CONCATENATE_BASE_AND_KEY`, you must specify the `CKA_ALLOWED_MECHANISMS` attribute in the template with the `CKM_CONCATENATE_BASE_AND_KEY` set. Specifying the `CKA_ALLOWED_MECHANISMS` in the template enables the setting of the nCore level ACL, which enables the key in this derive key operation. For more information about the `CKA_ALLOWED_MECHANISMS` attribute, see Attributes.

13.4. Footnote 4

Single-part operations only.

13.5. Footnote 5

`CKA_EC_PARAMS` is a DER-encoded PrintableString `curve25519`.

13.6. Footnote 6

If no capabilities are specified in the template, for example the CKA_DERIVE, CKA_SIGN and CKA_UNWRAP attributes are omitted, then the default capability is sign/verify.

Key generation does calculate its own curves but, as shown in the PKCS #11 standard, takes the CKA_PARAMS, which contains the curve information (similar to that of a discrete logarithm group in the generation of a DSA key pair). CKA_EC_PARAMS is a Byte array which is DER-encoded of an ANSI X9.62 Parameters value. It can take both named curves and custom curves.

The following PKCS #11-specific flags describe which curves are supported:

- CKF_EC_P: prime curve supported
- CKF_EC_2M: binary curve supported
- CKF_EC_PARAMETERS: supplying your own custom parameters is supported
- CKF_EC_NAMECURVE: supplying a named curve is supported
- CKF_EC_UNCOMPRESS: supports uncompressed form only, compressed form not supported.

13.7. Footnote 7

The CKM_ECDH1_DERIVE mechanism is supported. However, the mechanism only takes a CK_ECDH1_DERIVE_PARAMS struct in which CK_EC_KDF_TYPE can be one of the following:

- CKD_NULL
- CKD_SHA1_KDF, CKD_SHA1_KDF_SP800
- CKD_SHA224_KDF, CKD_SHA224_KDF_SP800
- CKD_SHA256_KDF, CKD_SHA256_KDF_SP800
- CKD_SHA384_KDF, CKD_SHA384_KDF_SP800
- CKD_SHA512_KDF, CKD_SHA512_KDF_SP800
- CKD_SHA3_224_KDF, CKD_SHA3_224_KDF_SP800
- CKD_SHA3_256_KDF, CKD_SHA3_256_KDF_SP800
- CKD_SHA3_384_KDF, CKD_SHA3_384_KDF_SP800
- CKD_SHA3_512_KDF, CKD_SHA3_512_KDF_SP800

For more information on CK_ECDH1_DERIVE_PARAMS, see the PKCS #11 standard.

For the pPublicData* parameter, a raw octet string value (as defined in section A.5.2 of ANSI X9.62) and DER-encoded ECPoint value (as defined in section E.6 of ANSI X9.62) are accepted for CKK_EC keys. RFC 7748 encoding should be used for CKK_EC_MONTGOMERY keys.

13.8. Footnote 8

Both the Ed25519 and Ed25519ph signature schemes are supported, The Ed25519 scheme requires either no CK_EDDSA_PARAMS to be passed or if it is passed it should have the following set:

- phFlag to CK_FALSE
- ulContextDataLen to 0.

The Ed25519ph signature scheme requires CK_EDDSA_PARAMS to have the following set:

- phFlag to CK_TRUE
- ulContextDataLen to 0.

13.9. Footnote 9

Wrap secret keys only.

13.10. Footnote 10

This mechanism depends on the vendor-defined key generation mechanism CKM_NC_SHA_1_HMAC_KEY_GEN, CKM_NC_SHA224_HMAC_KEY_GEN, CKM_NC_SHA256_HMAC_KEY_GEN, CKM_NC_SHA384_HMAC_KEY_GEN, or CKM_NC_SHA512_HMAC_KEY_GEN. For more information, see Vendor-defined mechanisms.

13.11. Footnote 11

The hashAlg and the mgf that are specified by the CK_RSA_PKCS_PSS_PARAMS must have the same SHA hash size. If they do not have the same hash size, then the signing or verify fails with a return value of CKR_MECHANISM_PARAM_INVALID.

The sLen value is expected to be the length of the message hash. If this is not the case, then the signing or verify again fails with a return value of CKR_MECHANISM_PARAM_INVALID. The Security World Software implementation of RSA_PKCS_PSS salt lengths are as follows:

Mechanism	Salt-length
SHA-1	160-bit
SHA-224	224-bit
SHA-256	256-bit

Mechanism	Salt-length
SHA-384	384-bit
SHA-512	512-bit
SHA3-224	224-bit
SHA3-256	256-bit
SHA3-384	384-bit
SHA3-512	512-bit

13.12. Footnote 12

The base key and the derived key are restricted to `DES`, `DES3`, `CAST5` or `Generic`, though they may be of different types.

13.13. Footnote 13

For wrap and unwrap with `CKM_AES_GCM`, the `IV` supplied in the `CKM_GCM_PARAMS` structure must be 12 bytes. For wrap the IV must be all zeroes. This will be overwritten by the actual value used when the wrap command has completed successfully. For unwrap the `IV` must be the value returned by the corresponding wrap.

13.14. Footnote 14

In order to create an unwrapping key for use with the mechanism `CKM_RSA_AES_KEY_WRAP` where `CKA_UNWRAP_TEMPLATE` is also set, you must:

- Specify the `CKA_ALLOWED_MECHANISMS` attribute in the template with `CKM_RSA_AES_KEY_WRAP` set as an allowed mechanism.
- Override the Security Assurance Mechanisms (SAMs) to permit use of `CKA_UNWRAP_TEMPLATE` with the mechanism `CKM_RSA_AES_KEY_WRAP`.

Keys with `CKA_WRAP_WITH_TRUSTED` set cannot be wrapped with the mechanism `CKM_RSA_AES_KEY_WRAP`. The `C_WrapKey` operation will return `CKR_KEY_NOT_WRAPPABLE` for such keys.

With firmware versions 13.4 or later, you do not need to override the Security Assurance Mechanisms. Keys with `CKA_WRAP_WITH_TRUSTED` can be wrapped with the mechanism `CKM_RSA_AES_KEY_WRAP`.

For more information about the SAMs, see PKCS #11 security assurance mechanism. For more information about the `CKA_ALLOWED_MECHANISMS` attribute, see Attributes.

13.15. Footnote 15

Sign only.

13.16. Footnote 16

Use of these mechanisms requires the `KISAAlgorithms` feature to be enabled, see the User Guide for your HSM for more information.

13.17. Footnote 17

Wraps secret keys only.

14. Vendor annotations on P11 mechanisms

Vendor notes on PKCS #11 mechanisms to complement the specification.

14.1. CKM_RSA_PKCS_OAEP

The hashAlg and the mgf values specified by CK_RSA_PKCS_OAEP_PARAMS must have the same SHA hash size. If they do not have the same hash size, then the encryption or decryption fails with a return value of CKR_MECHANISM_PARAM_INVALID. The supported pairs of values are as follows:

hashAlg	mgf
CKM_SHA_1	CKG_MGF1_SHA1
CKM_SHA224	CKG_MGF1_SHA224
CKM_SHA256	CKG_MGF1_SHA256
CKM_SHA384	CKG_MGF1_SHA384
CKM_SHA512	CKG_MGF1_SHA512
CKM_SHA3_224	CKG_MGF1_SHA3_224
CKM_SHA3_256	CKG_MGF1_SHA3_256
CKM_SHA3_384	CKG_MGF1_SHA3_384
CKM_SHA3_512	CKG_MGF1_SHA3_512

For a hash length h and RSA modulus length k in bytes, the longest message that can be encrypted is k-2h-2 bytes long.

14.2. CKM_RSA_PKCS_PSS and CKM_SHA*_RSA_PKCS_PSS

The hashAlg and the mgf values specified by CK_RSA_PKCS_PSS_PARAMS must have the same SHA hash size. If they do not have the same hash size, then the signing or verifying fails with a return value of CKR_MECHANISM_PARAM_INVALID.

The sLen value is expected to be the length of the message hash in bytes. If this is not the case, then the signing or verify again fails with a return value of CKR_MECHANISM_PARAM_INVALID.

The supported sets of values for hashAlg, mgf and sLen are as follows:

hashAlg	mgf	sLen
CKM_SHA_1	CKG_MGF1_SHA1	20
CKM_SHA224	CKG_MGF1_SHA224	28
CKM_SHA256	CKG_MGF1_SHA256	32
CKM_SHA384	CKG_MGF1_SHA384	48
CKM_SHA512	CKG_MGF1_SHA512	64
CKM_SHA3_224	CKG_MGF1_SHA3_224	28
CKM_SHA3_256	CKG_MGF1_SHA3_256	32
CKM_SHA3_384	CKG_MGF1_SHA3_384	48
CKM_SHA3_512	CKG_MGF1_SHA3_512	64

To use a mechanism with SHA hash size n bits, the public modulus of the RSA key must be at least 2n+2 bits long.

15. Vendor-defined mechanisms

The following vendor-defined mechanisms are also available. The numeric values of vendor-defined key types and mechanisms can be found in the supplied `pkcs11extra.h` header file.

Some mechanisms may be restricted from use in Security Worlds conforming to FIPS 140 Level 3. See the *User Guide* for your HSM for more information.

15.1. CKM_SEED_ECB_ENCRYPT_DATA and CKM_SEED_CBC_ENCRYPT_DATA

This mechanism derives a secret key by encrypting plain data with the specified secret base key. This mechanism takes as a parameter a `CK_KEY_DERIVATION_STRING_DATA` structure, which specifies the length and value of the data to be encrypted by using the base key to derive another key.

If no length or key type is provided in the template, the key produced by this mechanism is a generic secret key. Its length is equal to the length of the data.

If a length, but no key type, is provided in the template, the key produced by this mechanism is a generic secret key of the specified length.

If a key type, but no length, is provided in the template, the key type must have a well-defined length. If the length is well defined, the key produced by this mechanism is of the type specified in the template. If the length is not well defined, a `CKR_TEMPLATE_INCOMPLETE` error is returned.

If both a key type and a length are provided in the template, the length must be compatible with that key type, and `CKR_TEMPLATE_INCONSISTENT` is returned if it is not.

The key produced by the `CKM_SEED_ECB_ENCRYPT_DATA` or `CKM_SEED_CBC_ENCRYPT_DATA` mechanisms is of the specified type and length.

15.2. CKM_CAC_TK_DERIVATION

This mechanism uses `C_GenerateKey` to perform an `Import` operation using a Transport Key Component.

The mechanism accepts a template that contains three Transport Key Components (TKCs) with following attribute types:

- CKA_TKC1
- CKA_TKC2
- CKA_TKC3.

These attributes are all in the CKA_VENDOR_DEFINED range.

Each TKC should be the same length as the key being created. TKCs used for DES, DES2, or DES3 keys must have odd parity. The mechanism checks for odd parity and returns CKR_ATTRIBUTE_VALUE_INVALID if it is not found.

The new key is constructed by an XOR of the three TKC components on the module.

Although using C_GenerateKey creates a key with a known value rather than generating a new one, it is used because C_CreateObject does not accept a mechanism parameter.

CKA_LOCAL, CKA_ALWAYS_SENSITIVE, and CKA_NEVER_EXTRACTABLE are set to FALSE, as they would for a key imported with C_CreateObject. This reflects the fact that the key was not generated locally.

An example of the use of CKM_CAC_TK_DERIVATION is shown here:

```
CK_OBJECT_CLASS class_secret = CKO_SECRET_KEY;
        CK_KEY_TYPE key_type_des2 = CKK_DES2;
        CK_MECHANISM mech = { CKM_CAC_TK_DERIVATION, NULL_PTR, 0 };
        CK_BYTE TKC1[16] = { ... };
        CK_BYTE TKC2[16] = { ... };
        CK_BYTE TKC3[16] = { ... };
        CK_OBJECT_HANDLE kHey;
        CK_ATTRIBUTE pTemplate[] = {
                { CKA_CLASS, &class_secret, sizeof(class_secret) },
                { CKA_KEY_TYPE, &key_type_des2, sizeof(key_type_des2) },
                { CKA_TKC1, TKC1, sizeof(TKC1) },
                { CKA_TKC2, TKC1, sizeof(TKC2) },
                { CKA_TKC3, TKC1, sizeof(TKC3) },
                { CKA_ENCRYPT, &true, sizeof(true) },
        ....
        };

        rv = C_GenerateKey(hSession, &mechanism, pTemplate,
                (sizeof(pTemplate)/sizeof((pTemplate)[0])), &hKey);
```

15.3. CKM_SHA*_HMAC and CKM_SHA*_HMAC_GENERAL

This version of the library supports the following mechanisms:

- CKM_SHA_1_HMAC
- CKM_SHA_1_HMAC_GENERAL
- CKM_SHA224_HMAC

- CKM_SHA224_HMAC_GENERAL
- CKM_SHA256_HMAC
- CKM_SHA256_HMAC_GENERAL
- CKM_SHA384_HMAC
- CKM_SHA384_HMAC_GENERAL
- CKM_SHA512_HMAC
- CKM_SHA512_HMAC_GENERAL
- CKM_SHA3_224_HMAC
- CKM_SHA3_224_HMAC_GENERAL
- CKM_SHA3_256_HMAC
- CKM_SHA3_256_HMAC_GENERAL
- CKM_SHA3_384_HMAC
- CKM_SHA3_384_HMAC_GENERAL
- CKM_SHA3_512_HMAC
- CKM_SHA3_512_HMAC_GENERAL

For security reasons, the Security World Software supports these mechanisms only with their own specific key type. Thus, you can only use an HMAC key with the HMAC algorithm and not with other algorithms.

The key types provided for use with SHA<n> HMAC mechanisms are:

- CKK_SHA_1_HMAC
- CKK_SHA224_HMAC
- CKK_SHA256_HMAC
- CKK_SHA384_HMAC
- CKK_SHA512_HMAC
- CKK_SHA3_224_HMAC
- CKK_SHA3_256_HMAC
- CKK_SHA3_384_HMAC
- CKK_SHA3_512_HMAC

To generate the key, use the appropriate key generation mechanism (which does not take any mechanism parameters):

- CKM_NC_MD5_HMAC_KEY_GEN
- CKM_NC_SHA_1_HMAC_KEY_GEN
- CKM_NC_SHA224_HMAC_KEY_GEN

- CKM_NC_SHA256_HMAC_KEY_GEN
- CKM_NC_SHA384_HMAC_KEY_GEN
- CKM_NC_SHA512_HMAC_KEY_GEN
- CKM_SHA3_224_KEY_GEN
- CKM_SHA3_256_KEY_GEN
- CKM_SHA3_384_KEY_GEN
- CKM_SHA3_512_KEY_GEN

15.4. CKM_NC_ECKDF_HYPERLEDGER

This version of the library supports the vendor-defined CKM_NC_ECKDF_HYPERLEDGER mechanism. This key derivation function is used in the user/client enrolment process of a hyperledger system to generate transaction certificates by using the enrolment certificate as one of the inputs to the key derivation.

The parameters for the mechanism are defined in the following structure:

```
typedef struct CK_ECKDF_HYPERLEDGERCLIENT_PARAMS {
    CK_OBJECT_HANDLE hKeyDF_Key;
    CK_MECHANISM_TYPE HMACMechType;
    CK_MECHANISM_TYPE TCertEncMechType;
    CK_ULONG ulEksize;
    CK_BYTE_PTR pEncTCertData;
    CK_ULONG ulEvsize;
    CK_ULONG ulEndian;
} CK_ECKDF_HYPERLEDGERCLIENT_PARAMS
```

Where:

- hKeyDF_key is **KeyDF_Key**
- HMACMechType is **Hmac**
- TCertEncMechType is **Decrypt_Mech**
- ulEksize is **Eksize**
- pEncTCertData is a pointer to encrypted data containing TCertIndex together with padding and IV
- ulEvsize is **Evsize**
- ulEndian is **Big_Endian**

The function is then called as follows:

```
C_DeriveKey(
    hSession,
    &mechanism_hyperledger,
    EnrollPriv_Key,
```

```
    TCertPriv_Key_template,
    NUM(TCertPriv_Key_template,
    &TCertPriv_Key);
```

A **Template_Key** will be used to supply key attributes for the resulting derived key. The derived key can then be used in the normal way.

Derived keys can be exported and used outside the HSM only if the template key was created with attributes which allow export of its derived keys.

15.5. CKM_HAS160

This version of the library supports the vendor-defined `CKM_HAS160` hash (digest) mechanism for use with the `CKM_KCDSA` mechanism. For more information, see KISAAlgorithm mechanisms.

`CKM_HAS160` is a basic hashing algorithm. The hashing is done on the host machine. This algorithm can be used by means of the standard digest function calls of the PKCS #11 API.

15.6. CKM_PUBLIC_FROM_PRIVATE

`CKM_PUBLIC_FROM_PRIVATE` is a derive key mechanism that enables the creation of a corresponding public key from a private key. The mechanism also fills in the public parts of the private key, where this has not occurred.

`CKM_PUBLIC_FROM_PRIVATE` is an nShield specific nCore mechanism. The `C_Derive` function takes the object handle of the private key and the public key attribute template. The creation of the key is based on the template but also checked against the attributes of the private key to ensure the attributes are correct and match those of the corresponding key. If an operation that is not allowed or is not set by the private key is detected, then `CKR_TEMPLATE_INCONSISTANT` is returned.

Before you can use this mechanism, the HSM must already contain the private key. You must use `C_CreateObject`, `C_UnWrapKey`, or `C_GenerateKeyPair` to import or generate the private key.

If you use `C_GenerateKeyPair`, you always generate a public key at the same time as the private key. Some applications delete public keys once a certificate is imported, but in the case of both `C_GenerateKeyPair` and `C_CreateObject` you can use either the `CKM_PUBLIC_FROM_PRIVATE` mechanism or the `C_GetAttributeValue` to recreate a deleted public key.

15.7. CKM_NC_AES_CMAC

CKM_NC_AES_CMAC is based on the Mech_RijndaelCMAC nCore level mechanism, a message authentication code operation that is used with both C_Sign and C_SignUpdate, and the corresponding C_Verify and C_VerifyUpdate functions.

In a similar way to other AES MAC mechanisms, CKM_NC_AES_CMAC takes a plaintext type of any length of bytes, and returns a M_Mech_Generic128MAC_Cipher standard byte block. CKM_NC_AES_CMAC is a standard FIPS 140 Level 3 approved mechanism, and is only usable with CKK_AES key types.

CKM_NC_AES_CMAC has a CK_MAC_GENERAL_PARAMS which is the length of the MAC returned (sometimes called a tag length). If this is not specified, the signing operation fails with a return value of CKR_MECHANISM_PARAM_INVALID.

15.8. CKM_NC_AES_CMAC_KEY_DERIVATION and CKM_NC_AES_CMAC_KEY_DERIVATION_SCP03

This mechanism derives a secret key by validating parameters with the specified 128-bit, 192-bit, or 256-bit secret base AES key. This mechanism takes as a parameter a CK_NC_AES_CMAC_KEY_DERIVATION_PARAMS structure, which specifies the length and type of the resulting derived key.

CKM_NC_AES_CMAC_KEY_DERIVATION_SCP03 is a variant of CKM_NC_AES_CMAC_KEY_DERIVATION: it reorders the arguments in the CK_NC_AES_CMAC_KEY_DERIVATION_PARAMS according to payment specification SCP03, but is otherwise identical.

The standard key attribute behavior with sensitive and extractable attributes is applied to the resulting key as defined in PKCS #11 standard version 2.20 and later. The key type and template declaration is based on the PKCS #11 standard key declaration for derive key mechanisms.

If no length or key type is provided in the template, the key produced by this mechanism is a generic secret key. Its length is equal to the length of the data.

If a length, but no key type, is provided in the template, the key produced by this mechanism is a generic secret key of the specified length.

If a key type, but no length, is provided in the template, the key type must have a well-defined length. If the length is well defined, the key produced by this mechanism is of the type specified in the template. If the length is not well defined, a CKR_TEMPLATE_INCOMPLETE error is returned.

If both a key type and a length are provided in the template, the length must be compatible with that key type, and `CKR_TEMPLATE_INCONSISTENT` is returned if it is not.

The key produced by the `CKM_NC_AES_CMAC_KEY_DERIVATION` mechanism is of the specified type and length. If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key are set properly. If the requested type of key requires more bytes than are available by concatenating the original key values, an error is generated.

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

Attribute	**If the attributes for the *original keys* are...**	**The attribute for the *derived key* is...**
CKA_SENSITIVE	CK_TRUE for either one	CK_TRUE
CKA_EXTRACTABLE	CK_FALSE for either one	CK_FALSE
CKA_ALWAYS_SENSITIVE	CK_TRUE for both	CK_TRUE
CKA_NEVER_EXTRACTABLE	CK_TRUE for both	CK_TRUE

15.9. CK_NC_AES_CMAC_KEY_DERIVATION_PARAMS

```
typedef struct CK_NC_AES_CMAC_KEY_DERIVATION_PARAMS {
    CK_ULONG ulContextLen;
    CK_BYTE_PTR pContext;
    CK_ULONG ulLabelLen;
    CK_BYTE_PTR pLabel;
  } CK_NC_AES_CMAC_KEY_DERIVATION_PARAMS;
```

The fields of the structure have the following meanings:

Argument	**Meaning**
ulContextLen	Context data: the length in bytes.
pContext	Some data info context data (bytes to be CMAC'd). `ulContextLen` must be zero if `pContext` is not provided. Having `pContext` as NULL will result in the same predictable key each time not additional data to add to the mix when carrying out the CMAC.
ulLabelLen	The length in bytes of the other party EC public key
pLabel	Key derivation label data: a pointer to the other label to identify new key. `ulLabelLen` must be zero if the `pLabel` is not provided.

15.10. CKM_COMPOSITE_EMV_T_ARQC, CKM_WATCHWORD_PIN1 and CKM_WATCHWORD_PIN2

These mechanisms allow the module to act as a SafeSign Cryptomodule (SSCM). To obtain support for your product, visit https://nshieldsupport.entrust.com.

15.11. CKM_NC_ECIES

This version of the library supports the vendor defined CKM_NC_ECIES mechanism. This mechanism is used with C_WrapKey and C_UnwrapKey to wrap and unwrap symmetric keys using the Elliptic Curve Integrated Encryption Scheme (ECIES).

The parameters for the mechanism are defined in the following structure:

```
typedef struct CK_NC_ECIES_PARAMS {
       CK_MECHANISM_PTR <pAgreementMechanism>;
       CK_MECHANISM_PTR <pSymmetricMechanism>;
       CK_ULONG         <ulSymmetricKeyBitLen>;
       CK_MECHANISM_PTR <pMacMechanism>;
       CK_ULONG         <ulMacKeyBitLen>;
} CK_NC_ECIES_PARAMS;
```

Where:

- <pAgreementMechanism> is the key agreement mechanism, which must be CKM_ECDH1_DERIVE or CKM_ECDH1_COFACTOR_DERIVE
- <pSymmetricMechanism> is the confidentiality mechanism, currently only CKM_XOR_BASE_AND_DATA is supported
- <ulSymmetricKeyBitLen> is the confidentiality key length (in bits) and must be a multiple of 8. For CKM_XOR_BASE_AND_DATA the key length is irrelevant and can be set to zero.
- <pMacMechanism> is the integrity mechanism, currently only CKM_SHA<n>_HMAC_GENERAL is supported and <n> can be _1, 224, 256, 384 or 512
- <ulMacKeyBitLen> is the integrity key length (in bits) and must be a multiple of 8

The following example shows how to use CKM_NC_ECIES to wrap a symmetric key:

```
/* session represents an existing open session */
CK_SESSION_HANDLE session;

/* symmetric_key and wrapping_key represent existing keys. The code to import or
 * generate them is not shown here. Note wrapping_key must be a public EC key
 * with CKA_WRAP set to true */
CK_OBJECT_HANDLE symmetric_key;
CK_OBJECT_HANDLE wrapping_key;

CK_ECDH1_DERIVE_PARAMS ecdh1_params = { CKD_SHA256_KDF };
```

```
CK_MECHANISM agreement_mech = {
       CKM_ECDH1_DERIVE,
       &ecdh1_params,
       sizeof(CK_ECDH1_DERIVE_PARAMS)
};
CK_MECHANISM symmetric_mech = { CKM_XOR_BASE_AND_DATA };
CK_MAC_GENERAL_PARAMS mac_params = 16;
CK_MECHANISM mac_mech = {
       CKM_SHA256_HMAC_GENERAL,
       &mac_params,
       sizeof(CK_MAC_GENERAL_PARAMS)
};
CK_NC_ECIES_PARAMS ecies_params = {
       &agreement_mech,
       &symmetric_mech,
       0,
       &mac_mech,
       256
};
CK_MECHANISM ecies_mech = {
       CKM_NC_ECIES,
       &ecies_params,
       sizeof(CK_NC_ECIES_PARAMS)
};

/* Typical convention is to call C_WrapKey with the pWrappedKey parameter set to
 * NULL_PTR to determine the required size of the buffer - see Section 5.2 of
 * the PKCS#11 Base Specification - but for brevity we allocate a 1KB buffer */
CK_BYTE wrapped_key[1000] = { 0 };
CK_ULONG wrapped_len = sizeof(wrapped_key);
CK_RV rv = C_WrapKey(session, &ecies_mech, wrapping_key, symmetric_key,
                                            wrapped_key, &wrapped_len);
```

15.12. CKM_NC_MILENAGE_OPC

Derive CKK_NC_MILENAGEOPC key from CKK_NC_MILENAGEOP and CKK_NC_MILENAGESUBSCRIBER keys for use in the 3GPP mechanisms defined in ETSI TS 135 206 s4.1.

A C_DeriveKey function call is made. The function takes the CKK_NC_MILENAGESUBSCRIBER key handle as the base key and the CKK_NC_MILENAGEOP key handle as the mechanism parameter.

To generate the subscriber and OP keys, use the corresponding vendor-defined key generation mechanisms (which do not take any mechanism parameters):

- CKM_NC_MILENAGESUBSCRIBER_KEY_GEN
- CKM_NC_MILENAGEOP_KEY_GEN

15.13. CKM_NC_MILENAGE, CKM_NC_MILENAGE_AUTS, CKM_NC_MILENAGE_RESYNC

3GPP mechanisms for 5G mobile networks as defined by ETSI TS 135 206. Used with

C_SignInit and C_Sign function calls. The parameters for these mechanisms are defined in the following structure:

```
typedef struct CK_MILENAGE_SIGN_PARAMS {
  CK_ULONG ulMilenageFlags;
  CK_ULONG ulEncKiLen;            /* not used - must be 0 */
  CK_BYTE_PTR pEncKi;             /* not used */
  CK_ULONG ulEncOPcLen;           /* not used - must be 0 */
  CK_BYTE_PTR pEncOPc;            /* not used */
  CK_OBJECT_HANDLE hSecondaryKey; /* CKK_NC_MILENAGE_OPC key handle */
  CK_OBJECT_HANDLE hRCKey;        /* optional CKK_NC_MILENAGE_RC key handle */
  CK_BYTE sqn[6];                 /* sequence number */
  CK_BYTE amf[2];                 /* authentication management field */
} CK_MILENAGE_SIGN_PARAMS;
```

ulMilenageFlags can consist of the following flags:

```
#define CKF_NC_MILENAGE_OPC                0x00000001  /* secondary key is OPC (not OP) */
#define CKF_NC_MILENAGE_OP_OBJECT          0x00000004  /* secondary key is supplied by object handle */
#define CKF_NC_MILENAGE_USER_DEFINED_RC    0x00000010  /* MilenageRC key is present (hRC) */
```

Both the `CKF_NC_MILENAGE_OPC` and `CKF_NC_MILENAGE_OP_OBJECT` flags must be present. The nShield PKCS #11 library currently only supports passing the OPC key handle to the mechanism.

If the `CKF_NC_MILENAGE_USER_DEFINED_RC` flag is set, `hRCKey` must point to a `CKK_NC_MILENAGE_RC` key object handle.

15.13.1. CKM_NC_MILENAGE

Computes the MILENAGE f1/f2/f3/f4/f5 functions as defined in ETSI TS 135 206 s4.1 and thus generates the Authentication Vector (AV) as defined in the ETSI Authentication and Key Agreement (AKA) protocol. This single output vector is the concatenated values RAND||XRES||CK||IK||XOR(SQN,AK)||AMF||MAC.

The following example shows how to use `CKM_NC_MILENAGE`:

```
/* session represents an existing open session */
CK_SESSION_HANDLE session;

/* subscriber_key, opc_key and rc_key represent existing keys */
CK_OBJECT_HANDLE subscriber_key, opc_key, rc_key;

/* sqn, amf and rand represent existing byte arrays holding the sequence number,
 * authentication management field and RAND challenge respectively
 * rand is optional */
CK_BYTE sqn[6], amf[2], rand[16];

CK_MILENAGE_SIGN_PARAMS milenage_params;
milenage_params.ulMilenageFlags = CKF_NC_MILENAGE_OP_OBJECT | CKF_NC_MILENAGE_OPC;
milenage_params.hSecondaryKey = opc_key;
memcpy(&(milenage_params.sqn), sqn, 6);
```

```
memcpy(&(milenage_params.amf), amf, 2);

/* a user-defined RC key is optional */
milenage_params.ulMilenageFlags |= CKF_NC_MILENAGE_USER_DEFINED_RC;
milenage_params.hRCKey = rc_key;

CK_MECHANISM milenage_mech = {CKM_NC_MILENAGE, &milenage_params, sizeof(milenage_params)};

/* Typical convention is to call C_Sign with the pData parameter set to
 * NULL to determine the required size of the buffer - see Section 5.2 of
 * the PKCS#11 Base Specification - but for brevity we allocate a 72 byte buffer
 * since CKM_NC_MILENAGE output length is constant. */

CK_RV rv;
CK_BYTE milenage_result[72] = {0};
CK_ULONG milenage_len = sizeof(milenage_result);
rv = C_SignInit(session, &milenage_mech, subscriber_key);
if (rv != CKR_OK) return rv;
rv = C_Sign(session, rand, 16, milenage_result, &milenage_len);
if (rv != CKR_OK) return rv;
```

The RAND value passed to C_Sign is optional and can be left as NULL. A user-defined RC key is also optional and can be omitted by removing the `CKF_NC_MILENAGE_USER_DEFINED_RC` flag and leaving hRCKey as NULL.

An RC key can be generated using `CKM_NC_MILENAGERC_KEY_GEN` or created using custom values with C_CreateObject (see Object management functions for details). If no RC key is supplied, the default values defined in ETSI TS 135 206 s4.1 will be used.

15.13.2. CKM_NC_MILENAGE_RESYNC

Performs part of the resynchronization procedure as described in the AKA protocol. This computes the MILENAGE f1*/f5* functions as defined in ETSI TS 135 206 s4.1 and verifies AUTS (i.e. XOR(SQN_UE,AK)||MAC-S). If successful, the mechanism returns the sequence number SQN_UE.

The calls to C_SignInit and C_Sign are the same as during authentication, except the second argument passed to C_Sign is the concatenated vector RAND||AUTS instead of RAND. The `sqn` value in the parameters structure for this mechanism is not required and will be ignored.

15.13.3. CKM_NC_MILENAGE_AUTS (testing only)

This mechanism is only for testing the resynchronization operation. It computes the MILENAGE f1*/f5* functions as defined in ETSI TS 135 206 s4.1 and returns RAND||AUTS (required as an input to `CKM_NC_MILENAGE_RESYNC`).

The calls to C_SignInit and C_Sign are the same as during authentication. The RAND value is

optional.

15.14. CKM_NC_TUAK_TOPC

Derive CKK_NC_TUAKTOPC key from CKK_NC_TUAKTOP and CKK_NC_TUAKSUBSCRIBER keys for use in the 3GPP mechanisms defined in ETSI TS 135 231 s6.1.

A C_DeriveKey function call is made. The function takes the CKK_NC_TUAKSUBSCRIBER key handle as the base key and the following structure as the mechanism parameter:

```
typedef struct CK_NC_TUAK_DERIVE_PARAMS {
  CK_OBJECT_HANDLE hTOPKey;   /* CKK_NC_TUAK_TOP key handle */
  CK_ULONG ulIterations;      /* number of Keccak iterations (1 or 2) */
} CK_NC_TUAK_DERIVE_PARAMS;
```

To generate the subscriber and TOP keys, use the corresponding vendor-defined key generation mechanisms (which do not take any mechanism parameters):

- CKM_NC_TUAKSUBSCRIBER_KEY_GEN
- CKM_NC_TUAKTOP_KEY_GEN

15.15. CKM_NC_TUAK, CKM_NC_TUAK_AUTS, CKM_NC_TUAK_RESYNC

3GPP mechanisms for 5G mobile networks as defined by ETSI TS 135 231. Used with C_SignInit and C_Sign function calls. The parameters for these mechanisms are defined in the following structure:

```
typedef struct CK_TUAK_SIGN_PARAMS {
  CK_ULONG             ulTuakFlags;
  CK_ULONG             ulEncKiLen;     /* not used - must be 0 */
  CK_BYTE_PTR          pEncKi;         /* not used */
  CK_ULONG             ulEncTOPcLen;   /* not used - must be 0 */
  CK_BYTE_PTR          pEncTOPc;       /* not used */
  CK_ULONG             ulIterations;   /* number of Keccak iterations (1 or 2) */
  CK_OBJECT_HANDLE     hSecondaryKey;  /* existing CKK_NC_TUAK_TOPC key handle */
  CK_ULONG             ulResLen;       /* length of expected response (4, 8, 16 or 32 bytes) */
  CK_ULONG             ulMacALen;      /* length of MAC (8, 16 or 32 bytes) */
  CK_ULONG             ulCkLen;        /* length of crypto key CK (16 or 32 bytes) */
  CK_ULONG             ulIkLen;        /* length of identity key IK (16 or 32 bytes) */
  CK_BYTE              sqn[6];         /* sequence number */
  CK_BYTE              amf[2];         /* authentication management field */
} CK_TUAK_SIGN_PARAMS;
```

The ulTuakFlags can consist of the following flags:

```
#define CKF_NC_TUAK_TOPC               0x00000001  /* secondary key is TOPC (not TOP) */
```

```
#define CKF_NC_TUAK_TOP_OBJECT          0x00000004  /* secondary key is supplied by object handle */
```

Both the CKF_NC_TUAK_TOPC and CKF_NC_TUAK_TOP_OBJECT flags must be present. The nShield PKCS #11 library currently only supports passing the TOPC key handle to the mechanism.

15.15.1. CKM_NC_TUAK

Computes the TUAK f1/f2/f3/f4/f5 functions as defined in ETSI TS 135 231 s6.2/s6.4 and thus generates the Authentication Vector (AV) as defined in the ETSI Authentication and Key Agreement (AKA) protocol. This single output vector is the concatenated values RAND||XRES||CK||IK||XOR(SQN,AK)||AMF||MAC.

The following example shows how to use CKM_NC_TUAK:

```
/* session represents an existing open session */
CK_SESSION_HANDLE session;

/* subscriber_key and topc_key represent existing keys */
CK_OBJECT_HANDLE subscriber_key, topc_key;

/* sqn, amf and rand represent existing byte arrays holding the sequence number,
 * authentication management field and RAND challenge respectively
 * rand is optional */
CK_BYTE sqn[6], amf[2], rand[16];

CK_TUAK_SIGN_PARAMS tuak_params;
tuak_params.ulTuakFlags = CKF_NC_TUAK_TOP_OBJECT | CKF_NC_TUAK_TOPC;
tuak_params.hSecondaryKey = topc_key;
tuak_params.ulIterations = 1;         // 1 or 2
tuak_params.ulResLen = 32;            // 4, 8, 16 or 32
tuak_params.ulMacALen = 32;           // 8, 16 or 32
tuak_params.ulCkLen = 32;             // 16 or 32
tuak_params.ulIkLen = 32;             // 16 or 32
memcpy(&(tuak_params.sqn), sqn, 6);
memcpy(&(tuak_params.amf), amf, 2);

CK_MECHANISM tuak_mech = {CKM_NC_TUAK, &tuak_params, sizeof(tuak_params)};

/* Typical convention is to call C_Sign with the pData parameter set to
 * NULL to determine the required size of the buffer - see Section 5.2 of
 * the PKCS#11 Base Specification - but for brevity we allocate a 1KB buffer */

CK_RV rv;
CK_BYTE tuak_result[1000] = {0};
CK_ULONG tuak_len = sizeof(tuak_result);
rv = C_SignInit(session, &tuak_mech, subscriber_key);
if (rv != CKR_OK) return rv;
rv = C_Sign(session, rand, 16, tuak_result, &tuak_len);
if (rv != CKR_OK) return rv;
```

The RAND value passed to C_Sign is optional and can be left as NULL.

15.15.2. CKM_NC_TUAK_RESYNC

Performs part of the resynchronization procedure as described in the AKA protocol. This computes the TUAK f1*/f5* functions as defined in ETSI TS 135 231 s6.3/s6.5 and verifies AUTS (i.e. XOR(SQN_UE,AK)||MAC-S). If successful, the mechanism returns the sequence number SQN_UE.

The calls to C_SignInit and C_Sign are the same as during authentication, except the second argument passed to C_Sign is the concatenated vector RAND||AUTS instead of RAND. The `sqn` value in the parameters structure for this mechanism is not required and will be ignored.

15.15.3. CKM_NC_TUAK_AUTS (testing only)

This mechanism is only for testing the resynchronization operation. It computes the TUAK f1*/f5* functions as defined in ETSI TS 135 231 s6.3/s6.5 and returns RAND||AUTS (required as an input to `CKM_NC_TUAK_RESYNC`).

The calls to C_SignInit and C_Sign are the same as during authentication. The RAND value is optional. Only the `sqn`, `amf`, `ulMacALen` and `ulIterations` parameters are required. The remainder will be ignored.

16. KISAAlgorithm mechanisms

If you are using version 1.20 or greater and you have enabled the KISAAlgorithms feature, you can use the following mechanisms through the standard PKCS #11 API calls.

16.1. KCDSA keys

The CKM_KCDSA mechanism is a plain general signing mechanism that allows you to use a CKK_KCDSA key with any length of plain text or pre-hashed message. It can be used with the standard single and multipart C_Sign and C_Verify update functions.

The CKM_KCDSA mechanism takes a CK_KCDSA_PARAMS structure that states which hashing mechanism to use and whether or not the hashing has already been performed:

```
typedef struct CK_KCDSA_PARAMS {
        CK_MECHANISM_PTR digestMechanism;
        CK_BBOOL dataIsHashed;
}
```

The following digest mechanisms are available for use with the digestMechanism:

- CKM_SHA_1
- CKM_HAS160
- CKM_RIPEMD160

The dataIsHashed flag can be set to one of the following values:

- 1 when the message has been pre-hashed (pre-digested)
- 0 when the message is in plain text.

The CK_KCDSA_PARAMS structure is then passed in to the mechanism structure.

16.2. Pre-hashing

If you want to provide a pre-hashed message to the C_Sign() or C_Verify() functions using the CKM_KCDSA mechanism, the hash must be the value of $h(z||m)$ where:

- h is the hash function defined by the mechanism
- z is the bottom 512 bits of the public key, with the most significant byte first
- m is the message that is to be signed or verified.

The hash consists of the bottom 512 bits of the public key (most significant byte first), with

the message added after this.

If the hash is not formatted as described when signing, then incorrect signatures are generated. If the hash is not formatted as described when verifying, then invalid signatures can be accepted and valid signatures can be rejected.

16.3. CKM_KCDSA_SHA1, CKM_KCDSA_HAS160, CKM_KCDSA_RIPEMD160

These older mechanisms sign and verify using a CKK_KCDSA key. They now work with the C_Sign and C_Update functions, though they do not take the CK_KCDSA_PARAMS structure or pre-hashed messages. These mechanisms can be used for single or multipart signing and are not restricted as to message size.

16.4. CKM_KCDSA_KEY_PAIR_GEN

This mechanism generates a CKK_KCDSA key pair similar to that of DSA. You can supply in the template a discrete log group that consists of the CKA_PRIME, CKA_SUBPRIME, and CKA_BASE attributes. In addition, you must supply CKA_PRIME_BITS, with a value between 1024 and 2048, and CKA_SUBPRIME_BITS, which must have a value of 160. If you supply CKA_PRIME_BITS and CKA_SUBPRIME_BITS without a discrete log group, the module generates the group. CKR_TEMPLATE_INCOMPLETE is returned if CKA_PRIME_BITS and CKA_SUBPRIME_BITS are not supplied.

CKA_PRIME_BITS must have the same length as the prime and CKA_SUBPRIME-BITS must have the same length as the subprime if the discrete log group is also supplied. If either are different, PKCS #11 returns CKR_TEMPLATE_INCONSISTENT.

You can use the C_GenerateKeyPair function to generate a key pair. If you supply one or more parts of the discrete log group in the template, the PKCS #11 library assumes that you want to supply a specific discrete log group. CKR_TEMPLATE_INCOMPLETE is returned if not all parts are supplied. If you want the module to calculate a discrete log group for you, ensure that there are no discrete log group attributes present in the template.

A CKK_KCDSA private key has two value attributes, CKA_PUBLIC_VALUE and CKA_PRIVATE_VALUE. This is in contrast to DSA keys, where the private key has only the attribute CKA_VALUE, the private value. The public key in each case contains only the public value.

The standard key-pair attributes common to all key pairs apply. Their values are the same as those for DSA pairs unless specified differently in this section.

16.5. CKM_KCDSA_PARAMETER_GEN

For information about DOMAIN Objects, read the PKCS #11 specification v2.11.

Use this mechanism to create a `CKO_DOMAIN_PARAMETERS` object. This is referred to as a `KCDSAComm` key in the nCore interface.

Use `C_GenerateKey` to generate a new discrete log group and initialization values. The initialization values consist of a counter (`CKA_COUNTER`) and a hash (`CKA_SEED`) that is the same length as `CKA_PRIME_BITS`, which must have a value of 160. The `CKA_SEED` must be the same size as `CKA_SUBPRIME_BITS`. If this not the case, the PKCS #11 library returns `CKR_DOMAIN_PARAMS_INVALID`.

Optionally, you can supply the initialization values. If you supply the initialization values with `CKA_PRIME_BITS` and `CKA_SUBPRIME_BITS`, you can reproduce a discrete log group generated elsewhere. This allows you to verify that the discrete log group used in key pairs is correct. If the initialization values are not present in the template, a new discrete log group and corresponding initialization values are generated. These initialization values can be used to reproduce the discrete log group that has just been generated. The newly generated discrete log group can then be used in a PKCS #11 template to generate a `CKK_KCDSA` key using `C_Generate_Key_Pair`. `DOMAIN` keys can also be imported using the `C_CreateObject` call.

16.6. CKM_HAS160

`CKM_HAS160` is a basic hashing algorithm. The hashing is done on the host machine. This algorithm can be used by means of the standard digest function calls of the PKCS #11 API.

16.7. SEED secret keys

16.7.1. CKM_SEED_KEY_GEN

This mechanism generates a 128-bit SEED key. The standard secret key attributes are required, except that no length is required since this a fixed length key type similar to DES3. Normal return values apply when generating a `CKK_SEED` type key.

16.7.2. CKM_SEED_ECB, CKM_SEED_CBC, CKM_SEED_CBC_PAD

These mechanisms are the standard mechanisms to be used when encrypting and decrypting or wrapping with a CKK_SEED key. A CKK_SEED key can be used to wrap or unwrap both secret keys and private keys. A CKK_KCDSA key cannot be wrapped by any key type.

The CKM_SEED_ECB mechanism wraps only secret keys of exact multiples of the CKK_SEED block size (16) in ECB mode. The CKM_SEED_CBC_PAD key wraps the same keys in CBC mode.

The CKM_SEED_CBC_PAD key wraps keys of variable block size. It is the only mechanism available to wrap private keys.

A CKK_SEED key can be used to encrypt and decrypt with both single and multipart methods using the standard PKCS #11 API. The plain text size for multipart cryptographic function must be a multiple of the block size.

16.7.3. CKM_SEED_MAC, CKM_SEED_MAC_GENERAL

These mechanisms perform both signing and verification. They can be used with both single and multipart signing or verification using the standard PKCS #11 API. Message size does not matter for either single or multipart signing and verification.

17. Attributes

The following sections describe how PKCS #11 attributes map to the Access Control List (ACL) given to the key by the nCore API. nCore API ACLs are described in the *nCore API Documentation* (supplied as HTML).

17.1. CKA_SENSITIVE

In a FIPS 140 Level 2 world, `CKA_SENSITIVE=FALSE` creates a key with an ACL that includes `ExportAsPlain`. Keys are exported using `DeriveMech_EncryptMarshalled` even in a FIPS 140 Level 2 world. The presence of the `ExportAsPlain` permission makes the status of the key clear when a FIPS 140 Level 2 ACL is viewed using `GetACL`.

`CKA_SENSITIVE=FALSE` always creates a key with an ACL that includes `DeriveKey` with `DeriveRole_BaseKey` and `DeriveMech_EncryptMarshalled`.

See also `CKA_UNWRAP_TEMPLATE`.

17.2. CKA_PRIVATE

If `CKA_PRIVATE` is set to `TRUE`, keys are protected by the logical token of the OCS. If it is set to `FALSE`, public keys are protected by a well-known module key, and other keys and objects are protected by the Security World module key.

You must set `CKA_PRIVATE` to:

- `FALSE` for public keys
- `TRUE` for non-extractable keys on card slots.

17.3. CKA_EXTRACTABLE

`CKA_EXTRACTABLE` creates a key with an ACL including `DeriveKey` permissions listed in the following table:

Key Type	Role	Mechanism
Secret key	DeriveRole_BaseKey	DeriveMech_AESKeyWrap DeriveMech_RawEncrypt DeriveMech_RawEncryptZeroPad DeriveMech_ECIESKeyWrap
Private key	DeriveRole_BaseKey	DeriveMech_PKCS8Encrypt

17.4. CKA_ENCRYPT, CKA_DECRYPT, CKA_SIGN, CKA_VERIFY

These attributes create a key with ACL including Encrypt, Decrypt, Sign, or Verify permission.

17.5. CKA_WRAP, CKA_UNWRAP

CKA_WRAP creates a key with an ACL including the DeriveKey permissions listed in the following table:

Key Type	Role	Mechanism
Secret key	DeriveRole_WrapKey	DeriveMech_PKCS8Encrypt
Secret key (AES only)	DeriveRole_WrapKey	DeriveMech_AESKeyWrap
Secret key, public key (RSA only)	DeriveRole_WrapKey	DeriveMech_RawEncrypt DeriveMech_RawEncryptZeroPad
Public key (elliptic curve only)	DeriveRole_WrapKey	DeriveMech_ECIESKeyWrap

CKA_UNWRAP creates a key with an ACL including the DeriveKey permissions listed in the following table:

Key Type	Role	Mechanism
Secret key	DeriveRole_WrapKey	DeriveMech_PKCS8Decrypt DeriveMech_PKCS8DecryptEx
Secret key (AES only)	DeriveRole_WrapKey	DeriveMech_AESKeyUnwrap

Key Type	Role	Mechanism
Secret key, public key (RSA only)	DeriveRole_WrapKey	DeriveMech_RawDecrypt DeriveMech_RawDecryptZeroPad
Public key (elliptic curve only)	DeriveRole_WrapKey	DeriveMech_ECIESKeyUnwrap

17.6. CKA_WRAP_TEMPLATE, CKA_UNWRAP_TEMPLATE

CKA_WRAP_TEMPLATE and CKA_UNWRAP_TEMPLATE guard against non-compliance of keys by specifying an attribute template.

The CKA_WRAP_TEMPLATE attribute applies to wrapping keys and specifies the attribute template to match against any of the keys wrapped by the wrapping key. Keys which do not match the attribute template will not be wrapped.

The CKA_UNWRAP_TEMPLATE attribute applies to wrapping keys and specifies the attribute template to apply to any of the keys which are unwrapped by the wrapping key. Keys will not be unwrapped if there is attribute conflict between the CKA_UNWRAP_TEMPLATE and any user supplied template (pTemplate).

Nested occurrences of CKA_WRAP_TEMPLATE or CKA_UNWRAP_TEMPLATE are not supported.

If CKA_MODIFIABLE or CKA_SENSITIVE are defined within the CKA_UNWRAP_TEMPLATE, the behavior is as follows:

CKA_MODIFIABLE (TRUE)

PKCS #11 Attribute Types	Unwrap Template Attribute	C_Unwrap pTemplate Attribute	Attribute Value Comparison	Allowed
All supported	Defined	Defined	Equal	Yes
	Defined	Defined	Not Equal	Yes
	Undefined	Defined	N/A	Yes
	Defined	Undefined	N/A	Yes

CKA_MODIFIABLE (FALSE)

PKCS #11 Attribute Types	Unwrap Template Attribute	C_Unwrap pTemplate Attribute	Attribute Value Comparison	Allowed
All supported	Defined	Defined	Equal	Yes
	Defined	Defined	Not Equal	No
	Undefined	Defined	N/A	Yes
	Defined	Undefined	N/A	Yes

CKA_SENSITIVE (TRUE)

PKCS #11 Attribute Types	C_Unwrap pTemplate Attribute	C_Unwrap pTemplate Attribute Value	Allowed
CKA_SENSITIVE	Defined	FALSE	No
CKA_EXTRACTABLE	Defined	FALSE	No

CKA_SENSITIVE (FALSE)

PKCS #11 Attribute Types	C_Unwrap pTemplate Attribute	C_Unwrap pTemplate Attribute Value	Allowed
CKA_SENSITIVE	Defined	TRUE	Yes
		FALSE	Yes
CKA_EXTRACTABLE	Defined	TRUE	Yes
		FALSE	Yes

See also `CKA_ALLOWED_MECHANISMS` for more information about mechanism-specific restrictions applying to the use of `CKA_UNWRAP_TEMPLATE`.

17.7. CKA_SIGN_RECOVER

`C_SignRecover` checks `CKA_SIGN_RECOVER` but is otherwise identical to `C_Sign`. Setting `CKA_SIGN_RECOVER` creates a key with an ACL that includes `Sign` permission.

17.8. CKA_VERIFY_RECOVER

Setting `CKA_VERIFY_RECOVER` creates a public key with an ACL including `Encrypt` permission.

17.9. CKA_DERIVE

For Diffie-Hellman private keys, CKA_DERIVE creates a key with Decrypt permissions.

For secret keys, CKA_DERIVE creates a key with an ACL that includes DeriveRole_BaseKey with one of DeriveMech_DESsplitXOR, DeriveMech_DES2splitXOR, DeriveMech_DES3splitXOR, DeriveMech_RandsplitXOR, or DeriveMech_CASTsplitXOR as appropriate if the key is extractable, because this permission would effectively allow the key to be extracted. The ACL includes DeriveMech_RawEncrypt whether or not the key is extractable.

17.10. CKA_ALLOWED_MECHANISMS

CKA_ALLOWED_MECHANISMS is available as a full attribute array for all key types. The number of mechanisms in the array is the ulValueLen component of the attribute divided by the size of CK_MECHANISM_TYPE.

The CKA_ALLOWED_MECHANISMS attribute is set when generating, creating and unwrapping keys.

CKA_ALLOWED_MECHANISMS is an optional attribute and does not have to be set, except when the key is intended for use with one of the mechanisms described below. However, if CKA_ALLOWED_MECHANISMS is set, then the attribute is checked to see if the mechanism you want to use is in the list of allowed mechanisms. If the mechanism is not present, then an error occurs and a value of CKR_MECHANISM_INVALID is returned.

17.10.1. CKM_CONCATENATE_BASE_AND_KEY

You must set CKA_ALLOWED_MECHANISMS with the CKM_CONCATENATE_BASE_AND_KEY mechanism when generating or creating both of the keys that are used in the C_DeriveKey operation with the CKM_CONCATENATE_BASE_AND_KEY mechanism. If CKA_ALLOWED_MECHANISMS is not set at creation time then the correct ConcatenateBytes ACL is not set for the keys.

When CKM_CONCATENATE_BASE_AND_KEY is used with C_DeriveKey, CKA_ALLOWED_MECHANISMS is checked. If CKM_CONCATENATE_BASE_AND_KEY is not present, then an error occurs and a value of CKR_MECHANISM_INVALID is returned.

17.10.2. CKM_RSA_AES_KEY_WRAP

You must set CKA_ALLOWED_MECHANISMS with the CKM_RSA_AES_KEY_WRAP mechanism when generating or creating RSA keys that also have CKA_UNWRAP_TEMPLATE set on the private half if they are to be used in the C_UnwrapKey operation with the CKM_RSA_AES_KEY_WRAP mechanism.

When CKM_RSA_AES_KEY_WRAP is used with C_UnwrapKey, CKA_ALLOWED_MECHANISMS is checked. If CKM_RSA_AES_KEY_WRAP is not present but the unwrapping key has CKA_UNWRAP_TEMPLATE, then an error occurs and a value of CKR_MECHANISM_INVALID is returned.

RSA private keys that have CKA_ALLOWED_MECHANISMS set with the CKM_RSA_AES_KEY_WRAP mechanism cannot be copied if they also have both the following attributes set:

- CKA_TOKEN with a value of CK_TRUE
- CKA_UNWRAP_TEMPLATE

The C_CopyObject operation returns CKR_ACTION_PROHIBITED for such keys.

17.11. CKA_MODIFIABLE

CKA_MODIFIABLE only restricts access through the PKCS #11 API: all PKCS #11 keys have ACLs that include the ReduceACL permission.

See also CKA_UNWRAP_TEMPLATE.

17.12. CKA_TOKEN

Token objects are saved as key blobs. Session objects only ever exist on the module.

17.13. CKA_START_DATE, CKA_END_DATE

These attributes are ignored, and the PKCS #11 standard states that these attributes do not restrict key usage.

17.14. CKA_TRUSTED and CKA_WRAP_WITH_TRUSTED

CKA_TRUSTED and CKA_WRAP_WITH_TRUSTED guard against a key being wrapped and removed from the HSM by an untrusted wrapping key. A key with a CKA_WRAP_WITH_TRUSTED attribute can only be wrapped by a wrapping key with a CKA_TRUSTED attribute. A trusted key can only be given a CKA_TRUSTED attribute by the PKCS #11 Security officer.

The CKA_WRAP_WITH_TRUSTED attribute gives a key an ACL whose DeriveRole_BaseKey exists in a group protected by a certifier. The ACL therefore requires a certificate generated by the PKCS #11 Security Officer to be able to wrap the key.

The CKA_TRUSTED attribute stores on a wrapping key a certificate signed by the PKCS #11

Security Officer. This certificate can then be used to authenticate a wrapping operation.

CKA_TRUSTED can only be set if the session is logged in as CKU_SO, and the Security Officer's token and key has been preloaded. If not, the operation will return CKR_USER_NOT_LOGGED_IN.

CKA_WRAP_WITH_TRUSTED does not require the Security Officer token and key to be preloaded, or to be logged in as CKU_SO, but it does require that the role exists. If the role does not exist, the operation returns CKR_USER_NOT_LOGGED_IN. When attributes have been set, the PKCS #11 Security Officer is not needed for C_WrapKey to perform a trusted key wrapping.

If the PKCS #11 Security Officer is deleted, keys with existing CKA_TRUSTED or CKA_WRAP_WITH_TRUSTED attributes continue to be valid. If the PKCS #11 Security Officer is recreated, any new keys that are given the CKA_TRUSTED attribute will not be trusted by existing keys with CKA_WRAP_WITH_TRUSTED, and vice versa.

A CKO_CERTIFICATE object can also be given a CKA_TRUSTED attribute, and also requires the PKCS #11 Security Officer to do so. This includes using ckcerttool with the -T option, which sets CKA_TRUSTED to true.

17.15. CKA_COPYABLE and CKA_DESTROYABLE

The CKA_COPYABLE and CKA_DESTROYABLE attributes indicate whether an object can be copied using C_CopyObject or destroyed using C_DestroyObject. If the corresponding function is attempted when the attribute is set to false, the function returns CKR_ACTION_PROHIBITED.

CKA_COPYABLE and CKA_DESTROYABLE can be applied to objects through all interfaces that support setting attributes:

- C_GenerateKey and C_GenerateKeyPair
- C_CreateObject
- C_SetAttributeValue
- C_CopyObject

Existing and new objects have both attributes set to true by default. When changing an attribute, CKA_COPYABLE cannot be changed from false to true.

17.16. RSA key values

CKA_PRIVATE_EXPONENT is not used when importing an RSA private key using

`C_CreateObject`. However, it must be in the template, since the PKCS #11 standard requires it. All the other values are required.

The nCore API allows use of a default public exponent, but the PKCS #11 standard requires `CKA_PUBLIC_EXPONENT`.

Except for very small keys, the nShield default is 65537, which as a PKCS #11 big integer is `CK_BYTEpublic_exponent[ ] = { 1, 0, 1 };`

17.17. DSA key values

If `CKA_PRIME` is 1024 bits or less, then the `KeyType_DSAPrivate_GenParams_flags_Strict` flag is used, because it enforces a 1024 bit limit.

The implementation allows larger values of `CKA_PRIME`, but in those cases the `KeyType_DSAPrivate_GenParams_flags_Strict` flag is not used.

17.18. Vendor specific error codes

Security World Software defines the following vendor specific error codes:

`CKR_FIPS_TOKEN_NOT_PRESENT`

This error code indicates that an Operator Card is required even though the card slot is not in use.

`CKR_FIPS_MECHANISM_INVALID`

This error code indicates that the current mechanism is not allowed in FIPS 140 Level 3 mode.

`CKR_FIPS_FUNCTION_NOT_SUPPORTED`

This error code indicates that the function is not supported in FIPS 140 Level 3 mode (although it is supported in FIPS 140 Level 2 mode).

18. Utilities

This section describes command-line utilities Entrust provides as aids to developers.

18.1. ckdes3gen

```
ckdes3gen.exe [p|--pin-for-testing=<passphrase>] | [n|-nopin]
```

This utility is an example of Triple DES key generation using the nShield PKCS #11 library. The utility generates the DES3 key as a private object that can be used both to encrypt and decrypt.

By default, the utility prompts for a passphrase. You can supply a passphrase on the command line with the `--pin-for-testing` option, or suppress the passphrase request with the `--nopin` option. The passphrase is displayed in the clear on the command line, so this option is appropriate only for testing.

18.2. ckinfo

```
ckinfo.exe [r|--repeat-count=<COUNT>]
```

This utility displays `C_GetInfo`, `C_GetSlotInfo` and `C_GetTokenInfo` results. You can specify a number of repetitions of the command with `--repeat-count=<COUNT>`. The default is `1`.

18.3. cklist

```
cklist.exe [-p|--pin-for-testing=<passphrase>] [-n|-nopin]
```

This utility lists some details of objects on all slots. It lists public and private objects if invoked with a passphrase argument and public objects only if invoked without a passphrase argument.

It does not output any potentially sensitive attributes, even if the object has `CKA_SENSITIVE` set to `FALSE`.

By default, the utility prompts for a passphrase. You can supply a passphrase on the command line with the `--pin-for-testing option`, or suppress the passphrase request with the `--nopin` option. The passphrase is displayed in the clear on the command line, so this option is appropriate only for testing.

18.4. ckmechinfo

```
ckmechinfo.exe
```

The utility displays `C_GetMechanismInfo` results for each mechanism returned by `C_GetMechanismList`.

18.5. ckrsagen

```
ckrsagen.exe [-p|--pin-for-testing=<passphrase>] | [-n|-nopin]
```

The `ckrsagen` utility is an example of RSA key pair generation using the nShield PKCS #11 library. This is intended as a programmer's example only and not for general use. Use the key generation routines within your PKCS #11 application.

By default, the utility prompts for a passphrase. You can supply a passphrase on the command line with the `--pin-for-testing` option, or suppress the passphrase request with the `--nopin` option. The passphrase is displayed in the clear on the command line, so this option is appropriate only for testing.

18.6. cksotool

```
cksotool.exe [-h] [--version] [-m MODULE] [-c | -p | -i | --delete]
```

The `cksotool` utility can be used to create and manage the PKCS #11 Security Officer (SO). The SO consists of a token and an RSA key, and is necessary to be able to perform any operations that require a Security Officer as defined by the PKCS #11 specification. The utility can be used to view the current state of the SO using the `-i` or `--info` option, which provides details of the existence and validity of the underlying token and key.

The key and softcard created by `cksotool` is for Entrust internal use inside the PKCS #11 library. It is not to be used directly in an application.

19. Functions

The following sections list the PKCS #11 functions supported by the nShield PKCS #11 library. For a list of supported mechanisms, see Mechanisms.

Certain functions are included in PKCS #11 version 2.40 for compatibility with earlier versions only.

20. General purpose functions

The following functions perform as described in the PKCS #11 specification:

20.1. C_Finalize

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_Finalize`	tbc	Without modifications	2.40

20.1.1. Notes

The `CKNFAST_NONREMOVABLE` environment variable is only available for persistent tokens. When the variable is set, the rules for recognizing new cards are overridden, and the only way to invoke a new token is to call `C_Finalize` or `C_Initialize`.

20.2. C_GetInfo

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_GetInfo`	tbc	Without modifications	2.40

20.3. C_GetFunctionList

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_GetFunctionList`	tbc	Without modifications	2.40

20.4. C_Initialize

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_Initialize`	Yes	Without modifications	2.40

20.4.1. Notes

The CKNFAST_NONREMOVABLE environment variable is only available for persistent tokens. When the variable is set, the rules for recognizing new cards are overridden, and the only way to invoke a new token is to call C_Finalize or C_Initialize.

If your application uses multiple threads, you must supply such functions as CreateMutex (as stated in the PKCS #11 specification) in the CK_C_INITIALIZE_ARGS argument.

21. Slot and token management functions

The following functions perform as described in the PKCS #11 specification:

21.1. C_GetSlotInfo

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_GetSlotInfo	tbc	Without modifications	2.40

21.2. C_GetTokenInfo

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_GetTokenInfo	tbc	Without modifications	2.40

21.3. C_GetMechanismList

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_GetMechanismList	tbc	Without modifications	2.40

21.4. C_GetMechanismInfo

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_GetMechanismInfo	tbc	Without modifications	2.40

21.5. C_GetSlotList

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_GetSlotList	tbc	Without modifications	2.40

21.5.1. Notes

This function returns an array of PKCS #11 slots. Within each module, the slots are in the order:

1. module(s)
2. smart card reader(s)
3. software tokens, if present.

Each module is listed in ascending order by nShield `ModuleID`.

`C_GetSlotList` returns an array of handles. You cannot make any assumptions about the values of these handles. In particular, these handles are not equivalent to the slot numbers returned by the nCore API command `GetSlotList`.

21.6. C_InitToken

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_InitToken`	tbc	Without modifications	2.40

21.6.1. Notes

`C_InitToken` sets the card passphrase to the same value as the current token's passphrase and sets the `CKF_USER_PIN_INITIALIZED` flag.

This function is supported in load-sharing mode only when using softcards. To use `C_InitToken` in load-sharing mode, you must have created a softcard with the command `ppmk -n` before selecting the corresponding slot.

The `C_InitToken` function is *not* supported for use in non-load-sharing FIPS 140 Level 3 Security Worlds.

21.7. C_InitPIN

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_InitPin`	tbc	Without modifications	2.40

21.7.1. Notes

There is usually no need to call `C_InitPIN`, because `C_InitToken` sets the card passphrase.

Because the nShield PKCS #11 library can only maintain a single passphrase, `C_InitPIN` has the effect of changing the current token's passphrase.

This function is supported in load-sharing mode only when using softcards. To use `C_InitPIN` in load-sharing mode, you must have created a softcard with the command `ppmk -n` before selecting the corresponding slot.

21.8. C_SetPIN

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_SetPin`	tbc	Without modifications	2.40

21.8.1. Notes

The card passphrase may be any value.

Because the nShield PKCS #11 library can only maintain a single passphrase, `C_SetPIN` has the effect of changing the current token's passphrase or, if called in a Security Officer session, the card passphrase.

This function is supported in load-sharing mode only when using softcards. To use `C_SetPIN` in load-sharing mode, you must have created a Softcard with the command `ppmk -n` before selecting the corresponding slot.

22. Standard session management functions

These functions perform as described in the PKCS #11 specification:

22.1. C_OpenSession

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_OpenSession	tbc	Without modifications	2.40

22.2. C_CloseSession

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_CloseSession	tbc	Without modifications	2.40

22.3. C_CloseAllSessions

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_CloseAllSessions	tbc	Without modifications	2.40

22.4. C_GetOperationState

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_GetOperationState	tbc	Without modifications	2.40

22.5. C_SetOperationState

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_SetOperationState	tbc	Without modifications	2.40

22.6. C_Login

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_Login	tbc	Without modifications	2.40

22.7. C_Logout

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_Logout	tbc	Without modifications	2.40

23. nShield session management functions

The following are nShield-specific calls for *K/N* card set support:

23.1. C_LoginBegin

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_LoginBegin`	tbc	Without modifications	2.40

23.2. C_LoginNext

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_LoginNext`	tbc	Without modifications	2.40

23.3. C_LoginEnd

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_LoginEnd`	tbc	Without modifications	2.40

23.4. C_GetSessionInfo

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_GetSessionInfo`	tbc	Without modifications	2.40

23.5. nShield session management function notes

`ulDeviceError` returns the numeric value of the last status, other than `Status_OK`, returned by the module. This value is never cleared. Status values are enumerated in the header file `messages-args-en.h` on the nShield Developer's installation media. For descriptions of nShield status codes, see the *nCore API Documentation* (supplied as HTML).

24. Object management functions

These functions perform as described in the PKCS #11 specification:

24.1. C_CreateObject

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_CreateObject	tbc	Without modifications	2.40

24.1.1. CKK_NC_MILENAGERC

The MILENAGE mechanisms support providing a custom set of values for constants c1-c5 and r1-r5 as defined by ETSI TS 135 206 s4.1. A CKK_NC_MILENAGERC object must be created to store these custom values.

The key template passed to C_CreateObject in this case is a standard one for secret keys with either of the two following ways of providing the c and r values as attributes:

```
CK_BYTE cr_values[] = {
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* c1 */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* c2 */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* c3 */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* c4 */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* c5 */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00 /* r1, r2, r3, r4, r5 */
}

CK_ATTRIBUTE rc_template1[] = {
  /* default secret key attributes */
  {CKA_VALUE, &cr_values, sizeof(cr_values)}
}
```

```
CK_BYTE c1[] = {
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
}

CK_BYTE c2[] = {
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
}

CK_BYTE c3[] = {
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
```

```
}

CK_BYTE c4[] = {
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
}

CK_BYTE c5[] = {
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
}

CK_BYTE r1 = 0, r2 = 0, r3 = 0, r4 = 0, r5 = 0;

CK_ATTRIBUTE rc_template2[] = {
  /* default secret key attributes */
  {CKA_NC_MILENAGE_C1, &c1, sizeof(c1)},
  {CKA_NC_MILENAGE_C2, &c2, sizeof(c2)},
  {CKA_NC_MILENAGE_C3, &c3, sizeof(c3)},
  {CKA_NC_MILENAGE_C4, &c4, sizeof(c4)},
  {CKA_NC_MILENAGE_C5, &c5, sizeof(c5)},
  {CKA_NC_MILENAGE_R1, &r1, sizeof(r1)},
  {CKA_NC_MILENAGE_R2, &r2, sizeof(r2)},
  {CKA_NC_MILENAGE_R3, &r3, sizeof(r3)},
  {CKA_NC_MILENAGE_R4, &r4, sizeof(r4)},
  {CKA_NC_MILENAGE_R5, &r5, sizeof(r5)},
}
```

24.2. C_CopyObject

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_CopyObject	tbc	Without modifications	2.40

24.3. C_DestroyObject

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_DestroyObject	tbc	Without modifications	2.40

24.4. C_GetObjectSize

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_GetObjectSize	tbc	Without modifications	2.40

24.5. C_GetAttributeValue

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_GetAttributeValue	tbc	Without modifications	2.40

24.6. C_SetAttributeValue

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_SetAttributeValue	tbc	Without modifications	2.40

24.7. C_FindObjectsInit

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_FindObjectsInit	tbc	Without modifications	2.40

24.8. C_FindObjects

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_FindObjects	tbc	Without modifications	2.40

24.9. C_FindObjectsFinal

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_FindObjectsFinal	tbc	Without modifications	2.40

25. Encryption functions

These functions perform as described in the PKCS #11 specification:

25.1. C_EncryptInit

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_EncryptInit	tbc	Without modifications	2.40

25.2. C_Encrypt

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_Encrypt	tbc	Without modifications	2.40

25.3. C_EncryptUpdate

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_EncryptUpdate	tbc	Without modifications	2.40

25.4. C_EncryptFinal

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_EncryptFinal	tbc	Without modifications	2.40

26. Decryption functions

These functions perform as described in the PKCS #11 specification:

26.1. C_DecryptInit

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_DecryptInit	tbc	Without modifications	2.40

26.2. C_Decrypt

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_Decrypt	tbc	Without modifications	2.40

26.3. C_DecryptUpdate

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_DecryptUpdate	tbc	Without modifications	2.40

26.4. C_DecryptFinal

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_DecryptFinal	tbc	Without modifications	2.40

27. Message digesting functions

The following functions are performed on the host computer:

27.1. C_DigestInit

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_DigestInit	tbc	Without modifications	2.40

27.2. C_Digest

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_Digest	tbc	Without modifications	2.40

27.3. C_DigestUpdate

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_DigestUpdate	tbc	Without modifications	2.40

27.4. C_DigestFinal

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_DigestFinal	tbc	Without modifications	2.40

28. Signing and MACing functions

The following functions perform as described in the PKCS #11 specification:

28.1. C_SignInit

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_SignInit	tbc	Without modifications	2.40

28.2. C_Sign

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_Sign	tbc	Without modifications	2.40

28.3. C_SignRecoverInit

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_SignRecoverInit	tbc	Without modifications	2.40

28.4. C_SignRecover

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_SignRecover	tbc	Without modifications	2.40

28.5. C_SignUpdate

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_SignUpdate	tbc	Without modifications	2.40

28.5.1. Notes

This function is supported for:

- `CKM_SHA1_RSA_PKCS`
- `CKM_MD5_RSA_PKCS`

28.6. C_SignFinal

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_SignFinal`	tbc	Without modifications	2.40

28.6.1. Notes

This function is supported for:

- `CKM_SHA1_RSA_PKCS`
- `CKM_MD5_RSA_PKCS`

29. Functions for verifying signatures and MACs

The following functions perform as described in the PKCS #11 specification:

29.1. C_VerifyInit

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_VerifyInit	tbc	Without modifications	2.40

29.2. C_Verify

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_Verify	tbc	Without modifications	2.40

29.3. C_VerifyRecover

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_VerifyRecover	tbc	Without modifications	2.40

29.4. C_VerifyRecoverInit

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_VerifyRecoverInit	tbc	Without modifications	2.40

29.5. C_VerifyUpdate

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_VerifyUpdate	tbc	Without modifications	2.40

29.5.1. Notes

This function is supported for:

- `CKM_SHA1_RSA_PKCS`
- `CKM_MD5_RSA_PKCS`

29.6. C_VerifyFinal

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_VerifyFinal`	tbc	Without modifications	2.40

29.6.1. Notes

This function is supported for:

- `CKM_SHA1_RSA_PKCS`
- `CKM_MD5_RSA_PKCS`

30. Dual-purpose cryptographic functions

The following functions perform as described in the PKCS #11 specification:

30.1. C_DigestEncryptUpdate

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_DigestEncryptUpdate`	tbc	Without modifications	2.40

30.2. C_DecryptDigestUpdate

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_DecryptDigestUpdate`	tbc	Without modifications	2.40

30.3. C_SignEncryptUpdate

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_SignEncryptUpdate`	tbc	Without modifications	2.40

30.3.1. Notes

This function is supported for:

- `CKM_SHA1_RSA_PKCS`
- `CKM_MD5_RSA_PKCS`

30.4. C_DecryptVerifyUpdate

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_DecryptVerifyUpdate`	tbc	Without modifications	2.40

30.4.1. Notes

This function is supported for:

- `CKM_SHA1_RSA_PKCS`
- `CKM_MD5_RSA_PKCS`

31. Key-management functions

You can use the CKNFAST_OVERRIDE_SECURITY_ASSURANCES environment variable to modify the way that some functions, including key-management functions, are used.

You can set the CKNFAST_LOADSHARING environment variable to enable load sharing for the work allocation for key-management functions:

- When CKNFAST_LOADSHARING is not set, the first available module is selected.
- When CKNFAST_LOADSHARING is set, the work is shared between the available modules using a round-robin approach.

Module selection incurs additional overhead. Therefore, if the case load is light, load sharing might result in a small performance degradation. Most affected operations involve key creation, which which includes loading the keys on all modules when loadsharing is in use. For this reason, while there is an increase in throughput, it is not expected to be linear.

The vendor-defined boolean attribute CKA_NC_VALUE_ONLY is available for the C_DeriveKey function. It can only be used to derive a secret key with the following attribute settings:

- CKA_SENSITIVE set to FALSE
- CKA_TOKEN set to FALSE
- CKA_EXTRACTABLE set to TRUE

When CKA_NC_VALUE_ONLY is set to TRUE, it signals that the application intends only to extract the value of the derived key, via C_GetAttributeValue. The derived key will not be loadshared and is not guaranteed to be usable for other operations. If the derived key into which the key has been loaded becomes unavailable, the key will not be usable at all.

CKA_NC_VALUE_ONLY is defined in pkcs11extra.h in the nShield implementation of cryptoki.h.

CKA_NC_VALUE_ONLY provides a performance benefit even in the absence of loadsharing. However, its main benefit is in removing much of the loadsharing overhead and therefore in improving scalability.

31.1. C_GenerateKey

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_GenerateKey	tbc	Without modifications	2.40

31.2. C_GenerateKeyPair

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_GenerateKeyPair	tbc	Without modifications	2.40

31.3. C_WrapKey

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_WrapKey	tbc	Without modifications	2.40

31.4. C_UnwrapKey

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_UnwrapKey	tbc	Without modifications	2.40

31.5. C_DeriveKey

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_DeriveKey	tbc	Without modifications	2.40

32. Random number functions

The nShield module has an onboard, hardware random number generator to handle random number functions. Because it has an onboard random number generator, the nShield module does not use seed values.

32.1. C_GenerateRandom

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_GenerateRandom`	tbc	Without modifications	2.40

32.2. C_SeedRandom

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
`C_SeedRandom`	tbc	Without modifications	2.40

32.2.1. Notes

The `C_SeedRandom` function returns `CKR_RANDOM_SEED_NOT_SUPPORTED`.

33. Parallel function management functions

33.1. C_GetFunctionStatus

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_GetFunctionStatus	tbc	Without modifications	2.40

33.1.1. Notes

This function is supported in the approved fashion by returning the PKCS #11 status CKR_FUNCTION_NOT_PARALLEL.

33.2. C_CancelFunction

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_CancelFunction	tbc	Without modifications	2.40

33.2.1. Notes

This function is supported in the approved fashion by returning the PKCS #11 status CKR_FUNCTION_NOT_PARALLEL.

34. Callback functions

There are no vendor-defined callback functions. Surrender callback functions are never called.