THALES

www.thales-esecurity.com

Thales e-Security

Cryptographic API

Integration Guide

Version: 6.7

Date: 15 January 2016

Contents

Chapter 1: Introduction **9**

Read this guide if 9

Conventions 9

Typographical conventions 9

CLI command conventions 10

Model numbers 10

Security World Software 10

Document version numbers 12

Further information 12

Contacting Thales Support 12

Chapter 2: nShield architecture **13**

Security World Software modules 13

Security World Software server 13

Stubs and interface libraries 13

Using an interface library 14

Writing a custom application 14

Acceleration-only or key management 15

Chapter 3: PKCS #11 ..... **16**

PKCS #11 developer libraries 16

PKCS #11 security assurance mechanism 17

nCipher PKCS #11 library environment variables 17

CKNFAST_ASSUME_SINGLE_PROCESS 18

CKNFAST_ASSURANCE_LOG 19

CKNFAST_CARDSET_HASH 19

CKNFAST_DEBUG 19

CKNFAST_DEBUGDIR 19

CKNFAST_DEBUGFILE 20

CKNFAST_FAKE_ACCELERATOR_LOGIN 20

CKNFAST_LOADSHARING 20

CKNFAST_NO_ACCELERATOR_SLOTS 20

CKNFAST_NO_SYMMETRIC 20

CKNFAST_NO_UNWRAP 20

CKNFAST_NONREMOVABLE 21

CKNFAST_NVRAM_KEY_STORAGE 21

CKNFAST_OVERRIDE_SECURITY_ASSURANCES 21

CKNFAST_SEED_MAC_ZERO 25

CKNFAST_SESSION_THREADSAFE 26

CKNFAST_TOKENS_PERSISTENT 26

CKNFAST_USE_THREAD_UPCALLS 26

CKNFAST_LOAD_KEYS 26

CKNFAST_WRITE_PROTECTED 26

PKCS #11 with load-sharing 26

Logging in 27

Session objects 27

Module failure 27

Compatibility 28

Restrictions on function calls in load-sharing mode 28

PKCS #11 without load-sharing 28

K/N support for PKCS #11 29

Generating and deleting NVRAM-stored keys with PKCS #11 29

Generating NVRAM-stored keys 29

Deleting NVRAM-stored keys 30

nShield-specific PKCS #11 API extensions 30

C_LoginBegin 31

C_LoginNext 31

C_LoginEnd 31

Compiling and linking 32

Windows 32
Unix-based 32
Objects 34
Certificate Objects and Data Objects 34
Key Objects 35
Card pass phrases 35
Functions supported 35
General purpose functions 35
Slot and token management functions 36
Standard session management functions 37
nShield session management functions 37
Object management functions 37
Encryption functions 37
Decryption functions 38
Message digesting functions 38
Signing and MACing functions 38
Functions for verifying signatures and MACs 38
Dual-purpose cryptographic functions 39
Key-management functions 39
Random number functions 39
Parallel function management functions 39
Callback functions 39
Mechanisms 39
Vendor-defined mechanisms 44
Mechanisms for KISAAlgorithms 49
Attributes 52
Utilities 54
Chapter 4: CHIL 56
Structures the application must provide 56
Structures provided by hwcrhk 57

Multi-precision integers 57
Handling errors 58
HWCryptoHook_Init 59
HWCryptoHook_RandomBytes 62
HWCryptoHook_ModExp & HWCryptoHook_RSAImmedPub 63
HWCryptoHook_ModExpCRT and HWCryptoHook_RSAImmedPriv 64
HWCryptoHook_RSALoadKey 65
HWCryptoHook_RSA 66
HWCryptoHook_RSAUnloadKey 66
HWCryptoHook_RSAGetPublicKey 66
HWCryptoHook_DHLoadKey 67
HWCryptoHook_DH 67
HWCryptoHook_DHUnloadKey 68
HWCryptoHook_DHGetPublicKey 68
HWCryptoHook_DSALoadKey 69
HWCryptoHook_DSA 69
HWCryptoHook_DSAUnloadKey 70
HWCryptoHook_DSAGetPublicKey 70
HWCryptoHook_Finish 71
CHIL error logging 71
Example 73
Chapter 5: Microsoft CAPI CSP 75
Crypto API CSP 75
Supported algorithms 76
Symmetric algorithms 76
Asymmetric algorithms 76
Hash algorithms 77
Key generation and storage 77
User interface issues 78
Key counting 79

NVRAM-stored keys 79
CSP setup and utilities 80
Chapter 6: Microsoft CNG 82
CNG architecture overview 82
Supported algorithms for CNG 84
Signature interfaces (key signing) 84
Hashes 85
Asymmetric encryption 85
Symmetric encryption 85
Key exchange 86
Random Number Generation 86
Key authorization for CNG 86
Key use counting 88
Migrating keys for CNG 89
Importing a Microsoft CAPI key into the Security World Key Storage Provider 89
Importing a Microsoft CNG key into the Security World Key Storage Provider 90
Importing a Security World key into the Security World Key Storage Provider 91
Configuring the nCipher CNG CSP 92
Registering the CNG CSP 93
Unregistering or reregistering the CNG CSP 94
Uninstalling or reinstalling the CNG CSP 95
Utilities for CNG 96
cngimport 97
cnginstall 97
cngregister 98
cngsoak 99
ncsvcdep 99
cnglist 100
Chapter 7: nCipherKM JCA/JCE CSP 103
Installing the nCipherKM JCA/JCE CSP 104

Testing the nCipherKM JCA/JCE CSP installation 106
System properties 108
JCECSP_DEBUG property values 109
Compatibility 110
Architecture 111
Available functions 111
The KeyStore API 115
Initialization 115
Loading and storing keys 115
keytool 115
Using keys 116
Glossary 118
Internet addresses 125

Chapter 1: Introduction

This guide describes the following toolkits, supplied by Thales to help developers write applications that use nShield modules:

- nCipher PKCS #11 library
- Cryptographic Hardware Interface Library
- Microsoft CryptoAPI (MSCAPI)
- Microsoft Cryptography API: Next Generation (CNG)
- nCipherKM JCA/JCE cryptographic service provider.

These tool kits, like the application plug-ins supplied by Thales, use the Security World paradigm for key storage. For an introduction to Security Worlds, see the User Guide.

Read this guide if ...

Read this guide if you want to build an application that uses a Thales 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.

Conventions

Typographical conventions

Note: The word **Note** indicates important supplemental information.

If there is a danger of loss or exposure of key material (or any other security risk), this is indicated by a security triangle in the margin.

Keyboard keys that you must press are represented like this: `Enter`, `Ctrl-C`.

Examples of onscreen text from graphical user interfaces are represented by **boldface** text. Names of files, command-line utilities, and other system items are represented in `monospace` text. Variable text that you either see onscreen or that you must enter is represented in *italic*.

Examples of onscreen terminal display, both of data returned and of your input, are represented in a form similar to the following:

```
install
```

CLI command conventions

The basic syntax for a `CLI` command is:

```
command object <object_name> [parameter] [option] [modifier]
```

In this syntax, user-defined values are shown in *italics* and enclosed within the **< >** characters. Optional elements are shown enclosed within the **[ ]** characters. Mutually exclusive elements are separated by the **|** character.

Many system objects require the inclusion of a user-defined keyword value. For example, the **user** object is executed against a user-supplied *user_name*. Throughout this guide, all user-defined keyword values are shown in *italics*.

Each `CLI` command that you run performs an operation against the internal configuration of the appliance. The specific type of operation is specified by the first user-defined keyword value in the command string.

Model numbers

Model numbering conventions are used to distinguish different Thales hardware security devices. In the table below, *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+.
nC2021E-000, NCE2023E-000	An nToken (PCI express interface).
nC3*nnn*E-*nnn*, nC4*nnn*E-*nnn*	Thales nShield Solo HSM with a PCI Express (PCIe) interface.
nC30*nn*U-10, nC40*nn*U-10.	An nShield Edge module.

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 2008 or later	Unix-based
nShield Installation	`NFAST_HOME`	32 bit: `C:\Program Files\nCipher\nfast` 64 bit: `C:\Program Files (x86) \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/`

Note: 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.

Note: Dynamic feature certificates must be stored in the directory stated above. 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.

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 with percent signs (for example, *%NFAST_KMDATA%*)
- for Unix-based systems, absolute paths (for example, `/opt/nfast/kmdata/`).

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 Unix-based systems, 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.

Note: With previous versions of Security World Software for Windows platforms, the Key Management Data directory was located by default in `C:\nfast\kmdata`, the Feature

Certificates directory was located by default in `C:\nfast\fem`, and the Log Files directory was located by default in `C:\nfast\log`.

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.

Document version numbers

The version number of this document is shown on the copyright page of this guide. Quote the version number and the date on the copyright page if you need to contact Support about this document.

Further information

This guide forms one part of the information and support provided by Thales. You can find additional documentation in the `document` directory of the installation media for your product.

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`
- Unix-based:
  - 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.

Release notes containing the latest information about your product are available in the `release` directory of your installation media.

Note: We strongly recommend familiarizing yourself with the information provided in the release notes before using any hardware and software related to your product.

If you would like to receive security advisories from Thales, you can subscribe to the low volume security-announce mailing list by emailing the word **`subscribe`** in the message body to **nShield-securityadvisories@thales-esecurity.com**.

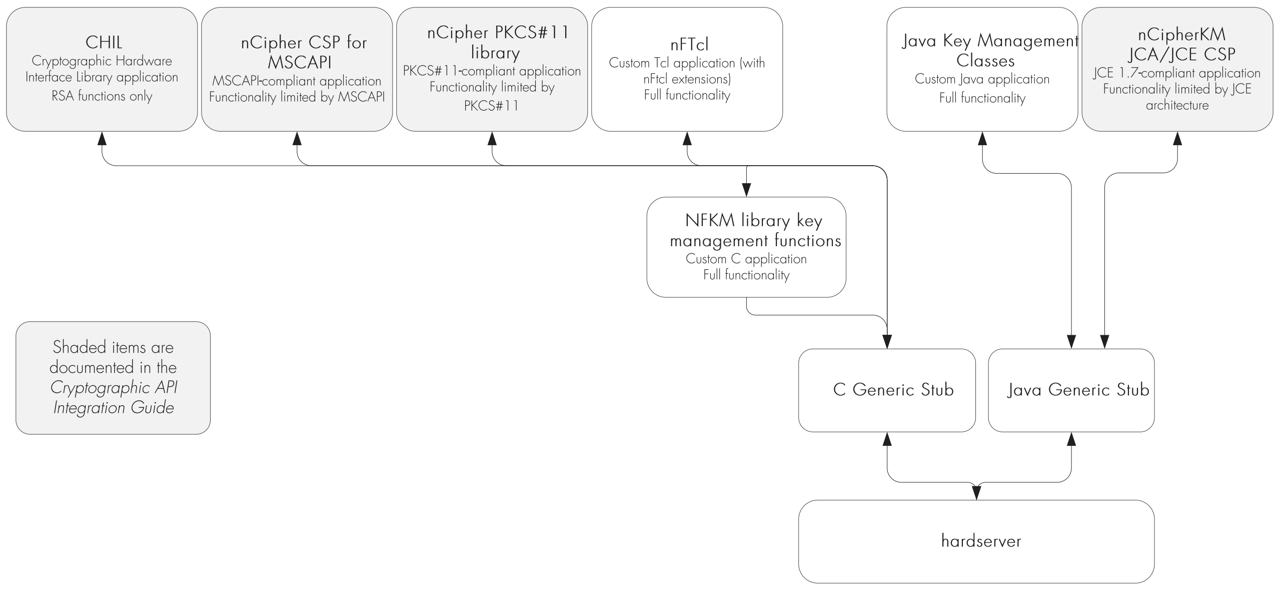
Contacting Thales Support

To obtain support for your product, visit: http://www.thales-esecurity.com/support-landing-page.

Chapter 2: nShield architecture

This chapter provides a brief overview of the Security World Software architecture. For a visual representation of nShield architecture and the documentation that relates to it, see Figure 1.

Figure 1. nShield Architecture


CHIL
Cryptographic Hardware Interface Library application
RSA functions only
nCipher CSP for MSCAPI
MSCAPI-compliant application
Functionality limited by MSCAPI
nCipher PKCS#11 library
PKCS#11-compliant application
Functionality limited by PKCS#11
nFTcl
Custom Tcl application (with nFtcl extensions)
Full functionality
Java Key Management Classes
Custom Java application
Full functionality
nCipherKM JCA/JCE CSP
JCE 1.7-compliant application
Functionality limited by JCE architecture
NFKM library key management functions
Custom C application
Full functionality
Shaded items are documented in the *Cryptographic API Integration Guide*
C Generic Stub
Java Generic Stub
hardserver

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).

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 Unix domain socket on Unix 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.

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 Thales 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.

Using an interface library

Thales supplies the following interface libraries:

- Cryptographic Hardware Interface Library
- Microsoft CryptoAPI
- 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 Thales cannot offer them in the interface library.

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, Thales provides generic stub libraries for C and Java, as well as a set of Tcl extensions that call the C generic stub library. If you want to use a language other than C, Java, or Tcl, you must write your own wrapper functions in your chosen programming language that call the C generic stub functions.

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

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 nShield Cryptographic Hardware Interface Library, 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.

Chapter 3: PKCS #11

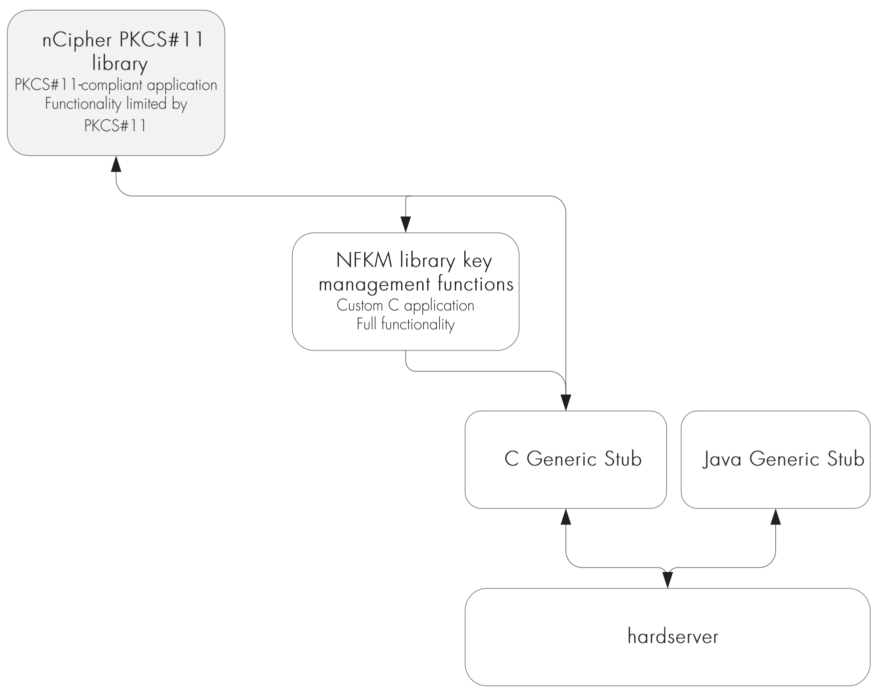
This chapter is intended for application developers who are writing PKCS #11 applications.

For an introduction to the PKCS #11 user library, including information about the environment variables and utilities available with common applications (for example, iPlanet), see the User Guide.

Before using the nCipher PKCS #11 libraries, we recommend that you read the *PKCS #11: Cryptographic Token Interface Standard*, version 2.01, published by RSA Laboratories.

For an illustration of the way that an nCipher PKCS #11 library works with the nShield APIs, see Figure 2.

Figure 2. nCipher PKCS #11 architecture



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

PKCS #11 developer libraries

The nCipher PKCS #11 libraries, `cknfast.lib` (`libcknfast.a` on Unix-based systems) are provided so that you can integrate your PKCS #11 applications with the nShield hardware security modules.

The nCipher PKCS #11 libraries:

- provide the PKCS #11 mechanisms listed in *Mechanisms* on page 39
- help you to identify potential security weaknesses, enabling you to create secure PKCS #11 applications more easily.

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 nCipher 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 nCipher 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 nCipher 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, see *CKNFAST_OVERRIDE_SECURITY_ASSURANCES* on page 21.

Note: 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 nCipher 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.

For more information about the environment variable and its parameters, see *CKNFAST_OVERRIDE_SECURITY_ASSURANCES* on page 21.

For more information about diagnostic warnings, see *Diagnostic warnings about questionable operations* on page 25.

nCipher PKCS #11 library environment variables

The nCipher PKCS #11 library uses the following environment variables:

- `CKNFAST_ASSUME_SINGLE_PROCESS`
- `CKNFAST_ASSURANCE_LOG`
- `CKNFAST_CARDSET_HASH`
- `CKNFAST_DEBUG`
- `CKNFAST_DEBUGDIR`
- `CKNFAST_DEBUGFILE`
- `CKNFAST_FAKE_ACCELERATOR_LOGIN`
- `CKNFAST_LOADSHARING`
- `CKNFAST_NO_ACCELERATOR_SLOTS`

- **`CKNFAST_NO_SYMMETRIC`**
- **`CKNFAST_NO_UNWRAP`**
- **`CKNFAST_NONREMOVABLE`**
- **`CKNFAST_NVRAM_KEY_STORAGE`**
- **`CKNFAST_OVERRIDE_SECURITY_ASSURANCES`**
- **`CKNFAST_SEED_MAC_ZERO`**
- **`CKNFAST_SESSION_THREADSAFE`**
- **`CKNFAST_TOKENS_PERSISTENT`**
- **`CKNFAST_USE_THREAD_UPCALLS`**
- **`CKNFAST_LOAD_KEYS`**
- **`CKNFAST_WRITE_PROTECTED`**

If you used the default values in the installation script, you should not need to change any of these environment variables.

You can set environment variables in the file **`cknfastrc`**. On Unix-based systems, this file is either in the **`/opt/nfast/`** or the **`/opt/nfast/bin/`** directory. On Windows, if the **`NFAST_HOME`** environment variable is not set, or if environment variables are cleared by your application, the file **`cknfastrc`** is in *NFAST_HOME*.

Each line of the file **`cknfastrc`** must be of the following form:

```
variable=value
```

Note: Variables set in the environment are used in preference to those set in the resource file.

Changing the values of these variables after you start your application has no effect until you restart the application.

If the description of a variable does not explicitly state what values you can set, the values you set are normally **`1`** or **`0`**, **`Y`** or **`N`**.

Note: For more information concerning Security World Software environment variables that are not specific to PKCS #11 and which are used to configure the behavior of your nShield installation, see the Security World Software installation instructions.

CKNFAST_ASSUME_SINGLE_PROCESS

By default, this variable is set to **`1`**. This specifies that only token objects that are loaded at the time **`C_Initialize`** is called are visible.

Setting this variable to **`0`** means that token objects created in one process become visible in another process when it calls **`C_FindObjects`**. Existing objects are also checked for modification on disc; if the key file has been modified, then the key is reloaded. Calling **`C_SetAttributeValues`** or **`C_GetAttributeValues`** also checks whether the object to be changed has been modified in another process and reloads it to ensure the most recent copy is changed.

Setting the variable to **`0`** can slow the library down because of the additional checking needed if a large number of keys are being changed and a large number of existing objects must be reloaded.

CKNFAST_ASSURANCE_LOG

This variable is used to direct all warnings from the Security Assurance Mechanism to a specific log file. For more information, see *Diagnostic warnings about questionable operations* on page 25.

You can specify the file name with an entry of the format:

```
CKNFAST_ASSURANCE_LOG=mylogfile.txt
```

CKNFAST_CARDSET_HASH

This variable enables you to specify a specific card set to be used in load-sharing mode. If this variable is set, only the virtual smart card slot that matches the specified hash is present (plus the accelerator slot). The hash that you use to identify the card set in **`CKNFAST_CARDSET_HASH`** is the SHA-1 hash of the secret on the card. Use the **`nfkminfo`** command-line utility to identify this hash for the card set that you want to use: it is listed as **`hkltu`**. For more information about using **`nfkminfo`**, see the *User Guide*.

CKNFAST_DEBUG

This variable is set to enable PKCS #11 debugging. The values you can set are in the range **`0`** - **`11`**. If you are using **`NFLOG_*`** for debugging, you must set **`CKNFAST_DEBUG`** to **`1`**. For more information about using **`NFLOG_*`**, see the *User Guide*.

Value	Description
`0`	None (default setting)
`1`	Fatal error
`2`	General error
`3`	Fix-up error
`4`	Warnings
`5`	Application errors
`6`	Assumptions made by the nCipher PKCS #11 library
`7`	API function calls
`8`	API return values
`9`	API function argument values
`10`	Details
`11`	Mutex locking detail

CKNFAST_DEBUGDIR

If this variable is set to the name of a writeable directory, log files are written to the specified directory. The name of each log file contains a process ID. This can make debugging easier for applications that fork a lot of child processes.

CKNFAST_DEBUGFILE

You can use this variable to write the output for **`CKNFAST_DEBUG`** (**`Path name > file name`**).

CKNFAST_FAKE_ACCELERATOR_LOGIN

If this variable is set, the nCipher PKCS #11 library accepts a PIN for a module-protected key, as required by Sun Java Enterprise System (JES), but then discards it. This means that a Sun JES user requesting a certificate protected by a load-shared module can enter an arbitrary PIN and obtain the certificate.

CKNFAST_LOADSHARING

Load-sharing is determined by the state of the **`CKNFAST_LOADSHARING`** environment variable.

To enable load-sharing mode, set the environment variable **`CKNFAST_LOADSHARING`** to a value that starts with something other than **`0`**, **`N`**, or **`n`**. The virtual slot behavior then operates. When this variable is not set (or is set to a value that starts with **`0`**, **`N`**, or **`n`**), you see two slots for every module connected.

Applications that enable the user to select the slot, or that dynamically select a slot based on its capabilities, normally work without requiring the backward capability mode.

For more information about virtual slot behavior, see your *User Guide*.

For more information about load-sharing, see *PKCS #11 with load-sharing* on page 26.

Note: To use softcards with PKCS #11, you must have **`CKNFAST_LOADSHARING`** set to a nonzero value. When using pre-loaded softcards or other objects, the PKCS #11 library automatically sets **`CKNFAST_LOADSHARING=1`** (load-sharing mode on) unless it has been explicitly set to **`0`** (load-sharing mode off).

CKNFAST_NO_ACCELERATOR_SLOTS

If this variable is set, the nCipher PKCS #11 library does not create the accelerator slot, and thus the library only presents the smart card slots (real or virtual, depending on whether load-sharing is in use).

Do not set this environment variable if you want to use the accelerator slot to create or load module protected keys.

Note: Setting this environment variable has no effect on **`ckcheckinst`** because **`ckcheckinst`** needs to list accelerator slots.

CKNFAST_NO_SYMMETRIC

If this variable is set, the nCipher PKCS #11 library does not advertise any symmetric key operations.

CKNFAST_NO_UNWRAP

If this variable is set, the nCipher PKCS #11 library does not advertise the **`c_wrap`** and **`c_unwrap`** commands. You should set this variable if you are using Sun Java Enterprise System (JES) or Netscape Certificate Management Server as it ensures that a standard SSL handshake is carried out. If this variable is not set, Sun JES or Netscape Certificate Management Server make extra calls, which reduces the speed of the library.

CKNFAST_NONREMOVABLE

When this environment variable is set, the state changes of the inserted card set are ignored by the nCipher PKCS #11 library.

Note: Since protection by non-persistent cards is enforced by the HSM, not the library, this variable does not make it possible to use keys after a non-persistent card is removed, or after a timeout expires.

CKNFAST_NVRAM_KEY_STORAGE

When this environment variable is set, the PKCS #11 library generates only keys in nonvolatile memory (NVRAM). You must also ensure this environment variable is set in order to delete NVRAM-stored keys. For more information, see *Generating and deleting NVRAM-stored keys with PKCS #11 on page 29*.

CKNFAST_OVERRIDE_SECURITY_ASSURANCES

This variable can be assigned one or more of the following parameters, with an associated value where appropriate, to override the specified security assurances in key operations where this is deemed acceptable:

- `all`
- `none`
- `tokenkeys`
- `longterm [=`*days*`]`
- `explicitness`
- `import`
- `unwrap_mech`
- `unwrap_kek`
- `derive_kek`
- `derive_xor`
- `derive_concatenate`
- `weak_`*algorithm*
- `shortkey_`*algorithm*=*bitlength*
- `silent`.

Each parameter specified is separated by a semicolon. On the command line, enter the following to set the variable:

```
CKNFAST_OVERRIDE_SECURITY_ASSURANCES="token1;token2=value3"
```

In the configuration file, enter the following to set the variable:

```
CKNFAST_OVERRIDE_SECURITY_ASSURANCES=token1;token2=value3
```

Unknown parameters generate a warning; see *Diagnostic warnings about questionable operations* on page 25.

The meaning of these parameters is described in the rest of this section.

all

The `all` parameter overrides all security checks and has the same effect as supplying all the other `CKNFAST_OVERRIDE_SECURITY_ASSURANCES` parameters except the `none` parameter. Using the `all` parameter prevents the library from performing any of the security checks and allows the library to perform potentially insecure operations. This parameter cannot be used with any other parameters.

none

The `none` parameter does not override any of the security checks and has the same effect as supplying no parameters. Using the `none` parameter allows the library to perform all security checks and warn about potentially insecure operations without performing them. This parameter cannot be used with any other parameters.

tokenkeys

The `tokenkeys` parameter permits applications to request that insecure keys are stored long-term by the cryptographic hardware and library.

Some PKCS #11 applications create short-term session keys as long-term objects in the cryptographic provider, for which strong protection by the module is not important. Therefore, provided that you intend to create long-term keys, the need to set this token does not always indicate a potential problem because the `longterm` keys restriction is triggered automatically. If you set the `tokenkeys` parameter, ensure that your Quality Assurance process tests all of your installation's functionality at least 48 hours after the system was set up to check that the key lifetimes are as expected.

When the `tokenkeys` parameter is set, the effect on the PKCS #11 library is to permit insecure Token keys. By default, any attempts to create, generate, or unwrap insecure keys with `CKA_TOKEN=true` fails with `CKR_TEMPLATE_INCONSISTENT` and a log message that explains the insecurity. When tokenkeys is included as a parameter for `CKNFAST_OVERRIDE_SECURITY_ASSURANCES`, attempts to create, generate, or unwrap insecure keys with `CKA_TOKEN=true` are allowed.

longterm[=*days*]

The `longterm` parameter permits an insecure key to be used for *days* after it was created. Usually insecure keys may not be used more than 48 hours after their creation. If *days* is not specified, there is no time limit.

Note: A need to set this variable usually means that some important keys that should be protected by the module's security are not secure.

When the `longterm` parameter is set, the PKCS #11 API permits the use of the following functions with an insecure key up to the specified number of *days* after its creation:

- `C_Sign` and `C_SignUpdate`
- `C_Verify` and `C_VerifyUpdate`

- **C_Encrypt** and **C_EncryptUpdate**
- **C_Decrypt** and **C_DecryptUpdate**.

By default these functions fail with **CKR_FUNCTION_FAILED**, or **CKR_KEY_FUNCTION_NOT_PERMITTED**, and a log message that explains the insecurity of these functions when used with an insecure private or secret key more than 48 hours after the creation of the key as indicated by **time()** on the host.

When the **longterm** parameter is set, the functions **C_SignInit**, **C_VerifyInit**, **C_EncryptInit**, and **C_DecryptInit** check the **CKA_CREATION_DATE** against the current time.

explicitness

The **explicitness** parameter permits applications to create insecure keys without explicitly recognizing that they are insecure by setting the flag which allows export as plain text. An insecure key is one whose plain text is available to an attacker on the host; thus it makes no sense to restrict legitimate users' access to the plain text of the key value.

Note: A need to set the **explicitness** parameter does not necessarily indicate a problem, but does usually indicate that a review of the application's security policies and use of the PKCS #11 API should be carried out.

Unless the **explicitness** parameter is set, attempts to create, generate, or unwrap insecure keys with **CKA_SENSITIVE=true**, or to set **CKA_SENSITIVE=true** on an existing key, fail by default with **CKR_TEMPLATE_INCONSISTENT** and a log message explaining the insecurity. However, when the **explicitness** parameter is set, these operations are allowed.

import

The **import** parameter allows keys that are to be imported into the module's protection from insecure external sources to be treated as secure, provided that the application requests security for them. Usually, the library treats imported keys as insecure for the purposes of checking the security policy of the application. Even though the imported copy may be secure, insecure copies of the key may still exist on the host and elsewhere.

If you are migrating from software storage to hardware protection of keys, you must enable the **import** parameter at the time of migration. You can disable **import** again after migrating the keys.

Note: Setting this variable at any other time indicates that the library regards the key as secure, even though it is not always kept within a secure environment.

When the **import** parameter is set, the PKCS #11 API treats keys that are imported through **C_CreateObject** or **C_UnwrapKey** as secure (provided there is no other reason to treat them as insecure). By default, keys which are imported through **C_CreateObject** or **C_UnwrapKey** without this option in effect are marked as being insecure. Only the setting of the parameter at the time of import is relevant.

unwrap_mech

The **unwrap_mech** parameter allows keys transferred into the module in an insecurely encrypted form to be treated as if the encryption had been secure. This parameter allows you to use key-decryption keys for insecure decryption mechanisms as well as for raw decryption.

There are no key decryption or wrapping mechanisms that are both secure and suitable for long keys. Set the **`unwrap_mech`** parameter to use PKCS #11 **`unwrap`** to create keys that are treated as secure. Set the **`unwrap_mech`** parameter at the time that the wrapping key is created or imported.

When the **`unwrap_mech`** parameter is set, the PKCS #11 API adds the **`CKA_DECRYPT`** permission on decryption, even if the template has **`CKA_DECRYPT=false`**. By default, trying to create a key with **`CKA_UNWRAP=true`** and **`CKA_DECRYPT=false`** fails with **`CKR_TEMPLATE_INCONSISTENT`**. If **`unwrap_mech`** is supplied as a parameter for **`CKNFAST_OVERRIDE_SECURITY_ASSURANCES`**, then when the **`CKA_UNWRAP`** permission is requested on a key, the library automatically adds the **`CKA_DECRYPT`** permission, even if the template has **`CKA_DECRYPT`** false, because abuse of the decryption mechanisms would allow a program to use the library to decrypt with the key.

unwrap_kek

When a key is transferred into the module in encrypted form, the key is usually treated as insecure unless the key that was used for the decryption only allows the import and export of keys and not the decryption of arbitrary messages. This behavior is necessary to prevent an unauthorized application from simply decrypting the encrypted key instead of importing it. However, because PKCS #11 wrapping mechanisms are insecure, all unwrapping keys have **`CKA_DECRYPT=true`**.

By default, keys that are unwrapped with a key that has **`CKA_DECRYPT`** permission are considered insecure. When the **`unwrap_kek`** parameter is set, the PKCS #11 API considers keys that are unwrapped with a key that also has **`CKA_DECRYPT`** permission as secure (provided there is no other reason to treat them as insecure).

derive_kek

By default, keys that have been derived by using **`CKM_DES3_ECB_ENCRYPT_DATA`** with a key that has **`CKA_ENCRYPT`** permission are considered insecure. However, when the **`derive_kek`** parameter is set, the PKCS #11 API considers keys that are derived with a key that has **`CKA_ENCRYPT`** permission as secure (provided that there is no other reason to treat them as insecure).

derive_xor

Normally, you can only use only extractable keys with **`CKM_XOR_BASE_AND_DATA`** and, on unextractable keys, only **`CKM_DES3_ECB_ENCRYPT_DATA`** is allowed by **`CKA_DERIVE`**. However, when the **`derive_xor`** parameter is set, the PKCS #11 API also allows such functions with keys that are not extractable and treats them as secure (provided that there is no other reason to treat them as insecure).

derive_concatenate

Normally, you can only use session keys with **`CKM_CONCATENATE_BASE_AND_KEY`** for use with the operation **`C_DeriveKey`**. However, when the **`derive_concatenate`** parameter is set, the PKCS#11 API also allows such functions with keys that are long term (token) keys. The PKCS#11 API treats these keys as secure, provided there is no other reason to treat them as insecure. Even if the **`all`** parameter is set, if you do not include the **`CKA_ALLOWED_MECHANISMS`** with **`CKM_CONCATENATE_BASE_AND_KEY`**, this **`C_DeriveKey`** operation will not be allowed.

weak_*algorithm*

The **weak_***algorithm* parameter allows you to treat keys used with a weak algorithm as secure. For example, DES is not secure, but setting the parameter **weak_des** means that such keys are considered secure. You can apply the **weak_***algorithm* parameter to all keys that have a short fixed key length or whose algorithms have other security problems. As a guide, weak algorithms are those whose work factor to break is less than approximately 80 bits.

shortkey_*algorithm=bitlength*

The **shortkey_***algorithm=bitlength* parameter permits excessively short keys for the specified *algorithm* to be treated as secure. The parameter *bitlength* specifies the minimum length, in bits, that is to be considered secure. For example, RSA keys must usually be at least 1024 bits long in order to be treated as secure, but **shortkey_rsa=768** would allow 768-bit RSA keys to be treated as secure.

silent

The **silent** parameter turns off the warning output. Checks are still performed and still return failures correctly according to the other variables that are set.

Diagnostic warnings about questionable operations

When the **CKNFAST_OVERRIDE_SECURITY_ASSURANCES** environment variable is set to a value other than **all**, diagnostic messages are always generated for questionable operations. Each message contains the following elements:

- the PKCS #11 label of the key, if available
- the PKCS #11 identifier of the key, if available
- the hash of the key
- a summary of the problem.

If the problem is not that a questionable operation has been permitted because of a setting in **CKNFAST_OVERRIDE_SECURITY_ASSURANCES** it could be that an operation has failed. In such a case, the setting required to authorize the operation is noted.

By default diagnostic messages are sent to **stderr**. On Windows platforms, they are also sent to the Event Viewer. If a file name has been specified in the **CKNFAST_ASSURANCE_LOG** environment variable, diagnostic messages are also written to this file.

If **CKNFAST_DEBUG** is **1** or greater and a file is specified in **CKNFAST_DEBUGFILE**, the PKCS #11 library Security Assurance Mechanism log information is sent to the specified file. These variables must be set whenever **generatekey**atekey or KeySafe are used.

Note: If a file is specified in **CKNFAST_ASSURANCES_LOG** and no file is specified in **CKNFAST_DEBUGFILE** (or if **CKNFAST_DEBUG** is **0**), diagnostic messages are sent to **stderr** as well as to the file specified in **CKNFAST_ASSURANCES_LOG**.

CKNFAST_SEED_MAC_ZERO

Set this variable, to **1**, to cause the Korean SEED MAC mechanisms (**CKM_SEED_MAC** and **CKM_SEED_MAC_GENERAL**) to use zero padding. If this variable is not set, or set to **none**, then the SEED MAC mechanisms will use the default PKCS#5 padding scheme.

CKNFAST_SESSION_THREADSAFE

You must set this environment variable to **`yes`** if you are using the Sun PKCS #11 provider when running nCipherKM JCA/JCE code.

CKNFAST_TOKENS_PERSISTENT

This variable controls whether or not the Operator Cards that are created by your PKCS #11 application are persistent. If this variable is set when your application calls the PKCS #11 function that creates tokens, the Operator Card created is persistent. For more information about persistent OCSs, see your *User Guide*.

Note: Use of the nCipher PKCS #11 library to create tokens is deprecated, because it can only create 1/1 tokens in FIPS 140-2 level 2 security worlds. Use KeySafe or one of the command-line utilities to create OCSs.

CKNFAST_USE_THREAD_UPCALLS

If this variable is set and **`CKF_OS_LOCKING_OK`** is passed to **`C_Initialize`**, **`NFastApp_SetThreadUpcalls`** is called by means of **`nfast_usencthreads`** and only a single **`NFastApp_Connection`** is used, shared between all threads.

If this variable is set and mutex callbacks are passed to **`C_Initialize`** but **`CKF_OS_LOCKING_OK`** is not passed, **`C_Initialize`** fails with **`CKR_FUNCTION_FAILED`**. (**`NFastApp_SetThreadUpcalls`** requires more callbacks than just the mutex ones that PKCS #11 supports.)

If neither mutex callbacks nor **`CKF_OS_LOCKING_OK`** is passed, this variable is ignored. Only a single connection is used because the application must be single threaded in this case.

CKNFAST_LOAD_KEYS

This variable will load private objects at **`C_Login`** time, rather than at the first cryptographic operation.

CKNFAST_WRITE_PROTECTED

Set this variable to make your OCS or softcard (token) write-protected. If a token is write-protected, you cannot:

- Generate certificate, data, and key objects for that token.
- Modify attributes of an existing object.

Note: This environment variable does not prevent you from deleting an object from your token.

PKCS #11 with load-sharing

The behavior of the nCipher PKCS #11 library varies depending on whether or not you have enabled load-sharing. If you have enabled load sharing, the nCipher PKCS #11 library creates one virtual slot for each OCS and, optionally, also creates one slot for the module (or modules). If you have not enabled load-sharing, then for each module the nCipher PKCS #11 library creates one slot for the module and one slot for the smart-card reader.

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

Load-sharing enables you to load a single PKCS #11 token onto several nShield modules to improve performance. To enable successful load-sharing:

- you must connect all the modules to the same host
- you must have an Operator Card inserted into every slot from the same 1/*N* card set
- all the Operator Cards must have the same pass phrase. This is necessary in order to benefit from scaling and failover.

The nShield-specific API calls, **`C_LoginBegin`**, **`C_LoginNext`**, and **`C_LoginEnd`** (introduced in version 1.13.x to support *K*/*N* card sets) do not function in load-sharing mode. *K*/*N* support for card sets in load-sharing mode is only available if you first use **`with-nfast`** to load the logical token.

Note: Load-sharing must be enabled in PKCS11 in order to use softcards. Once they are correctly enabled, softcards appear as additional slots.

Logging in

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

The nCipher 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`**.

Session objects

Session objects are loaded on all modules.

Module failure

If a subset of the modules fails, the nCipher 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 nCipher PKCS #11 library will return a PKCS #11 error. Subsequent cryptographic commands will be run on other modules.

Compatibility

Before the implementation of load-sharing, the nCipher PKCS #11 library put 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.

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`.

Note: 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.

Note: The `C_InitToken` function is *not* supported for use in non-load-sharing FIPS 140-2 level 3 Security Worlds.

PKCS #11 without load-sharing

The nCipher 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).

Note: 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.

Note: If you have loaded a soft token, this appears as an additional slot. We do not recommend the use of soft tokens with PKCS #11.

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 on page 29). By default, when you insert a new card into a slot, the nCipher PKCS #11 library automatically logs out any token that had been logged in to the slot previously.

Note: The **`C_InitToken`** function is *not* supported for use in non-load-sharing FIPS 140-2 level 3 Security Worlds.

K/N support for PKCS #11

If you do not use the nCipher PKCS #11 library in load-sharing 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`** (introduced in version 1.13.*x*)
- by using the **`with-nfast`** command-line utility to load the logical token first, provided that the **`CKNFAST_LOADSHARING`** environment variable is set to **`0`**.

Generating and deleting NVRAM-stored keys with PKCS #11

You can use the nCipher PKCS #11 library to generate keys stored in nonvolatile memory (up to a maximum of 4 keys) if you have set the **`CKNFAST_NVRAM_KEY_STORAGE`** environment variable.

Generating NVRAM-stored keys

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

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

```
with-nfast --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:

```
with-nfast -t pkcs11app
```

   where *pkcs11app* is the name of your PKCS #11 application.
5. Generate the NVRAM-stored keys that you need (up to a maximum of 4 keys) as normal.
6. Stop or close *pkcs11app*.
7. Return to the command-line window you opened in step 1 and terminate the **`with-nfast --admin=nv pause`** process.

   Note: Do not allow the **`with-nfast --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.

Deleting NVRAM-stored keys

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

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

   ```
   with-nfast --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.
   Note: 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:

   ```
   with-nfast -tM pkcs11app
   ```

   where *pkcs11app* is the name of the PKCS #11 application that you use to delete 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 **`with-nfast --admin=nv pause`** process.
   Note: Do not allow the **`with-nfast --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.

nShield-specific PKCS #11 API extensions

In version 1.13.*x* and later of the nCipher PKCS #11 libraries, there are new nShield-specific API calls to support using nShield *K/N* card sets. These new 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 new API calls include three new functions, **`C_LoginBegin`**, **`C_LoginNext`** and **`C_LoginEnd`**, that are described in this chapter.

Note: The login sequence must occur in the same session.

Note: You cannot use the new API calls in load-sharing mode. To use *K/N* card sets in load-sharing mode, use **`with-nfast`** to load the logical token first. The new API calls also work in a non-load-sharing FIPS 140-2 level 3 Security Worlds.

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 */)
```

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 pass phrase 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 pass phrase 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 */)
```

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*/)
```

Note: 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.

Compiling and linking

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

- statically link the nCipher PKCS #11 library directly into your application
- dynamically link the nCipher 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 nCipher 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.

Note: 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.

Windows

All versions are built with Visual C++ 12.0. Thales supplies the following files:

- *%NFAST_HOME%*`\bin\cknfast.dll` and *%NFAST_HOME%*`\toolkits\pkcs11\cknfast.dll`: a dynamically linked library
  Note: 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
- *%NFAST_HOME%*`\c\ctd\lib\libdcknfast.lib`: a static library with position-independent code
- *%NFAST_HOME%*`\c\ctd\lib\libtcknfast.lib`: a threadsafe library.

Unix-based

For each of the various supported Unix-based systems, Thales supplies some or all of the following libraries:

- `libcknfast.so`, `libcknfast.so.a`, or `libcnfast.sl`: a standard, dynamically linked, shared library that can be used to create applications that must be dynamically linked with the nCipher 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. On HP-UX and AIX, both threadsafe and non-threadsafe versions are provided. HP-UX also provides a packed library pragma; for more information about this see the RSA Laboratories PKCS #11 documentation.

- HP-UX
  - **`ansic-thr/lib/libcknfast.sl`** (threadsafe)
  - **`ansic/lib/libcknfast.sl`** (non-threadsafe)
  - **`ansic64-thr/lib/libcknfast.sl`** (threadsafe)
  - **`ansic64/lib/libcknfast.sl`** (non-threadsafe)
- AIX
  - **`xlc_r/lib/libcknfast.so.a`** (threadsafe)
  - **`xlc/lib/libcknfast.so.a`** (non-threadsafe)

Note: When using the 64-bit xlc_r64 utilities on AIX platforms, the NFAST_HOME/toolkits/PKCS#11/ directory contains both **`libcknfast.so.a`** (32-bit library) and **`libcknfast-64.so.a`** (64-bit library). The 32-bit library is incorrectly used instead of the 64-bit library. The workaround is to move the 32-bit library out of this directory, and then rename the 64-bit library to **`libcknfast.so.a`**.

- **`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:

Solaris

Library type	Build notes
`/opt/nfast/c/ctd/gcc/lib`	This type of library is built with gcc 3.4.3.
`/opt/nfast/c/ctd/swspro/lib`	This type of library is built with Sun Workshop Compiler 5.0 (for Solaris 11).
`/opt/nfast/c/ctd/swspro64/lib`	This type of library is built with Sun Workshop Compiler 5.0 (for Solaris 11).

HP-UX

For HP-UX 11i v3, all of the following libraries are built with HP C/aC++ B3910B A.06.15 [May 16 2007]:

library type	Build notes
`/opt/nfast/c/ctd/ansic/lib`	This type of library is built for the ILP32 data model.
`/opt/nfast/c/ctd/ansic-thr/lib`	This type of library is built for the ILP32 data model and for threadsafe programs.
`/opt/nfast/c/ctd/ansic64/lib`	This type of library is built for the LP64 data model.
`/opt/nfast/c/ctd/ansic64-thr/lib`	This type of library is built for the LP64 data model and for threadsafe programs.

AIX

The following types of library are compiled on AIX 6.1 and are designed to be compatible with AIX 6.1 and AIX 7.1.

Note: The libraries have not been optimised for the Power 8 architecture.

library type	Build notes
`/opt/nfast/c/ctd/xlc/lib`	This type of library is built with xlc (`/usr/vac/bin/xlc`) in 32-bit mode.
`/opt/nfast/c/ctd/xlc_r/lib`	This type of library is built with xlc_r (`/usr/vac/bin/xlc_r`) in 32-bit mode for threadsafe programs.
`/opt/nfast/c/ctd/xlc64/lib`	This type of library is built with xlc (`/usr/vac/bin/xlc`) in 64-bit mode.
`/opt/nfast/c/ctd/xlc_r64/lib`	This type of library is built with xlc_r (`/usr/vac/bin/xlc`) in 64-bit mode for threadsafe programs.

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.

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 pass phrase.

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 pass phrase 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 nCipher PKCS #11 library can be used to manipulate Data Objects, Certificate Objects, and Key Objects.

Certificate Objects and Data Objects

The nCipher 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).

Key Objects

The following restrictions apply to keys.

Key types	Restrictions
RSA	Modulus greater than or equal to 1024. The nCipher PKCS #11 library requires all of the attributes for an RSA key object to be supplied, as listed in Table 27 of PKCS #11 Cryptographic Token Interface Standard version 2.01.
DSA	Modulus greater than or equal to 1024 in multiples of 8 bits.
Diffie-Hellman	Modulus greater than or equal to 1024.

Card pass phrases

All pass phrases 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 pass phrase supplied can be of any length.

Note: The **`ckinittoken`** program imposes a 512-byte limit on the pass phrase.

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

When **`C_Login`** is called, the pass phrase 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 pass phrase.

Note: 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.PINtPINtToken

Note: The **`C_InitToken`** function is *not* supported for use in non-load-sharing FIPS 140-2 level 3 Security Worlds.

Functions supported

The following sections list the PKCS #11 functions supported by the nCipher PKCS #11 library. For a list of supported mechanisms, see *Mechanisms* on page 39.

Note: Certain functions are included in PKCS #11 version 2.01 for compatibility with earlier versions only.

General purpose functions

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

- **`C_Finalize`**
- **`C_GetInfo`**
- **`C_GetFunctionList`**.

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.

Slot and token management functions

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

- **`C_GetSlotInfo`**
- **`C_GetTokenInfo`**
- **`C_GetMechanismList`**
- **`C_GetMechanismInfo`**.

C_GetSlotList

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`**.

Note: **`C_GetSlotList`** returns an array of handles. You can not 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`**.

C_InitToken

`C_InitToken` sets the card pass phrase to the same value as the PKCS #11 security officer's pass phrase and sets the **`CKF_USER_PIN_INITIALIZED`** flag.

Note: 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.

Note: The **`C_InitToken`** function is *not* supported for use in non-load-sharing FIPS 140-2 level 3 Security Worlds.

C_InitPIN

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

Because the nCipher PKCS #11 library can only maintain a single pass phrase, **`C_InitPIN`** has the effect of changing the PKCS #11 security officer's pass phrase.

Note: 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.

C_SetPIN

The card pass phrase may be any value.

Because the nCipher PKCS #11 library can only maintain a single pass phrase, **`C_SetPIN`** has the effect of changing the PKCS #11 security officer's pass phrase or, if called in a security officer session, the card pass phrase.

Note: 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.

Standard session management functions

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

- **`C_OpenSession`**
- **`C_CloseSession`**
- **`C_CloseAllSessions`**
- **`C_GetOperationState`**
- **`C_SetOperationState`**
- **`C_Login`**
- **`C_Logout`**

nShield session management functions

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

- **`C_LoginBegin`**
- **`C_LoginNext`**
- **`C_LoginEnd`**
- **`C_GetSessionInfo`**

`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).

Object management functions

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

- **`C_CreateObject`**
- **`C_CopyObject`**
- **`C_DestroyObject`**
- **`C_GetObjectSize`**
- **`C_GetAttributeValue`**
- **`C_SetAttributeValue`**
- **`C_FindObjectsInit`**
- **`C_FindObjects`**
- **`C_FindObjectsFinal`**

Encryption functions

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

- `C_EncryptInit`
- `C_Encrypt`
- `C_EncryptUpdate`
- `C_EncryptFinal`

Decryption functions

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

- `C_DecryptInit`
- `C_Decrypt`
- `C_DecryptUpdate`
- `C_DecryptFinal`

Message digesting functions

The following functions are performed on the host computer:

- `C_DigestInit`
- `C_Digest`
- `C_DigestUpdate`
- `C_DigestFinal`

Signing and MACing functions

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

- `C_SignInit`
- `C_Sign`
- `C_SignRecoverInit`
- `C_SignRecover`.

The functions `C_SignUpdate` and `C_SignFinal` are supported for:

- `CKM_SHA1_RSA_PKCS`
- `CKM_MD5_RSA_PKCS`.

Functions for verifying signatures and MACs

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

- `C_VerifyInit`
- `C_Verify`
- `C_VerifyRecover`
- `C_VerifyRecoverInit`.

The `C_VerifyUpdate` and `C_VerifyFinal` functions are supported for:

- `CKM_SHA1_RSA_PKCS`
- `CKM_MD5_RSA_PKCS`

Dual-purpose cryptographic functions

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

- `C_DigestEncryptUpdate`
- `C_DecryptDigestUpdate`.

The `C_SignEncryptUpdate` and `C_DecryptVerifyUpdate` functions are supported for:

- `CKM_SHA1_RSA_PKCS`
- `CKM_MD5_RSA_PKCS`

Key-management functions

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

- `C_GenerateKey`
- `C_GenerateKeyPair`
- `C_WrapKey`
- `C_UnwrapKey`
- `C_DeriveKey`

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

Random number functions

The nShield module has an onboard, hardware random number generator to handle the following random number functions:

- `C_GenerateRandom`
- `C_SeedRandom`

For this reason, it does not use seed values, and the `C_SeedRandom` function returns `CKR_RANDOM_SEED_NOT_SUPPORTED`.

Parallel function management functions

The following functions are supported in the approved fashion by returning the PKCS #11 status `CKR_FUNCTION_NOT_PARALLEL`:

- `C_GetFunctionStatus`
- `C_CancelFunction`

Callback functions

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

Mechanisms

The following table lists the mechanisms currently supported by the nCipher PKCS #11 library. Thales also provides vendor-supplied mechanisms, described in *Vendor-defined mechanisms* on page 44.

Mechanism	Functions						
	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[5]	—
CKM_AES_CMAC_GENERAL	—	Y	—	—	—	—	—
CKM_AES_CMAC	—	Y	—	—	—	—	—
CKM_AES_ECB_ENCRYPT_DATA	—	—	—	—	—	—	Y
CKM_AES_ECB	Y	—	—	—	—	Y[5]	—
CKM_AES_KEY_GEN	—	—	—	—	Y	—	—
CKM_AES_MAC_GENERAL	—	Y	—	—	—	—	—
CKM_AES_MAC	—	Y	—	—	—	—	—
CKM_CONCATENATE_BASE_AND_KEY	—	—	—	—	—	—	Y[8]
CKM_DES_CBC_ENCRYPT_DATA	—	—	—	—	—	—	Y
CKM_DES_CBC_PAD	Y[7]	—	—	—	—	Y[7]	—
CKM_DES_CBC	Y[7]	—	—	—	—	Y[7]	—
CKM_DES_ECB_ENCRYPT_DATA	—	—	—	—	—	—	Y[7]
CKM_DES_ECB	Y[7]	—	—	—	—	Y[7]	—
CKM_DES_KEY_GEN	—	—	—	—	Y[7]	—	—
CKM_DES_MAC_GENERAL	—	Y[7]	—	—	—	—	—
CKM_DES_MAC	—	Y[7]	—	—	—	—	—
CKM_DES2_KEY_GEN	—	—	—	—	Y	—	—
CKM_DES3_CBC_ENCRYPT_DATA	—	—	—	—	—	—	Y
CKM_DES3_CBC_PAD	Y	—	—	—	—	Y	—
CKM_DES3_CBC	Y	—	—	—	—	Y[5]	—
CKM_DES3_ECB_ENCRYPT_DATA	—	—	—	—	—	—	Y
CKM_DES3_ECB	Y	—	—	—	—	Y[5]	—
CKM_DES3_KEY_GEN	—	—	—	—	Y	—	—
CKM_DES3_MAC_GENERAL	—	Y	—	—	—	—	—
CKM_DES3_MAC	—	Y	—	—	—	—	—

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive Key
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[1]	—	—	—	—	—
CKM_EC_KEY_PAIR_GEN	—	—	—	—	Y[11]	—	—
CKM_ECDH1_DERIVE	—	—	—	—	—	—	Y[6]
CKM_ECDSA_SHA1	—	Y	—	—	—	—	—
CKM_ECDSA	—	Y[1]	—	—	—	—	—
CKM_GENERIC_SECRET_KEY_GEN	—	—	—	—	Y	—	—
CKM_MD5_HMAC_GENERAL	—	Y[7]	—	—	—	—	—
CKM_MD5_HMAC	—	Y[7]	—	—	—	—	—
CKM_MD5	—	—	—	Y	—	—	—
CKM_NC_MD5_HMAC_KEY_GEN	—	—	—	—	Y[7]	—	—
CKM_PBE_MD5_DES_CBC	—	—	—	—	Y	—	—
CKM_RIPEMD160	—	—	—	Y	—	—	—
CKM_RSA_9796	—	Y[1]	Y[1]	—	—	—	—
CKM_RSA_PKCS_KEY_PAIR_GEN	—	—	—	—	Y	—	—
CKM_RSA_PKCS_OAEP	Y	—	—	—	—	Y	—
CKM_RSA_PKCS_PSS[9]	Y	Y	—	—	—	—	—
CKM_RSA_PKCS	Y[1]	Y[1]	Y[1]	—	—	Y	—
CKM_RSA_X_509	Y[1]	Y[1]	Y[1]	—	—	X	—
CKM_RSA_X9_31_KEY_PAIR_GEN	—	—	—	—	Y	—	—
CKM_SHA_1_HMAC_GENERAL	—	Y[4]	—	—	—	—	—
CKM_SHA_1_HMAC	—	Y[4]	—	—	—	—	—
CKM_SHA_1	—	—	—	Y	—	—	—
CKM_SHA1_RSA_PKCS_PSS[9]	—	Y	—	—	—	—	—

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive Key
`CKM_SHA1_RSA_PKCS`	—	Y	—	—	—	—	—
`CKM_SHA224_HMAC_GENERAL`	—	Y[4]	—	—	—	—	—
`CKM_SHA224_HMAC`	—	Y[4]	—	—	—	—	—
`CKM_SHA224_RSA_PKCS_PSS`[9]	—	Y	—	—	—	—	—
`CKM_SHA224`	—	—	—	Y	—	—	—
`CKM_SHA256_HMAC_GENERAL`	—	Y[4]	—	—	—	—	—
`CKM_SHA256_HMAC`	—	Y[4]	—	—	—	—	—
`CKM_SHA256_RSA_PKCS_PSS`[9]	—	Y	—	—	—	—	—
`CKM_SHA256_RSA_PKCS`	—	Y	—	—	—	—	—
`CKM_SHA256`	—	—	—	Y	—	—	—
`CKM_SHA384_HMAC_GENERAL`	—	Y[4]	—	—	—	—	—
`CKM_SHA384_HMAC`	—	Y[4]	—	—	—	—	—
`CKM_SHA384_RSA_PKCS_PSS`[9]	—	Y	—	—	—	—	—
`CKM_SHA384_RSA_PKCS`	—	Y	—	—	—	—	—
`CKM_SHA384`	—	—	—	Y	—	—	—
`CKM_SHA512_HMAC_GENERAL`	—	Y[4]	—	—	—	—	—
`CKM_SHA512_HMAC`	—	Y[4]	—	—	—	—	—
`CKM_SHA512_RSA_PKCS_PSS`[9]	—	Y	—	—	—	—	—
`CKM_SHA512_RSA_PKCS`	—	Y	—	—	—	—	—
`CKM_SHA512`	—	—	—	Y	—	—	—
`CKM_WRAP_RSA_CRT_COMPONENTS`	—	—	—	—	—	Y[12]	—
`CKM_XOR_BASE_AND_DATA`	—	—	—	—	—	—	Y[3]

The nCipher library supports some mechanisms that are defined in versions of the PKCS #11 standard later than 2.01, although the nCipher 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 nCipher PKCS #11 library.
- The entry "X" indicates that a mechanism is not supported by the nCipher PKCS #11 library.

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

1. Single-part operations only.
2. This mechanism uses the eight octets following the key as the initializing vector as specified in PKCS#5 v2.
3. The base key and the derived key are restricted to **`DES`**, **`DES3`**, **`CAST5`** or **`Generic`**, though they may be of different types.
4. 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* on page 44.
5. Wrap secret keys only (private key wrapping must use **`CBC_PAD`**).
6. The **`CKM_ECDH1_DERIVE`** mechanism is supported. However, the mechanism only takes a **`CK_ECDH1_DERIVE_PARAMS`** struct in which **`CK_EC_KDF_TYPE`** is **`CKD_NULL`**, **`CKD_SHA1_KDF`**, **`CKD_SHA224_KDF`**, **`CKD_SHA256_KDF`**, **`CKD_SHA384_KDF`**, or **`CKD_SHA512_KDF`**. 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 now accepted.
7. These mechanisms are not supported in FIPS 140-2 Level 3 Security Worlds in firmware version 2.33.60 or later.
8. Before you can create a key for use with the derive mechanism **`CKM_CONCATENATE_BASE_AND_KEY`**, you must first 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 Security Assurance Mechanisms (SAMs) on the **`CKM_CONCATENATE_BASE_AND_KEY`** mechanism, see *derive_concatenate* on page 24
   - the **`CKA_ALLOWED_MECHANISMS`** attribute, see *Attributes* on page 52.
9. 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
SHA-384	384-bit
SHA-512	512-bit

10. The **`hashAlg`** and the **`mgf`** that are specified by the **`CK_RSA_PKCS_OEAP_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`**.

It is possible to specify a byte array using a data source if **`CKZ_DATA_SPECIFIED`** is set. If **`CKZ_DATA_SPECIFIED`** is not present, then **`pSourceData`** and **`pSourceDatalen`** are ignored. It is not a requirement to have source set, and the value can be zero.

11. For elliptic curve key pairs: when generating a key pair using **`C_GenerateKeyPair()`**, you may specify either **`CKA_DERIVE`** or **`CKA_SIGN`** but not both. This means that your **`CKK_EC`** key can only be used for either sign/verify or derive operations. If both types are included in the template, generation fails with **`CKR_TEMPLATE_INCONSISTENT`**. If nothing is specified in the template, then the default 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.
    Note: Brainpool curves cannot be specified as named curves: they must be supplied as custom curves with all parameters DER-encoded.
    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.
12. Wrap only.

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.

CKM_WRAP_RSA_CRT_COMPONENTS

This wrapping mechanism is available in version 1.17 or later of the nCipher PKCS #11 library. It uses a **`pMechanism->pParameter`** argument that is itself a **`CK_MECHANISM_PTR`** appropriate for the underlying encryption mechanism. The wrapping mechanism takes a pointer to a PKCS #11 template as its **`pWrappedKey`** argument.

The **`CK_ATTRIBUTE_PTR`** template is allocated by the calling application. The template is filled in by the calling application with the attribute types (**`CKA_PRIME_1`**, **`CKA_PRIME_2`**, **`CKA_EXPONENT_1`**, **`CKA_EXPONENT_2`**, **`CKA_COEFFICIENT`**), and the lengths of the value buffers, which are also allocated by the application. The **`pulWrappedKeyLen`** argument contains the length in bytes of the template, which is **`(5 * sizeof (CK_ATTRIBUTE_PTR))`**.

The usual method of calling **`C_WrapKey`** is with a **`NULL`** buffer to determine its output length. This is not available because **`C_WrapKey`** cannot specify the multiple levels of allocation required. If any part of this structure has an inappropriate size, the mechanism fails with a **`CKR_WRAPPED_KEY_LEN_RANGE`** error.

CKM_SEED_ECB_ENCRYPT_DATA & 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.

Note: These mechanisms are not supported in FIPS 140-2 Level 3 Security Worlds.

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);
```

CKM_SHA*_HMAC and CKM_SHA*_HMAC_GENERAL

This version of the library supports the PKCS #11 standard mechanisms for SHA-1 and SHA-2 HMAC as defined in PKCS #11 standard version 2.30:

- **`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`**

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 PKCS #11 standard does not provide an appropriate key type. Therefore, the vendor-defined key types **`CKK_SHA_1_HMAC`**, **`CKK_SHA224_HMAC`**, **`CKK_SHA256_HMAC`**, **`CKK_SHA384_HMAC`**, and **`CKK_SHA512_HMAC`**, are provided for use with these SHA-1 and SHA-2 HMAC mechanisms. To generate the key, use the appropriate vendor-defined 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_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 *Mechanisms for KISAAlgorithms* on page 49.

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.

Note: 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.

Note: 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.

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 140-2 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`**.

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

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.

Argument	Meaning
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.

Mechanisms for KISAAlgorithms

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.

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.

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.

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.

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.

CKM_KCDSA_PARAMETER_GEN

Note: 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.

SEED secret keys:

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.

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.

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.

For information on the padding schemes used by these mechanisms, see *CKNFAST_SEED_MAC_ZERO* on page 25.

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.

Attributes

The following sections describe how PCKS #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).

CKA_SENSITIVE

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

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

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.

CKA_EXTRACTABLE

`CKA_EXTRACTABLE` creates a key with an ACL including **`DeriveKey`** permission with **`DeriveRole_BaseKey`** and **`DeriveMech_RawEncrypt`**, as well as with **`DeriveMech_PKCS8Encrypt`** and **`DeriveMech_RSAComponents`** for private keys. IGN_RECOVER

CKA_ENCRYPT, CKA_DECRYPT, CKA_SIGN, CKA_VERIFY

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

CKA_WRAP, CKA_UNWRAP

`CKA_WRAP` creates a key with an ACL including **`DeriveKey`** permission with **`DeriveRole_WrapKey`**, **`DeriveMech_RawEncrypt`**, as well as with **`DeriveMech_PKCS8Encrypt`** and **`DeriveMech_RSAComponents`** for secret keys.

`CKA_UNWRAP` creates a key with an ACL including **`DeriveKey`** permission with **`DeriveRole_WrapKey`** and **`DeriveMech_RawDecrypt.`**

There is no unwrap mechanism that corresponds to **`DeriveMech_RSAComponents`**.

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.

ERIFY_RECOVERCKA_VERIFY_RECOVER

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

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.

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. 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.

`CKA_ALLOWED_MECHANISMS` is an optional attribute and does not have to be set for any other operations. 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.

CKA_MODIFIABLE

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

CKA_TOKEN

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

CKA_START_DATE, CKA_END_DATE

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

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 nCipher default is 65537, which as a PKCS #11 big integer is **`CK_BYTEpublic_exponent[ ] = { 1, 0, 1 };`**

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.

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 140-2 level 3 mode.

`CKR_FIPS_FUNCTION_NOT_SUPPORTED`

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

Utilities

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

ckdes3gen

```
ckdes3.gen.exe [p|--pin-for-testing=passphrase] | [n|-nopin]
```

This utility is an example of Triple DES key generation using the nCipher 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 pass phrase. You can supply a pass phrase on the command line with the **`--pin-for-testing`** option, or suppress the pass phrase request with the **`--nopin`** option. The pass phrase is displayed in the clear on the command line, so this option is appropriate only for testing.

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`**.

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 pass phrase argument and public objects only if invoked without a pass phrase 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 pass phrase. You can supply a pass phrase on the command line with the **`--pin-for-testing option`**, or suppress the pass phrase request with the **`--nopin`** option. The pass phrase is displayed in the clear on the command line, so this option is appropriate only for testing.

ckmechinfo

```
ckmechinfo.exe
```

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

ckrsagen

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

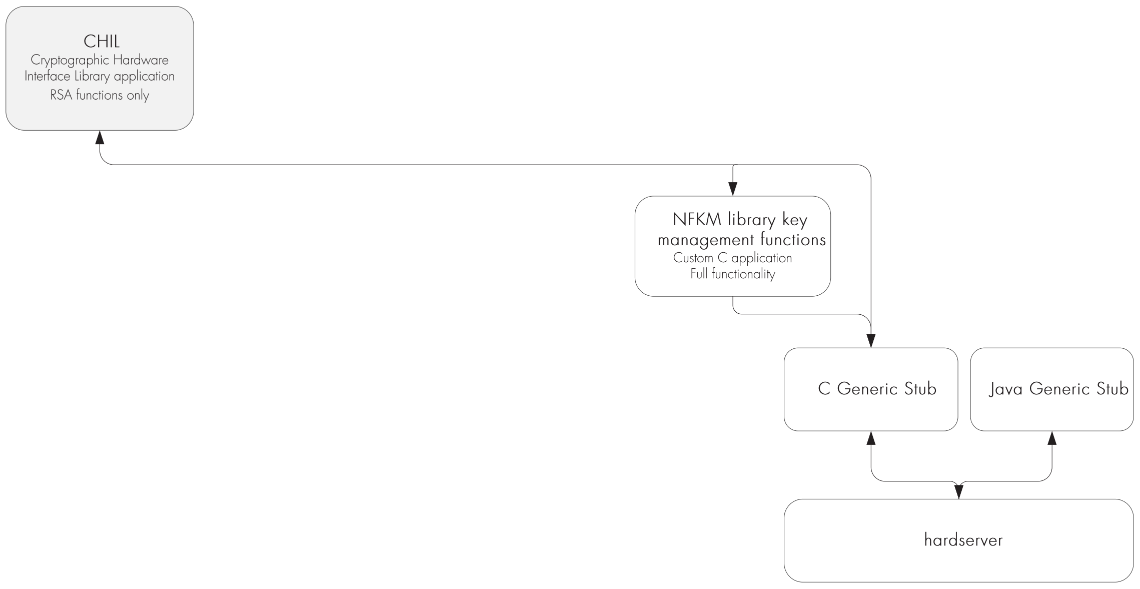
The **`ckrsagen`** utility is an example of RSA key pair generation using the nCipher PKCS #11 library. This is intended as a programmer’s example only and not for general use. Use the key generation routines within your PKCS #11 application.

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

Chapter 4: CHIL

The Cryptographic Hardware Interface Library (CHIL) is an API for cryptographic modules that perform modulo exponentiation, RSA cryptography, Diffie-Hellman key exchange and DSA for signature creation. RSA encryption can be performed on keys provided in the command or with keys protected by the cryptographic module.

Figure 3. CHIL architecture



The CHIL does not provide a user interface. Instead, it defines a number of callback functions. It is up to the application that is calling the library to implement the user interface required when a callback is made and to pass the input from the user back to the library.

The CHIL is supplied as a library that exports entry point(s) defined in the file `hwcryptohook.h`. This set of entry points provides a multithreaded, synchronous-within-each-thread facility. There is no support for asynchronous operation. If you require asynchronous operation, you must write directly to the nCore API.

For more information about the way the CHIL interface works with the nShield APIs, see Figure 3.

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

Structures the application must provide

Your application must define the structures described in this section. They are opaque to the CHIL plug-in. Your application may define them as it sees fit.

These structures are required if your application is multi-threaded:

```
typedef struct HWCryptoHook_MutexValue HWCryptoHook_Mutex;
typedef struct HWCryptoHook_CondVarValue HWCryptoHook_CondVar;
```

This structure is required if your application supports keys or cards protected by pass phrases:

```
typedef struct HWCryptoHook_PassphraseContextValue HWCryptoHook_PassphraseContext;
```

This structure is required if your application passes non-**`NULL`** values of context parameter to **`hwcrhk`** functions for the library to pass on to pass phrase or card change callbacks:

```
typedef struct HWCryptoHook_CallerContextValue HWCryptoHook_CallerContext;
```

The header files supplied by Thales provide the above declarations for these structures. If you want to use your own declarations, define the following before including **`hwcryptohook.h`** to prevent these declarations:

```
#define HWCRYPTOHOOK_DECLARE_APPTYPES 0
```

Note: If you define your own structures, the pointers to these structures must be ordinary pointers to **`structs`** or **`unions`**, otherwise the resulting combined program have type inconsistencies.

Structures provided by hwcrhk

The following structures are defined by the CHIL plug-in and are opaque to the application:

```
typedef struct HWCryptoHook_Context *HWCryptoHook_ContextHandle;
typedef struct HWCryptoHook_RSAKey *HWCryptoHook_RSAKeyHandle;
typedef struct HWCryptoHook_DHKey *HWCryptoHook_DHKeyHandle;
typedef struct HWCryptoHook_DSAKey *HWCryptoHook_DSAKeyHandle;
```

The CHIL plug-in returns pointers to these structures. The caller simply manipulates the pointers.

Multi-precision integers

RSA operations require large integers consisting of hundreds of bytes.

CHIL expects multi-precision integers to be stored as an array of limbs, each of which consists of a number of bytes. For example, each limb might be a 4-byte word in native byte order.

When you initialize the library, you set the limb size, the order of limbs within an integer, and the order of the bytes within a limb. You can choose whatever settings are most suitable for your application, but all the multi-precision integers within an application must use the same settings.

Note: Zero limbs at the Most Significant end are not permitted. The multi-precision value 0 is represented with no limbs (**`size==0`**).

CHIL uses the following structure to hold multi-precision integers:

```
typedef struct HWCryptoHook_MPIStruct {
        unsigned char *buf;
        size_t size;
} HWCryptoHook_MPI;
```

In this structure:

- **`*buf`**
  This is a pointer to the buffer holding the multi-precision integer or available for the library to return a multi-precision integer. The library does not update this pointer.
- **`size`**
  The size of the buffer in bytes. When the library returns a multi-precision integer, it is set to the actual size of the multi-precision integer. The **`size`** can be **`0`** but must be a multiple of the limb size, which is set in the **`HWCryptoHook_InitInfo`**.

Handling errors

When a **`HWCryptoHook`** function fails, it returns an error value. This value is:

- **`0`** for pointer-valued functions
- a negative number for integer-valued functions.

`HWCRYPTOHOOK_ERROR_FAILED` means that the failure is permanent and definite and that there should be no attempt to fall back to software. For applications that support just the acceleration functions, the "key material" can be an encoded key identifier; doing the operation in software would give incorrect answers.

`HWCRYPTOHOOK_ERROR_FALLBACK` means that doing the computation in software would seem reasonable. If an application pays attention to this value and is able to fall back, it should also set the **`Fallback`** init flags.

`HWCRYPTOHOOK_ERROR_MPISIZE` means that the output multi-precision integer buffer was full. The **`size`** has been set to the desired size.

Additionally, if you passed an **`ErrMsgBuf`** to the function, it outputs an error message there.

```
typedef struct {
        char *buf;
        size_t size;
} HWCryptoHook_ErrMsgBuf;
```

In this structure:

- **`*buf`**
  This is a pointer to the buffer. If you pass a NULL pointer, you must set **`size`** to 0, and nothing is recorded.
- **`size`**
  The value **`size`** is the size of the buffer. This value is not modified when an error is recorded.

When the buffer is filled, it is always null-terminated.

HWCryptoHook_Init

This function initializes the Cryptographic Hardware Interface Library (CHIL).

`HWCryptoHook_Init` takes a **`HWCryptoHook_InitInfo`** structure that includes settings for multi-precision integers and a pointer to the functions used for mutexes, condition variables, and the callback functions that provide the user interface.

All the callback functions must return 0 on success, or a nonzero integer (whose value is visible in the error message put in the buffer passed to the call). If a callback is not available, pass a null function pointer. The callbacks do not call down again into the CHIL plug-in.

A single operation might cause several calls to **`getpassphrase`** and **`requestphystoken`**. It is not necessary for **`getpassphrase`** or **`getphystoken`** to check that the pass phrase has been entered correctly or that the correct token has been inserted; the CHIL plug-in makes these checks. If the pass phrase has not been entered correctly, then the CHIL plug-in is responsible for calling these routines again, as appropriate, until the correct token(s) and pass phrase(s) are supplied as required or until any retry limits implemented by the CHIL plug-in are reached. For either callback, the application must allow the user to say "No" or "Cancel" to indicate that they neither know the pass phrase nor have the appropriate token. This situation should cause the callback to a return nonzero, indicating an error:

```
typedef
HWCryptoHook_ContextHandle HWCryptoHook_Init_t(const
  HWCryptoHook_InitInfo *initinfo,
                size_t initinfosize,
                const
  HWCryptoHook_ErrMsgBuf *errors,
  HWCryptoHook_CallerContext *cactx);
extern HWCryptoHook_Init_t HWCryptoHook_Init;
```

This function writes a message that includes the name and version number of the plug-in to **`msgbuf`**. In this function:

- **`initinfosize`**
  This is the size of the **`HWCryptoHook_InitInfo`** structure.
- **`initinfo`**
  This is a pointer to a **`HWCryptoHook_InitInfo`** structure:

```
typedef struct {
unsigned long flags;
FILE *logstream;
size_t limbsize;
int mslimbfirst;
int msbytefirst;
int maxmutexes;
int maxsimultaneous;
size_t mutexsize;
int (*mutex_init)(HWCryptoHook_Mutex*, HWCryptoHook_CallerContext *cactx);
int (*mutex_acquire)(HWCryptoHook_Mutex*);
void (*mutex_release)(HWCryptoHook_Mutex*);
void (*mutex_destroy)(HWCryptoHook_Mutex*);
size_t condvarsize;
int (*condvar_init)(HWCryptoHook_CondVar*, HWCryptoHook_CallerContext *cactx);
int (*condvar_wait)(HWCryptoHook_CondVar*, HWCryptoHook_Mutex*);
void (*condvar_signal)(HWCryptoHook_CondVar*);
void (*condvar_broadcast)(HWCryptoHook_CondVar*);
void (*condvar_destroy)(HWCryptoHook_CondVar*);
int (*getpassphrase)(const char *prompt_info,
      int *len_io,
      char *buf,
      HWCryptoHook_PassphraseContext *ppctx,
      HWCryptoHook_CallerContext *cactx);
int (*getphystoken)(const char *prompt_info,
      const char *wrong_info,
      HWCryptoHook_PassphraseContext *ppctx,
      HWCryptoHook_CallerContext *cactx);
void (*logmessage)(void *logstream, const char *message);
void *(*malloc_hook)(size_t, HWCryptoHook_CallerContext *cactx);
void *(*realloc_hook)(void*, size_t, HWCryptoHook_CallerContext *cactx);
void (*free_hook)(void*, HWCryptoHook_CallerContext *cactx);
} HWCryptoHook_InitInfo;
```

In this structure:

- **`unsigned long flags`**
  The following flags are defined:
  - **`#define HWCryptoHook_InitFlags_FallbackModExp 0x02UL`**
    This flag enables requests for fallback to software in case of problems with the hardware support. It indicates to the cryptographic provider that the application is prepared to fall back to software operation if the **`ModExp`*** or **`RSAImmed`*** functions return **`HWCRYPTOHOOK_ERROR_FALLBACK`**. If this flag is not set, the function never returns **`HWCRYPTOHOOK_ERROR_FALLBACK`**. The flag also causes the cryptographic provider to avoid repeated attempts to contact dead hardware within a short interval, if appropriate.
  - **`#define HWCryptoHook_InitFlags_FallbackRSAImmed 0x04UL`**
    This flag enables requests for fallback to software in case of problems with the hardware support. It indicates to the cryptographic provider that the application is prepared to fall back to software operation if the **`ModExp`*** or **`RSAImmed`*** functions return **`HWCRYPTOHOOK_ERROR_FALLBACK`**. If this flag is not set, the function never returns **`HWCRYPTOHOOK_ERROR_FALLBACK`**. The flag also causes the cryptographic provider to avoid repeated attempts to contact dead hardware within a short interval, if appropriate.
  - **`#define HWCryptoHook_InitFlags_SimpleForkCheck 0x0010UL`**

If this flag is not set, the library is allowed to assume that the application does not fork and to call the library in the child process (or processes). When this flag is set, such functionality is allowed. However, after a fork, neither parent nor a child process can unload any loaded keys or call **`HWCryptoHook_Finish`**. Instead, they should call exit (or die with a signal) without calling **`HWCryptoHook_Finish`**. After all the children have died, the parent may unload keys or call **`HWCryptoHook_Finish`**.

Note: This flag only has the desired effect on Unix platforms.

- **`*logstream`**

  A log message is generated at least every time something goes wrong, and an **`ErrMsgBuf`** is filled in (or would be filled in if one were provided). Other diagnostic information can also be written to the log message, including more detailed reasons for errors that are reported in an **`ErrMsgBuf`**. When a log message is generated, a message is sent to the logstream if the logstream is nonzero.

  The CHIL plug-in may also provide facilities to specify that copies of log messages be sent elsewhere and make adjustments to the verbosity of the log messages.

  Log messages consist of whole lines. Each line is prefixed by a descriptive string containing the date, time, and identity of the CHIL plug-in. This descriptive string ends at the first occurrence of the string ": " (that is, a colon followed by a space) in the message. Errors on the logstream are not reported anywhere.
- **`limbsize`**

  This is the size in bytes of limbs within multi-precision integers. It must be a power of 2.
- **`mslimbfirst`**

  This is the order of limbs within multi-precision integers:
  - 1 (most significant limb first)
  - 0 (least significant limb first).
- **`msbytefirst`**

  This is the order of bytes within a limb, which is independent of the order of the limbs themselves:
  - 1 (most significant byte first)
  - 0 (least significant byte first)
  - -1 (native order for the platform).
- **`*maxmutexes`**

  This is a small limit on the number of simultaneous mutexes that are requested by the library. If there is no small limit, set it to **`0`**.

  If the CHIL plug-in cannot create the advertised number of mutexes, the calls to its functions may fail. If a low number of mutexes is advertised, the plug-in does the best it can. Making larger numbers of mutexes available may improve performance and parallelism by reducing contention over critical sections.

  Unavailability of any mutexes, implying single-threaded operation, should be indicated by the setting the mutex pointers to **`NULL`**.
- **`maxsimultaneous`**

  This sets **`maxsimultaneous`** to the maximum number of simultaneous calls to the modulo exponentiation functions that your application makes. If you do not know what this number is, set this value to **`0`** in order to make the library use a default value.

- **`mutexsize`**

  The semantics of acquiring and releasing mutexes, and broadcasting and waiting on condition variables, are expected to be those from POSIX threads (pthreads). The mutexes may be (in the terminology of pthread) fast mutexes, recursive mutexes, or nonrecursive mutexes.
  The **`mutex_release`**, **`condvar_signal`**, **`condvar_broadcast`**, and **`condvar_destroy`** functions must always succeed when given a valid argument. If they are given an invalid argument, then the program (that is, the CHIL plug-in and the application) has an internal error, and they should abort the program.
  In single-threaded programs, set all mutex and condition variable entries to **`0`**.
- **`condvarsize`**

  For greater efficiency, the plug-in may use condition variables internally for synchronization. In this case, **`maxsimultaneous`** is ignored, but the mutex functions must be available.
- **`int (*getpassphrase)(const char *prompt_info`**

  This is a pointer to the callback function used to prompt the user for a pass phrase. If this pointer is set to **`NULL`**, the program can not use Operator Card Sets protected by pass phrases. Pass phrases and the **`prompt_info`**, if they contain high-bit-set characters, are UTF-8. The **`prompt_info`** can be a null pointer if no prompt information is available (it cannot be an empty string). It cannot contain text like "enter pass phrase"; instead its text should be of a form like "Operator Card for John Smith" or "SmartCard in Module#1, Slot#1".
- **`int (*getphystoken)(const char *prompt_info`**

  This flag requests that the human user physically insert a different smart card. The plug-in checks to see whether the currently inserted token or tokens are appropriate. If they are, the plug-in does make this call. If you pass a **`NULL`** pointer for **`requestphystoken`**, the program does not support loading keys if either the key requires authorization by several cards or the card protecting the key is not present when the key is loaded.
  `* prompt_info` is as for **`getpassphrase`** (see *HWCryptoHook_Init* on page 59). **`wrong_info`** is a description of the currently inserted token(s) so that the user is told what something is. **`wrong_info`** is a description of the currently inserted token(s) so that the user is told what something is. **`wrong_info`**, like **`prompt_info`**, can be null, but cannot be an empty string. Its contents are syntactically similar to that of **`prompt_info`**.
- **`void (*logmessage)(void *logstream, const char *message);`**

  This is the size of the **`HWCryptoHook_InitInfo`** structure.
- **`void *(*malloc_hook)(size_t, HWCryptoHook_CallerContext *cactx);,`**
- **`void *(*realloc_hook)(void*, size_t, HWCryptoHook_CallerContext *cactx);,`**
- **`void (*free_hook)(void*, HWCryptoHook_CallerContext *cactx);`**

  Pass **`0`** in order to use the standard C library **`malloc/realloc/free`**. If callbacks are supplied, they must have standard ANSI C89 semantics.

HWCryptoHook_RandomBytes

This command returns a block of bytes filled with random data generated by the module's on-board hardware random number generator.

```
typedef
int HWCryptoHook_RandomBytes_t(HWCryptoHook_ContextHandle hwctx,
    unsigned char *buf,
    size_t len,
    const HWCryptoHook_ErrMsgBuf *errors);
extern HWCryptoHook_RandomBytes_t HWCryptoHook_RandomBytes;
```

In this command:

- **`hwctx`**
  This is returned by **`HWCryptoHook_Init`**.
- **`*buf`**
  This is a pointer to a buffer to hold the returned bytes.
- **`len`**
  This is the size of the buffer in bytes.

HWCryptoHook_ModExp & HWCryptoHook_RSAImmedPub

These commands perform a modulo exponentiation on multi-precision integers supplied with the command.

```
typedef
int HWCryptoHook_ModExp_t(HWCryptoHook_ContextHandle hwctx,
        HWCryptoHook_MPI a,
        HWCryptoHook_MPI p,
        HWCryptoHook_MPI n,
        HWCryptoHook_MPI *r,
const HWCryptoHook_ErrMsgBuf *errors); extern HWCryptoHook_ModExp_t HWCryptoHook_ModExp;
```

```
typedef
int HWCryptoHook_RSAImmedPub_t(HWCryptoHook_ContextHandle hwctx,
        HWCryptoHook_MPI m,
        HWCryptoHook_MPI e,
        HWCryptoHook_MPI n,
        HWCryptoHook_MPI *r,
const HWCryptoHook_ErrMsgBuf *errors); extern HWCryptoHook_RSAImmedPub_t HWCryptoHook_
RSAImmedPub;
```

In this command:

- **`HWCryptoHook_MPI a`**
  A base.
- **`HWCryptoHook_MPI p`**
  P power.
- **`HWCryptoHook_MPI n`**
  N modulus.
- **`HWCryptoHook_MPI *r`**
  = A^P MOD_N.
- **`HWCryptoHook_MPI m`**

  Message.
- **`HWCryptoHook_MPI e`**
  Exponent.
- **`HWCryptoHook_MPI n`**
  Key modulus.
- **`HWCryptoHook_MPI *r`**
  Result.

HWCryptoHook_ModExpCRT and HWCryptoHook_RSAImmedPriv

These commands perform modulo exponentiation using the Chinese Remainder Theorem on multi-precision integers supplied with the command. Use **`HWCryptoHook_RSALoadKey`** and **`HWCryptoHook_RSA`** to perform these operations on stored keys.

```
typedef
int HWCryptoHook_ModExpCRT_t(HWCryptoHook_ContextHandle hwctx,
        HWCryptoHook_MPI a,
        HWCryptoHook_MPI p,
        HWCryptoHook_MPI q,
        HWCryptoHook_MPI dmp1,
        HWCryptoHook_MPI dmq1,
        HWCryptoHook_MPI iqmp,
        HWCryptoHook_MPI *r,
        const HWCryptoHook_ErrMsgBuf *errors);
extern HWCryptoHook_ModExpCRT_t HWCryptoHook_ModExpCRT;
```

```
typedef
int HWCryptoHook_RSAImmedPriv_t(HWCryptoHook_ContextHandle hwctx,
        HWCryptoHook_MPI m,
        HWCryptoHook_MPI p,
        HWCryptoHook_MPI q,
        HWCryptoHook_MPI dmp1,
        HWCryptoHook_MPI dmq1,
        HWCryptoHook_MPI iqmp,
        HWCryptoHook_MPI *r,
        const HWCryptoHook_ErrMsgBuf *errors);
extern HWCryptoHook_RSAImmedPriv_t HWCryptoHook_RSAImmedPriv;
```

In these commands:

- **`HWCryptoHook_MPI a`**
  A base
- **`HWCryptoHook_MPI p`**
  - **`HWCryptoHook_ModExpCRT`**
    P modulus larger factor.
  - **`HWCryptoHook_RSAImmedPriv`**
    First factor.

- **`HWCryptoHook_MPI q`**
  - **`HWCryptoHook_ModExpCRT`**
    Q modulus smaller factor.
  - **`HWCryptoHook_RSAImmedPriv`**
    Second factor.
- **`HWCryptoHook_MPI dmp1`**
  D MOD_{P-1}.
- **`HWCryptoHook_MPI dmq1`**
  D MOD_{Q-1}.
- **`HWCryptoHook_MPI iqmp`**
  Q^{-1} MOD_{P}.
- **`HWCryptoHook_MPI *r`**
  Result.
- **`HWCryptoHook_MPI m`**
  Ciphertext.

HWCryptoHook_RSALoadKey

This function loads a private key from a key store. You can then use **`HWCryptoHook_RSA`** to perform decryptions or signatures with this key. Use **`HWCryptoHook_RSAUnloadKey`** to unload the key when you have finished using it.

To perform encryptions or verifications using a public key, use **`HWCryptoHook_RSAGetPublicKey`** and then **`HWCryptoHook_RSAImmedPub.`**

The function may issue a callback to **`getphystoken`** or **`getpassphrase`**.

```
typedef
int HWCryptoHook_RSALoadKey_t(HWCryptoHook_ContextHandle hwctx,
        const char *key_ident,
        HWCryptoHook_RSAKeyHandle *keyhandle_r,
        const HWCryptoHook_ErrMsgBuf *errors,
        HWCryptoHook_PassphraseContext *ppctx);
extern HWCryptoHook_RSALoadKey_t HWCryptoHook_RSALoadKey
```

In this function:

- **`*key_ident`**
  This is a null-terminated string configured by the user through the application's usual configuration mechanisms. It is provided to the user by the cryptographic provider's key-management system. The user must be able to enter at least any string of between 1 and 1023 characters inclusive that consists of printable 7-bit ASCII characters. The provider avoids using any characters except alphanumeric characters and the punctuation characters _ `- + . / @` ~ (the user is expected to be able to enter these without quoting). The string can be case sensitive. The application can allow the user to enter other null-terminated strings, and the provider must cope (returning an error if the string is not valid).
- **`*keyhandle_r`**
  If the key does not exist, **`keyhandle_r`** is set to **`0`** instead of to a key handle. This is not an error.

HWCryptoHook_RSA

This function performs an RSA decryption using a previously loaded private key.

```
typedef
int HWCryptoHook_RSA_t(HWCryptoHook_MPI m,
        HWCryptoHook_RSAKeyHandle k,
        HWCryptoHook_MPI *r,
        const HWCryptoHook_ErrMsgBuf *errors);
extern HWCryptoHook_RSA_t HWCryptoHook_RSA;
```

In this function:

- **`HWCryptoHook_MPI m`**
  This is a message to decrypt or sign. The message must have been padded according to the appropriate specification. The **`HWCryptoHook_RSA`** function only completes M^d **`mod N`**.
- **`HWCryptoHook_RSAKeyHandle k`**
  This is a key handle returned by **`HWCryptoHook_RSALoadKey`**.
- **`HWCryptoHook_MPI *r`**
  This is the plain text returned.

HWCryptoHook_RSAUnloadKey

This function unloads an RSA key that you have previously loaded.

```
typedef
int HWCryptoHook_RSAUnloadKey_t(HWCryptoHook_RSAKeyHandle k,
        const HWCryptoHook_ErrMsgBuf *errors);
extern HWCryptoHook_RSAUnloadKey_t HWCryptoHook
```

In this function **`HWCryptoHook_RSAKeyHandle k`** is the key handle returned by **`HWCryptoHook_RSALoadKey`**. It fails only when there are locking problems or other serious internal problems.

HWCryptoHook_RSAGetPublicKey

This function returns the public half of a key pair identified by a key handle.

Although this function is provided for acquiring the public key value, it is not the purpose of this API to deal fully with the handling of the public key.

The CHIL plug-in does not store certificates. It is expected that the CHIL supplier's key-generation program provides general facilities for producing X.509 self-certificates and certificate requests in PEM format, which the application should be able to import. If this certificate handling is not appropriate, the application should instruct the user not to use the provider's tools to generate certificate requests. Instead the application should use **`HWCryptoHook_RSAGetPublicKey`** and generate and store appropriate certificates and requests itself.

```
typedef
int HWCryptoHook_RSAGetPublicKey_t(HWCryptoHook_RSAKeyHandle k,
        HWCryptoHook_MPI *n,
        HWCryptoHook_MPI *e,
        const HWCryptoHook_ErrMsgBuf *errors);
extern HWCryptoHook_RSAGetPublicKey_t HWCryptoHook_RSAGetPublicKey;
```

In this function:

- **`HWCryptoHook_RSAKeyHandle k`**
  This is the key handle returned by **`HWCryptoHook_RSALoadKey`**.
- **`HWCryptoHook_MPI *n`**
  This is the public key modulus.
- **`HWCryptoHook_MPI *e`**
  This is the public key exponent.

HWCryptoHook_DHLoadKey

This function loads a private key from a key store. You can then use **`HWCryptoHook_DH`** to perform a key exchange with this key. Use **`HWCryptoHook_DHUnloadKey`** to unload the key when you have finished using it.

```
typedef
int HWCryptoHook_DHLoadKey_t(HWCryptoHook_ContextHandle hwctx,
        const char *key_ident,
        HWCryptoHook_DHKeyHandle *keyhandle_r,
        const HWCryptoHook_ErrMsgBuf *errors,
        HWCryptoHook_PassphraseContext *ppctx);
extern HWCryptoHook_DHLoadKey_t HWCryptoHook_DHLoadKey;
```

In this function:

- **`*key_ident`**
  This is a null-terminated string configured by the user through the application's usual configuration mechanisms. It is provided to the user by the cryptographic provider's key-management system. The user must be able to enter at least any string of between 1 and 1023 characters inclusive that consists of printable 7-bit ASCII characters. The provider avoids using any characters except alphanumeric characters and the punctuation characters _ `- + . / @` ~ (the user is expected to be able to enter these without quoting). The string can be case sensitive. The application can allow the user to enter other null-terminated strings, and the provider must cope (returning an error if the string is not valid).
- **`*keyhandle_r`**
  If the key does not exist, **`keyhandle_r`** is set to **`0`** instead of to a key handle. This is not an error.

HWCryptoHook_DH

This function performs a Diffie-Hellman key exchange using a previously loaded private key.

```
typedef
int HWCryptoHook_DH_t(HWCryptoHook_MPI gx,
HWCryptoHook_DHKeyHandle k,
HWCryptoHook_MPI *r,
const HWCryptoHook_ErrMsgBuf *errors);
extern HWCryptoHook_DH_t HWCryptoHook_DH;
```

In this function:

- **gx**
  gx is the public key value of the other party.
- **k**
- **k** is a key handle returned by HWCryptoHook_DHLoadKey.
  r
- **r** is the shared secret returned.
  The public value **gx** and the private key **k** are assumed to share the same discrete log group parameters.

HWCryptoHook_DHUnloadKey

This function unloads a Diffie-Hellman key that you have previously loaded.

```
typedef
int HWCryptoHook_DHUnloadKey_t(HWCryptoHook_DHKeyHandle k,
const HWCryptoHook_ErrMsgBuf *errors);
extern HWCryptoHook_DHUnloadKey_t HWCryptoHook_DHUnloadKey;
```

In this function **k** is the key handle returned by **HWCryptoHook_DHLoadKey**. It fails only when there are locking problems or other serious internal problems.

HWCryptoHook_DHGetPublicKey

This function returns the public half of a key pair identified by a key handle.

Although this function is provided for acquiring the public key value, it is not the purpose of this API to deal fully with the handling of the public key.

The CHIL plug-in does not store certificates. It is expected that the cryptographic supplier's key-generation program provides general facilities for producing X.509 self-certificates and certificate requests in PEM format, which the application should be able to import. If this certificate handling is not appropriate, the application should instruct the user not to use the provider's tools to generate certificate requests. Instead the application should use **HWCryptoHook_DHGetPublicKey** and generate and store appropriate certificates and requests itself.

```
typedef
int HWCryptoHook_DHGetPublicKey_t(HWCryptoHook_DHKeyHandle k,
HWCryptoHook_MPI *p,
HWCryptoHook_MPI *g,
HWCryptoHook_MPI *gx,
const HWCryptoHook_ErrMsgBuf *errors);
extern HWCryptoHook_DHGetPublicKey_t HWCryptoHook_DHGetPublicKey;
```

In this function:

- **`p`**
  `p` is the prime from the discrete log group.
- **`g`**
  `g` is the generator from the discrete log group.
- **`gx`**
  `gx` is the public key value.

HWCryptoHook_DSALoadKey

This function loads a private key from a key store. You can then use **`HWCryptoHook_DSA`** to perform signing operation with this private key. Use **`HWCryptoHook_DSAUnloadKey`** to unload the key when you have finished with it.

```
typedef
int HWCryptoHook_DSALoadKey_t(HWCryptoHook_ContextHandle hwctx,
const char *key_ident,
HWCryptoHook_DSAKeyHandle *keyhandle_r,
const HWCryptoHook_ErrMsgBuf *errors,
HWCryptoHook_PassphraseContext *ppctx);
extern HWCryptoHook_DSALoadKey_t HWCryptoHook_DSALoadKey;
```

In this function:

- **`*key_ident`**
  This is a null-terminated string configured by the user through the application's usual configuration mechanisms. It is provided to the user by the cryptographic provider's key-management system. The user must be able to enter at least any string of between 1 and 1023 characters inclusive that consists of printable 7-bit ASCII characters. The provider avoids using any characters except alphanumeric characters and the punctuation characters _ `-` `+` `.` `/` `@` and ~(the user is expected to be able to enter these without quoting). The string can be case sensitive. The application can allow the user to enter other null-terminated strings, and the provider must cope (returning an error if the string is not valid).
- **`*keyhandle_r`**
  If the key does not exist, **`keyhandle_r`** is set to **`0`** instead of to a key handle. This is not an error.

HWCryptoHook_DSA

This function performs a DSA signing operation with a previously loaded private key.

```
typedef
int HWCryptoHook_DSA_t(const unsigned char *h,
HWCryptoHook_DSAKeyHandle k,
HWCryptoHook_MPI *r,
HWCryptoHook_MPI *s,
const HWCryptoHook_ErrMsgBuf *errors;
extern HWCryptoHook_DSA_t HWCryptoHook_DSA;
```

In this function:

- **`*h`**
  `h` is expected to be a 20-byte SHA-1 hash.
- **`k`**
  `k` is a key handle returned by HWCryptoHook_DSALoadKey.
- **`*r`**
  `r` is the **`r`**-value of the signature returned.
- **`*s`**
  `s` is the **`s`**-value of the signature returned.

HWCryptoHook_DSAUnloadKey

This function unloads a DSA key that you have previously loaded.

```
typedef
int HWCryptoHook_DSAUnloadKey_t(HWCryptoHook_DSAKeyHandle k,            See 1 below
const HWCryptoHook_ErrMsgBuf *errors);
extern HWCryptoHook_DSAUnloadKey_t HWCryptoHook_DSAUnloadKey;
```

In this function **`k`** is the key handle returned by **`HWCryptoHook_DSALoadKey`**. It fails only when there are locking problems or other serious internal problems.

HWCryptoHook_DSAGetPublicKey

This function returns the public half of a key pair identified by a key handle.

Although this function is provided for acquiring the public key value, it is not the purpose of this API to deal fully with the handling of the public key.

The CHIL plug-in does not store certificates. It is expected that the cryptographic supplier’s key-generation program provides general facilities for producing X.509 self-certificates and certificate requests in PEM format, which the application should be able to import. If this certificate handling is not appropriate, the application should instruct the user not to use the provider’s tools to generate certificate requests. Instead, applications should use **`HWCryptoHook_DSAGetPublicKey`** and generate and store appropriate certificates and requests itself.

```
typedef
int HWCryptoHook_DSAGetPublicKey_t(HWCryptoHook_DSAKeyHandle k,
HWCryptoHook_MPI *p,
HWCryptoHook_MPI *q,
HWCryptoHook_MPI *g,
HWCryptoHook_MPI *y,
const HWCryptoHook_ErrMsgBuf *errors);
extern HWCryptoHook_DSAGetPublicKey_t HWCryptoHook_DSAGetPublicKey;
```

In this function:

- **`*p`**
  `p` is the prime from the discrete log group.
- **`*q`**
  `q` is the **`q`** value from the discrete log group (A 160-bit prime factor of **`p-1`**).
- **`*g`**
  `g` is the generator from the discrete log group.
- **`*y`**
  `y` is the public key value.

HWCryptoHook_Finish

This function closes the CHIL session. You must not have any calls going or keys loaded when you call this function.

```
typedef
void HWCryptoHook_Finish_t(HWCryptoHook_ContextHandle hwctx);
extern HWCryptoHook_Finish_t HWCryptoHook_Finish;
```

CHIL error logging

Error message logging is implemented in CHIL using the **`logmessage`** callback defined in **`hwcryptohook.h`**:

```
  void (*logmessage)(void *logstream, const char *message);
```

When a log message is generated, this callback is called. It should write a message to the relevant logging arrangements. A log message is generated at least every time something goes wrong and an **`ErrMsgBuf`** is filled in (or would be if one was provided). Other diagnostic information may be written there too, including more detailed reasons for errors which are reported in an **`ErrMsgBuf`**.

The message string passed is null-terminated and can be of arbitrary length. It is not prefixed by the time and date, nor by the name of the library that is generating it; if this is required, the **`logmessage`** callback must do it. The message does not have a trailing newline (though it can contain internal newlines).

If a null pointer is passed for **`logmessage`** a default function is used. The default function treats **`logstream`** as a **`FILE*`** which has been converted to a **`void*`**. If **`logstream`** is 0 (zero) it does nothing. Otherwise it adds the date and time and library name to the beginning of the message and writes it to **`logstream`**. Each line is prefixed by a descriptive string containing the date, time and identity of the cryptographic plug-in. Errors on **`logstream`** are not reported anywhere, and the default function doesn't flush the stream, so the application must set the buffering how it wants it.

The cryptographic plug-in may also provide a facility to have copies of log messages sent elsewhere, and/or for adjusting the verbosity of the log messages; any such facilities must be configured by some means external to the CHIL.

The CHIL API permits the plug-in to return a single-line error string for each call. This error string sometimes contains a list of errors, in a form like this:

```
Failed to load key (codes: m1MU m2b0BL28 m2BN)
```

Each code consists of a sequence of alphanumeric characters, comprised of:

- optional prefixes, identifying where the error occurred
- a 2-character error code
- an optional suffix.

These codes are an important diagnostic tool, and are useful for debugging.

The following table lists the prefixes, suffixes and their meanings:

Code	Meaning
Prefixes	
m	On module # (for module 1, m=1).
b	Blob # (usually there is only one blob, which is numbered b0).
s	Slot # (usually s0).
i	Share index # (usually i1).
Suffixes	
#	Trailing digits are a status code value.
*	A numerical value (see details of message).
&	Integer from **`getphystoken`**/**`getpassphrase`** upcall.

The following table lists the error codes and the corresponding long messages which appear in the log file(s).

Error code	Long message
MQ	Module in unknown state.
MM	Module in maintenance mode.
MU	Module in uninitialized mode.
MF	Module has failed.
MP	Module in Pre-Init mode.
MS*	Module in strange state (* is an NFKM library state code).
BE#	Blob Examine failed.
BR	Blob requires recovery key.
BL#	Blob Load failed.
BN	Blob(s) not found or unsuitable (after other B… codes).
CU	Card set protecting blob is unknown.
CI#	Card info unavailable.
MNA	Module no longer available.
CB#	Card set begin loading failed.
SE	Slot empty.
SC*	Slot does not contain operator card (* is NFKM card code).
SD	Slot contains different operator card.
SA	Slot contains card/share already loaded.
CP	Card has a pass phrase, cannot load.
PR&	Pass phrase read failed.
PH#	Pass phrase hash failed.
CRU#	Card read failed with unexpected error.
CR#	Card read failed.
PI	Pass phrase incorrect.
CA	Card(s) inserted are not (yet) appropriate.
CN&	Card(s) needed but not acquired.
CF#	Card loading finish failed.

Example

```
Failed to load key (codes: m1MU m2b0BL28 m2BN)
```

The example error message above indicates that:

- CHIL tried to load the key on module #1, but it was in uninitialized mode
- CHIL then tried to load the key on module 2. Trying the first blob for the key (usually a key has only one blob) on module #2, it got error Status 28 from **`LoadBlob`** (that is **`UnknownKM`**). This indicates that the module is probably not programmed into the correct Security World.)
- the third code is just a confirmation that no suitable blobs were found for module 2.

Because the key was not loaded on any module, the operation failed.

Chapter 5: Microsoft CAPI CSP

We provide a Cryptographic Service Provider (CSP) that implements the Crypto API (CAPI) supported in Windows 2008 and later.

The rest of this chapter details the features and implementation details of the CAPI. Except where this chapter specifies otherwise, the Security World Software implementation conforms to the Microsoft CSP interface. For more information, see the Microsoft CSP documentation.

Crypto API CSP

The following provider types are supported:

- `PROV_RSA_FULL` (nShield Enhanced Cryptographic Provider)
- `PROV_RSA_AES` (nShield Enhanced RSA and AES Cryptographic Provider)
- `PROV_RSA_SCHANNEL` (nShield Enhanced SChannel Cryptographic Provider)
- `PROV_DSS` (nShield DSS Signature Cryptographic Provider)
- `PROV_DSS_DH` (nShield Enhanced DSS and Diffie-Hellman Cryptographic Provider)
- `PROV_DH_SCHANNEL` (nShield Enhanced DSS and Diffie-Hellman SChannel Cryptographic Provider)

We also provide a modulo exponentiation offload DLL that enables the Microsoft CSP to take advantage of the computational power of an nShield module without added security benefits. This is useful for interoperation with applications that do not allow the user to choose the CSP.

Note: Unlike the Microsoft CSPs, the nShield CSPs do not support the exporting of private keys.

You should not need to make any adjustments to your code in order to use the nShield CSPs. However, the nShield module is an asynchronous device capable of performing several operations at once. In order to achieve maximum performance from the module, structure your application in a multithreaded manner so that it can make several simultaneous requests to the CSP.

Figure 4. Microsoft CryptoAPI architecture

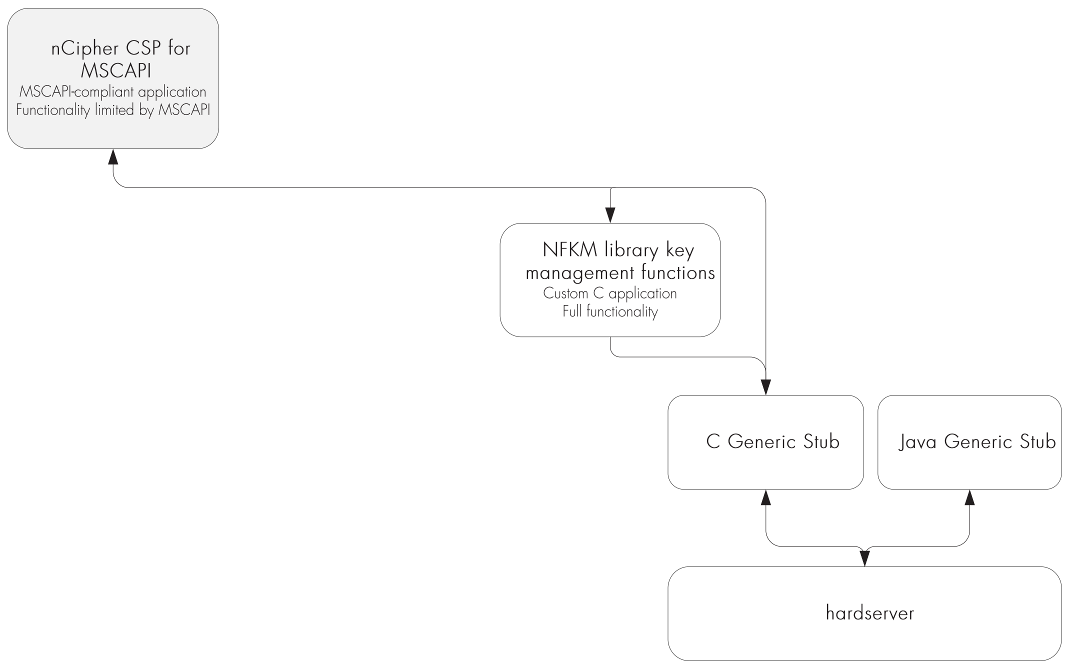


Figure 4 shows the way that the Microsoft CryptoAPI interface works with the nShield APIs.

Supported algorithms

The nShield CSPs support a similar range of algorithms to the Microsoft CSP.

Symmetric algorithms

- `CALG_DES`
- `CALG_3DES_112` (double-DES)
- `CALG_3DES`
- `CALG_RC4`
- `CALG_AES_128`
- `CALG_AES_192`
- `CALG_AES_256`

Asymmetric algorithms

- `CALC_RSA_SIGN` (only Enhanced RSA and AES Cryptographic Provider)
- `CALC_RSA_KEYX` (only Enhanced RSA and AES Cryptographic Provider)
- `CALC_DSA_SIGN` (only Enhanced DSS and Diffie-Hellman Cryptographic Provider and DSS Signature Cryptographic Provider)
- `CALC_DSS_SIGN` (only Enhanced DSS and Diffie-Hellman Cryptographic Provider)
- `CALC_DH_KEYX` (only Enhanced DSS and Diffie-Hellman Cryptographic Provider)
- `CALC_DH_SF` (only Enhanced DSS and Diffie-Hellman Cryptographic Provider)
- `CALC_DH_EPHEM` (only Enhanced DSS and Diffie-Hellman Cryptographic Provider)

Hash algorithms

- **`CALG_SHA1`**
- **`CALG_SHA256`**
- **`CALG_SHA384`**
- **`CALG_SHA512`**
- **`CALG_SSL3_SHAMD5`**
- **`CALG_MD5`**
- **`CALG_MAC`**
- **`CALG_HMAC`**

In addition, the Enhanced SChannel Cryptographic Provider and the Enhanced DSS and Diffie-Hellman SChannel Cryptographic Provider support all the internal algorithm types necessary for SSL3 and TLS1 support.

The nShield CSPs do not support SSL2.

Key generation and storage

The nShield CSP generates public/private key pairs (RSA, DSA, and Diffie-Hellman keys) in the module. The keys are stored in the Security World as protected by key blobs. (For details of the Security World, see the User Guide). Natively generated keys have **`mscapi`** as the **`appname`** and the hash of the key as the **`ident`**.

As in the Microsoft CSP, up to two keys are allowed for each container. Containers themselves are stored as opaque data in the Security World. Containers contain no key information but serve to associate NFKM keys with CSP containers, as well as storing other miscellaneous information. They have **`mscapi`** as the **`appname`** and **`container-`***containerID* as the **`ident`**, where *containerID* is calculated from a combination of the CSP name, the user's unique SID and the container name.

Note: The default permissions on new containers created by the nShield CSP have changed in order to solve a problem with IIS version 6: in this version of IIS it was possible to create containers with an empty ACL, such that they were completely inaccessible.

The previous default container permissions came from the inherited permissions on the **`NFAST_KMLOCAL`** directory, and had no non-inherited permissions. The default Security World Software installation gives everyone full control of the **`NFAST_KMLOCAL`** directory.

The current software sets an explicit ACL on new containers created by the CSP but does not alter permissions on previously created containers. The new permissions are as follows:

- **`READ`** access for **`EVERYONE`**
- **`FULL`** access for **`BUILTIN\Administrators`**
- for user containers: **`FULL`** access for the current user
- for machine containers: **`FULL`** access for **`LOCALSYSTEM`**

Note: No action is required on the user's part to invoke the new behavior.

Symmetric keys in the nShield CSP are generated and stored entirely in software. These keys are not hardware protected and are no more secure than the corresponding keys in the Microsoft CSP.

Note: The values of the **`KP_PERMISSIONS`** flags for hardware protected keys are enforced in software, except for **`CRYPT_EXPORTABLE`** which is ignored.

All CSP-generated, hardware-protected keys have ACLs that allow both signing and encryption. Hardware-protected keys that have been generated by the CSP are never exportable by the CSP; `CryptExportKey` always fails with a permissions error when called on such a key.

Container files and their associated key files can be moved freely between machines, as long as the user's SID is also valid on the destination machine. This is the case if the user in question is a domain user and both machines are on that domain. If the user's SID is not valid on the destination machine and keys are required to be shared between multiple machines, then the `cspimport` utility must be used to reassociate the Security World key file with the required destination container.

User interface issues

The nShield CSP supports hardware keys protected by either the module itself or by OCSs. Protecting keys with OCSs raises some user interface issues because the user interface needs to be displayed both at key-creation time and at key-loading time.

The choice of using module-protected keys or keys protected by OCSs is made in the install wizard. If, however, you generate keys protected by OCSs and then switch to module protection, then in most cases the keys protected by OCSs still require the user interface to be displayed in order to load them.

At key-generation time, if the `always display UI at key gen` flag is unset and an automatic Operator Card is present, the CSP uses the card set to protect the key, loading the shares automatically on all modules that contain a suitable card. (The flag is set using the install wizard.) Otherwise the CSP displays the user interface and blocks until the user interface is completed.

At key-loading time, if the key is protected by an automatic OCS, and the card set is present, then the key is loaded on all modules that contain a suitable card. Otherwise, the CSP displays the user interface and blocks until the user interface is completed; this requires the same steps as for key generation except for choosing the card set.

An automatic OCS means a card from a 1/*N* card set that is not protected by a pass phrase. At either time, the user interface is completed when the user has chosen a card set and the modules on which to load the key and has performed the card and pass phrase operations.

The CSP requires authorization to import keys (including public keys) and to generate keys when you have initialized your modules in the mode compatible with FIPS 140-2 level 3. This means that you must have a card from your current Security World in the slot when you attempt any of these operations, even if you are generating a module-protected key. If a card is not present, the operation blocks, and the CSP displays a user interface that prompts you to insert a card.

The CSP honors the `CRYPT_SILENT` flag to `CryptAcquireContext`. If this flag is passed in and the CSP would otherwise have to put up the user interface for any of the reasons in the two previous paragraphs, it fails with the appropriate error message.

If the CSP is being loaded from a service process (e.g. when used from within IIS or the main Certificate Authority process), then that process does not necessarily have access to the user's desktop. This means that any UI displayed by the CSP may not appear on an attended desktop (or at all), and the underlying operation may well time out.

If this is the case (and you are not using the **CRYPT_SILENT** flag, for whatever reason), we recommend that either you do not use OCS-protected keys or you use an automatic card set, so that the CSP does not display the UI.

Key counting

The nShield CSP supports the **PP_CRYPT_COUNT_KEY_USE** parameter to CryptAcquireContext as long as the module with NVRAM is attached. Setting this parameter to a nonzero value causes all keys generated from that point to have nonvolatile use counters. The counter persists until CryptReleaseContext is called or until the **PP_CRYPT_COUNT_KEY_USE** parameter is reset to **0**.

Note: Key counting is not directly supported by end-user applications such as IIS . It is only supported by Microsoft Certificate Services under Windows 2003 and later. However, it is possible to create a certificate that uses a key counter in cases where key counting is not directly supported. For more information about key counting, see the User Guide.

Keys that have counters can only be loaded on one module at a time. The key-generation and key-loading functions enforce this behavior. When you generate these keys, you must present your Administrator Cards in order to authorize the creation of the new NVRAM area.

Note: You must not insert your Administrator Cards in an untrusted host.

To minimize the exposure of the Security Officer root key ($\mathbf{K}_{NSO}$) when you generate a key with key counting enabled, you should create the Security World with an NVRAM delegation key that requires the presentation of fewer Administrative Cards than are required to load $\mathbf{K}_{NSO}$.

If you reinitialize your module for any reason, all the NVRAM areas on that module are erased. You must then use **cspnvfix** to recreate the NVRAM areas for all the keys that have counters.

NVRAM-stored keys

The nShield CSP now supports creating keys protected by the module NVRAM. The **PP_NO_HOST_STORAGE** parameter to **CryptAcquireContext** is supported as long as the module with NVRAM is attached. Setting this parameter to a nonzero value causes all keys generated from that point to be generated with blobs in NVRAM. The counter persists until CryptReleaseContext is called or until the **PP_NO_HOST_STORAGE** parameter is reset to **0**.

The method of creating NVRAM-stored keys is very similar to the method of creating keys with NVRAM counters:

1. call **CryptAcquireContext** to get a handle to a container.
2. call **CryptSetProvParam** and set the **PP_NO_HOST_STORAGE** property to a non-zero value.

This causes any keys generated with that container handle to be generated with blobs in NVRAM until either of the following occurs:

- **CryptReleaseContext** is called with that container handle
- **CryptSetProvParam** is called to set **PP_NO_HOST_STORAGE** to zero

Creating NVRAM-stored keys requires insertion of the ACS quorum for NVRAM, in the same way as creating key counted keys.

`PP_NO_HOST_STORAGE` is a new value and will be set in the **`wincrypt.h`** header file in future versions of the Microsoft Platform SDK. The following example code can be used until then to define the value correctly:

```
#ifndef PP_NO_HOST_STORAGE
#define PP_NO_HOST_STORAGE 44
#endif
```

This feature is only available to users writing CAPI code directly. To use a NVRAM-stored key in a client application (for example IIS or the Microsoft Certificate Authority), first create the key with the **`keytst`** command-line tool, and then transfer the key across to the required container with the **`cspimport`** utility.

Also, the **`keytst`** and **`csptest`** utilities have gained an extra command-line parameter. **`keytst --help`** now gives output containing the following information:

```
Key creation flags (only valid with -cx or -cs):
-e, --export                Create the key(s) with the 'exportable' bit set.
  -L, --length=BITLEN       Specify the new key length (default = 1024).
  -C, --counter             Create key counters (if supported).
  -K, --kitb                Create NVRAM-stored key(s) (if supported).
```

Note: The **`-C`** and **`-K`** options require you to insert your ACS.

The command **`csptest --help`** outputs the following usage message:

```
Program options:
  -f, --flood               Run a continuous signature test.
  -d, --dsa                 Use DSA signatures rather than RSA signatures.
  -m, --ms                  Use the MS AES provider rather than nCipher's one
                              (possibly with modexp offload).
  -C, --counters            Generate keys with counters (needs NVRAM and ACS).
  -K, --kitb                Generate keys using KITB (needs NVRAM and ACS).
```

The **`csputils`** utility displays the NVRAM status of keys using the **`--detail`** option.

CSP setup and utilities

Thales provides a CSP installation wizard that creates a new Security World, loads an existing Security World, or sets up the modexp offload DLL. The CSP installation wizard also generates new OCSs and the set-up parameters of the CSP. However, the installation wizard is not suitable for complex Security World setups. If you require more flexibility than the CSP install wizard provides, use **`new-world`** and **`createocs`**, or **`KeySafe`**, to create your Security World.

The standard Security World utility **`nfkmverify`** should be used to check the security of all stored keys in the Security World; **`nfkminfo`**, **`nfkmcheck`** and other standard utilities can also be used to assist in this process.

Additionally, Thales provides some CSP-specific command-line utilities:

- `csputils` provides an overview of the containers and keys present and also tells you the values of the counters for key-counted keys
- `cspcheck` is for use alongside `nfkmcheck`
- `cspimport` allows you to move keys between containers or to import a pre-generated NFKM key into a container
- `cspnvfix` allows you to regenerate NVRAM areas in modules where these have been erased (for example, by reinitialization)
- `csptest` is a general test utility that can be used to list the capabilities of installed nShield and Microsoft CSPs or to perform a soak test
- `keytst` allows you to generate containers and keys and also to list the available containers.

For more information about these utilities, see the User Guide.

Chapter 6: Microsoft CNG

Cryptography API: Next Generation (CNG) is the successor to the Microsoft Crypto API (CAPI) and its long-term replacement. The Security World Software implementation of Microsoft CNG is supported on Microsoft Windows Windows Server 2008 (for both x86 and x64 architectures) and later releases, including Windows Server 2012. The nShield CNG providers offer the benefits of hardware-based encryption accessed through the standard Microsoft API, and support the National Security Agency (NSA) classified Suite B algorithms.

Before using the nShield CNG providers, run the nShield CNG Configuration Wizard to:

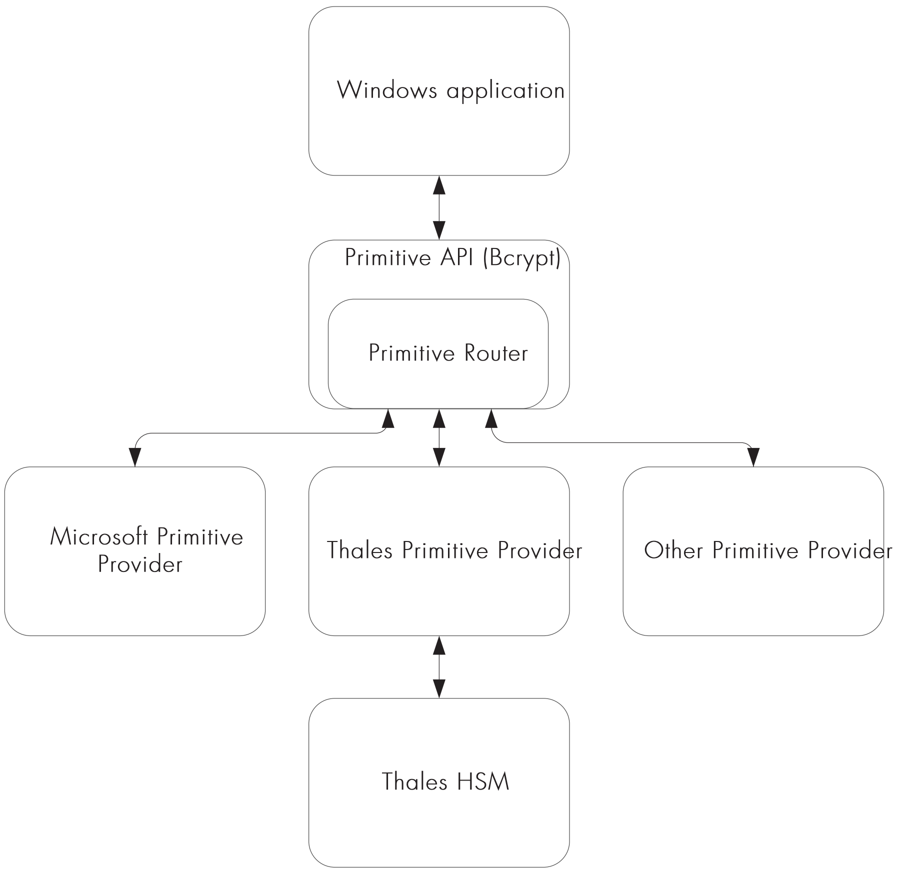
- create a new Security World or specify an existing Security World to use
- register the nShield CNG providers
- configure the nShield CNG providers as default CNG providers for specific tasks.

This chapter describes the features and implementation details of the nShield CNG providers. For more information, see the Microsoft CNG documentation: http://msdn2.microsoft.com/en-us/library/aa376210.aspx.

CNG architecture overview

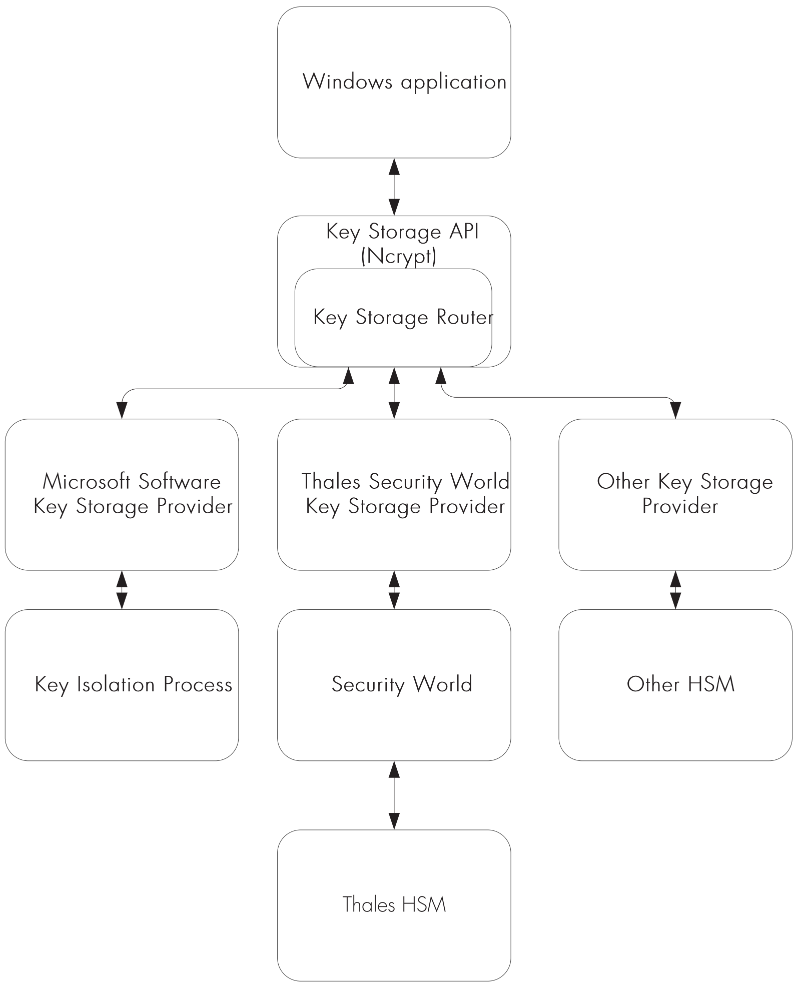
CNG handles cryptographic primitives and key storage through separate APIs. In both cases a Windows application contacts a router, which forwards the cryptographic operation to the provider that is configured to handle the request. For an illustration of communication between the architecture layers for cryptographic primitives, see Figure 5.

Figure 5. CNG primitives



For an illustration of communication between the architecture layers for cryptographic key storage, see Figure 6.

Figure 6. Key storage



Supported algorithms for CNG

This section lists the National Security Agency (NSA) classified Suite B algorithms supported by the nShield CNG providers.

Note: The MQV algorithm is not supported by the nShield CNG providers.

Signature interfaces (key signing)

Interface name	*Type of support*
RSA PKCS#1 v1	Hardware
RSA PSS	Hardware
DSA	Hardware
ECDSA_P224	Hardware
ECDSA_P256	Hardware
ECDSA_P384	Hardware
ECDSA_P521	Hardware

Note: Hashes used with ECDSA must be of the same length or shorter than the curve itself. If you attempt to use a hash longer than the curve the operation returns **`NOT_SUPPORTED`**. In FIPS 140-2 level 3 Security Worlds ECDSA signing is only supported where the length of the curve is approximately the length of the hash. See *Key authorization for CNG* on page 86.

Hashes

Hash name	*Type of support*
SHA1	Hardware (HMAC only)/software
SHA256	Hardware (HMAC only)/software
SHA384	Hardware (HMAC only)/software
SHA512	Hardware (HMAC only)/software
SHA224	Hardware (HMAC only, requires firmware version 2.33.60 or later)/software
MD5	Hardware (HMAC only)/software

Note: MD5 is not supported in FIPS 140-2 mode.

Asymmetric encryption

Algorithm name	*Type of support*
RSA Raw (NCRYPT_NO_PADDING_FLAG)	Hardware
RSA PKCS#1 v1 (NCRYPT_PAD_PKCS1_FLAG)	Hardware
RSA OAEP (NCRYPT_PAD_OAEP_FLAG)	Hardware

Symmetric encryption

Algorithm name	*Type of support*
RC2 ECB, CBC	Hardware and software
RC4	Hardware and Software (not supported in FIPS 140-2 level 3 mode)
AES ECB,CBC	Hardware and Software
DES ECB,CBC	Hardware and Software (DES is not supported in FIPS 140-2 level 3 mode)
3DES ECB,CBC	Hardware and Software
3DES_112 ECB,CBC	Hardware and Software

Key exchange

Protocol name	*Type of support*
DH	Hardware
ECDH_P224	Hardware
ECDH_P256	Hardware
ECDH_P348	Hardware
ECDH_P521	Hardware

Note: Elliptic curve cryptography algorithms must be enabled before use. Use the `fet` command-line utility with an appropriate certificate to enable a purchased feature. If you enable the elliptic curve feature on your modules after you first register the CNG providers, you must run the configuration wizard again for the elliptic curve algorithm providers to be registered. For more information about registering the CNG providers, see *Registering the CNG CSP* on page 93.

Random Number Generation

Name	*Type of support*
RNG	Hardware

Key authorization for CNG

When an application needs keys that are protected by a smart card or a logical token, a user interface is invoked to prompt the application user to insert the smart card or enter appropriate pass phrases.

Note: The user interface prompt is not provided if your application is working in silent mode. The nShield CNG providers attempt to load the required authorization (for example, from an Operator Card that has already been inserted) but fail if no authorization can be found. For more information about silent mode, refer to the documentation of the CNG Key Storage Functions at: http://msdn2.microsoft.com/en-us/library/aa376208.aspx.

Note: When the CNG application is running in Session 0 (i.e. loaded by a Windows service), the user interface is provided by an agent process **nShield Service Agent** that is started when the user logs in. This agent, when running, is shown in the Windows System Tray. All user interaction requests from a CNG application running in Session 0 cause dialogs to be raised by the agent allowing the user to select cardsets, modules and enter passphrases. The interaction with the user is functionally identical to that described in this section.

There can only be one instance of the agent running. Attempts to start a second instance will fail with a **CreateNamedPipe** error. If the agent is not running, attempts to invoke dialogs through it will fail and this is logged in the Windows Event Log. It can be restarted by logging off and on or by explicitly executing either **`%NFAST_HOME%\bin\nShield_service_agent64.exe`** or **`%NFAST_HOME%\bin\nShield_service_agent.exe`**. On 64 bit platforms either of these can be used irrespective of the bit size of the underlying application.

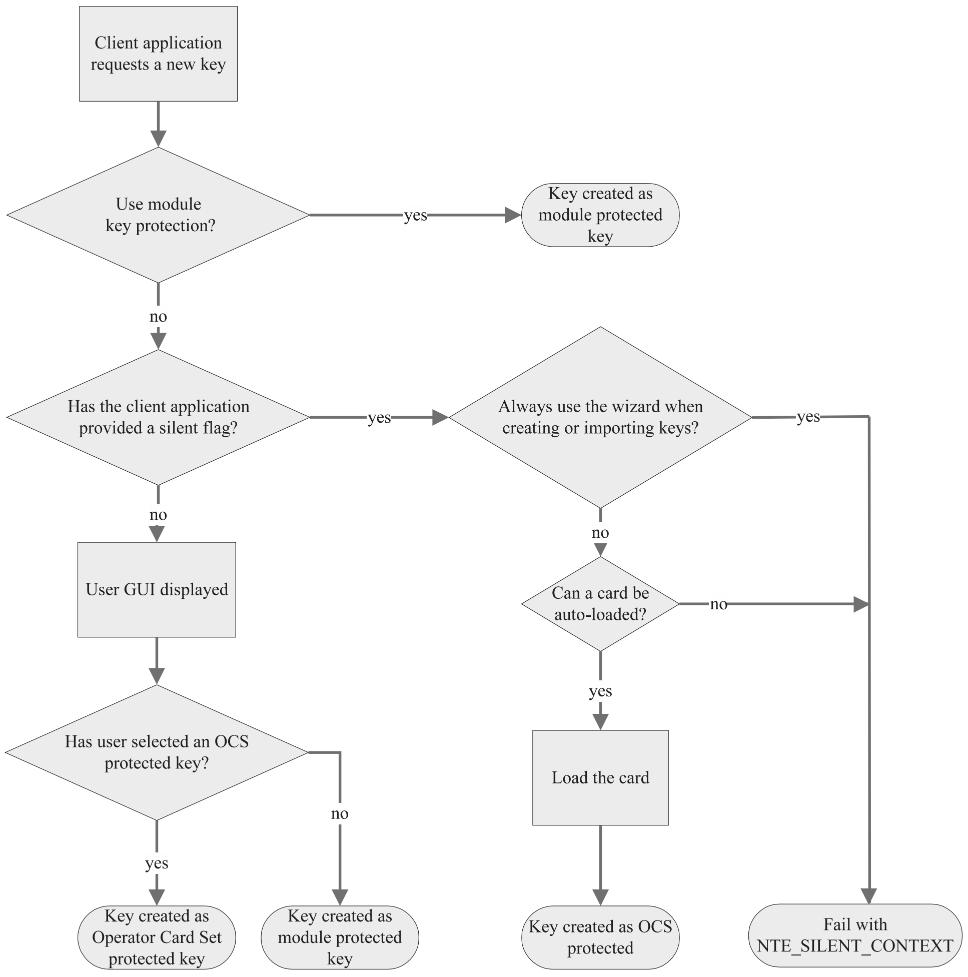
For more information about auto-loadable card sets and the considerations of silent mode, see Figure 7.

You define key protection and authorization settings with the CNG Configuration Wizard on the **Key Protection Setup** screen. For more information about the CNG Configuration Wizard, see *Configuring the nCipher CNG CSP* on page 92.

The options on this screen that are relevant to key protection and authorization are:

- **Module protection**
  Select this option to make keys module protected.
- **Operator Card Set protection**
  Select this option to display a user interface, at the point of key generation, that allows you to choose:
  - between module protection and OCS protection
  - which Operator Cards to use for the protection.
- **Always use the wizard when creating or importing keys**
  This option is only enabled when OCS protection is selected. Select this option to suppress the auto-load of authorization when working in silent mode. For more information about how key authorization requests are processed by the nShield CNG providers, see Figure 7.
  Note: The nShield GUI is never enabled for calls with a valid `Silent` option. Although the nShield CNG providers always attempt to obtain necessary authorization (for example, from an Operator Card that has already been inserted), they fail any operation for which they are unable to obtain the required authorization.

Figure 7. Key authorization requests


Client application requests a new key
Use module key protection?
yes
Key created as module protected key
no
Has the client application provided a silent flag?
yes
Always use the wizard when creating or importing keys?
yes
no
User GUI displayed
no
Can a card be auto-loaded?
no
yes
Load the card
Has user selected an OCS protected key?
no
yes
Key created as Operator Card Set protected key
Key created as module protected key
Key created as OCS protected
Fail with NTE_SILENT_CONTEXT

Note: FIPS 140-2 level 3 environments always require card authorization for key creation. When using the CNG Primitive Functions the user is not prompted to provide card authorization, but the request fails if no card is provided.

The key storage providers always respect calls made with the **`Silent`** option. Primitive providers never display a user interface.

Applications may have a mechanism to disable silent mode operation, thereby allowing appropriate pass phrases to be entered. Ensure that you configure applications to use an appropriate level of key protection. For example, in Microsoft Certificate Services, you must select the **Use strong private key protection features provided by the CSP** option to disable silent mode operation.

Key use counting

You can configure the CNG provider to count the number of times a key is used. Use this functionality, for example, to retire a key after a set number of uses, or for auditing purposes.

To enable key use counting in the Security World Key Storage Provider, call **`NCryptSetProperty`** with **`NCRYPT_USE_COUNT_ENABLED_PROPERTY`** on the provider handle. Alternatively, to override the behavior of third-party software that would not otherwise provide the user with the option to enable key use counting, use one of the following methods:

- set the environment variable **`NCCNG_USE_COUNT_ENABLED`** to **`1`**
- set the registry key **`Software\nCipher\CryptoNG\UseCountEnabled`** to **`1`**

Keys created while the provider has key use counting enabled continue to have their use counts incremented, regardless of the state of the provider's handle. Key use counts are not recorded for keys created while the **`NCRYPT_USE_COUNT_ENABLED_PROPERTY`** is disabled on the provider handle.

Because the key counter is a 64-bit area in a specific module's NVRAM, the counted keys are specific to a single module. When a key is created you are prompted to specify which module to use, unless there is only one module in the Security World, or **`preload`** was used to preload authorization from an ACS on only one module.

The key counter is incremented each time a private key is used to:

- sign
- decrypt
- negotiate a secret agreement.

To test the performance of keys with counters, run the **`cngsoak`** command with the **`-C`** option:

```
cngsoak -C --sign --length=1024
```

To view the current key use count for keys, run the **`cnglist`** command with the **`--list-keys`** and **`--verbose`** options:

```
cnglist --list-keys --verbose
```

Migrating keys for CNG

We provide functionality for migrating existing keys from other providers into the Security World Key Storage Provider. To identify installed providers, run the command:

```
cnglist --list-providers
```

To identify the keys that are available from a particular provider, run the command:

```
cnglist --list-keys --provider="ProviderName"
```

In this command, *ProviderName* is the name of the provider.

The following command provides an example of identifying keys from the Security World Key Storage Provider:

```
cnglist --list-keys --provider="nCipher Security World Key Storage Provider"
MyApp Personal Data Key: RSA
CertReq-5eb45f6d-6798-472f-b668-288bc5d961da: ECDSA_P256 machine
WebServer Signing Key: DSA machine
ADCS-Root-Key: ECDSA_P521 machine
```

Note: To list the keys available from the Security World Key Storage Provider, run the command **`cnglist --list-keys`** (without specifying the **`--provider`** option) .

Importing a Microsoft CAPI key into the Security World Key Storage Provider

To import a Microsoft CAPI key into the Security World Key Storage Provider, first run the CAPI utility **`csputils`** to identify the existing CAPI containers and their key contents.

CAPI containers can contain either a signing key or a key exchange key, or both. The following example shows how to import both a signing key and a key exchange key from a Microsoft CAPI container:

```
cngimport -m --csp="Microsoft Strong Cryptographic Provider"
  -k "EXAMPLE_CAPICONTAINER"
     "EXAMPLE_IMPORTED_SIGNATURE_CAPICONTAINER"
     "EXAMPLE_IMPORTED_KEYEXCHANGE_CAPICONTAINER"
```

To check the success of the import, list the keys present in the Security World Key Storage Provider:

```
cnglist --list-keys
EXAMPLE_IMPORTED_SIGNATURE_CAPICONTAINER: RSA
EXAMPLE_IMPORTED_KEYEXCHANGE_CAPICONTAINER: DH
```

The following example command shows how to import a single signing key:

```
cngimport -m -s --csp="Microsoft Strong Cryptographic Provider"
  --key="EXAMPLE_CAPICONTAINER"
        "EXAMPLE_IMPORTED_SIGNATURE_ONLY_CAPICONTAINER"
```

Run the **`cnglist`** command with the **`--list-keys`** option to check the success of the key import:

```
cnglist --list-keys
EXAMPLE_IMPORTED_SIGNATURE_ONLY_CAPICONTAINER: RSA
```

Note: The **`cngimport`** option **`-m/--migrate`** cannot be used to migrate nShield CAPI container keys to CNG. For information about importing nShield CAPI container keys into CNG, see *Importing a Microsoft CNG key into the Security World Key Storage Provider* on page 90.

Importing a Microsoft CNG key into the Security World Key Storage Provider

To import a Microsoft CNG key into the Security World Key Storage Provider, run the **`cngimport`** command as shown in the following example:

```
cngimport -m
  -k "EXAMPLE_RSA_1024"
     "IMPORTED_RSA_1024"
```

Run the **`cnglist`** command to confirm that the key has been successfully imported:

```
cnglist --list-keys
  IMPORTED_RSA_1024: RSA
```

The original key is not deleted from the provider from which it was imported:

```
cnglist --list-keys --provider="Microsoft Software Key Storage Provider"
  EXAMPLE_RSA_1024
```

Note: Certain applications, such as Certificate Services, create keys using the Microsoft Software Key Storage Provider which cannot be exported. Attempting to import such a key into the nCipher provider results in the following message:

```
cngimport -m -k WIN-KQ1Z6JMCUTB-CA WIN-ncipher-CA
Unable to continue. This key can not be exported from Microsoft Software Key Storage
Provider.
```

Importing a Security World key into the Security World Key Storage Provider

To import a Security World key into the Security World Key Storage Provider, run the **`cngimport`** utility as shown in the following example:

```
cngimport --import --key=nfkmsimple1 --appname=simple nfkmsimple1
Found key 'nfkmsimple1'
Importing NFKM key.. done
```

Run **`cnglist`** with the **`--list-keys`** option to confirm that the key has been successfully imported:

```
cnglist --list-keys
nfkmsimple1: RSA
```

To import an nShield CAPI container into the Security World Key Storage Provider, run the **`csputils`** command to identify the container name:

```
csputils -l
File ID    Container name       Container owner      DLL name   S X
=========  ===================  ===================  =========  = =
31e994f07  CONTAINER2           SYWELL\Administrato  ncsp       * *
3a2b082a8  CAPICONTAINER        SYWELL\Administrato  ncsp       * *
2 containers and 4 keys found.
```

Note: Run the **`csputils`** command with the **`-l`** and **`-m`** options to migrate an nShield CAPI machine container.

Identify the Security World key names of the keys in the container by running the **`csputils`** command as follows:

```
csputils -d -n CAPICONTAINER
Detailed report for container ID #3a2b082a8f2ee1a5acb756d5e95b09817072807a
Filename:       key_mscapi_container-3a2b082a8f2ee1a5acb756d5e95b09817072807a
Container name: CAPICONTAINER
User name:      SYWELL\Administrator
User SID:       s-1-5-21-352906761-2625708315-3490211485-500
CSP DLL name:   ncsp.dll
Filename for signature key is key_mscapi_ce51a0ee0ea164b993d1edcbf639f2be62c53222
   Key was generated by the CSP
   Key hash:    ce51a0ee0ea164b993d1edcbf639f2be62c53222
   Key is recoverable.
   Key is cardset protected.
      Cardset name:             nopin
      Sharing parameters:       1 of 1 shares required.
      Cardset hash:             d45b30e7b60cb226f5ade5b54f536bc1cc465fa4
      Cardset is non-persistent.
Filename for key exchange key is key_mscapi_dbd84e8155e144c59cf8797d16e7f8bd19ac446a
   Key was generated by the CSP
   Key hash:    dbd84e8155e144c59cf8797d16e7f8bd19ac446a
   Key is recoverable.
   Key is cardset protected.
      Cardset name:             nopin
      Sharing parameters:       1 of 1 shares required.
      Cardset hash:             d45b30e7b60cb226f5ade5b54f536bc1cc465fa4
      Cardset is non-persistent.
1 container and 2 keys found.
```

The key name to pass to the **`cngimport`** command **`--key`** option is the part of the key name that follows **`key_mscapi_`** in the output line that starts **`Filename for signature key is key_mscapi_`**.

For example, the signature key file name for **`CAPICONTAINER`** in the example shown above is **`key_mscapi_ce51a0ee0ea164b993d1edcbf639f2be62c53222`**, so **`ce51a0ee0ea164b993d1edcbf639f2be62c53222`** is the key name that should be passed to **`cngimport`**:

```
cngimport --import --key="ce51a0ee0ea164b993d1edcbf639f2be62c53222" --
appname="mscapi" Signature_Key_Imported_From_nCipher_CAPI
Found unnamed key
Importing NFKM key.. done
```

Run **`cnglist`** with the **`--list-keys`** option to confirm that the key has been successfully imported:

```
cnglist --list-keys
Signature_Key_Imported_From_nCipher_CAPI: RSA
cngsoak: ECDH_P256
```

Follow the same procedure for importing the key exchange key from the nShield CAPI container.

Configuring the nCipher CNG CSP

The DLL files that support the nCipher CNG CSP are installed during product installation. However, you need to register the nCipher CNG CSP before you can use it.

You can unregister the nCipher CNG CSP without removing the provider DLL files from your system. After unregistering, you can reregister the nCipher CNG CSP at any time as long as the files have not been uninstalled from your system. For more information, see *Unregistering or reregistering the CNG CSP* on page 94.

You can completely uninstall the nCipher CNG CSP, removing the files from your system. After uninstalling, you must reinstall the files and then reregister the nCipher CNG CSP before you can use it. For more information, see *Uninstalling or reinstalling the CNG CSP* on page 95.

Registering the CNG CSP

You can register the nCipher CNG CSP with:

- CNG Configuration Wizard
- the **`cngregister`** command-line utility

To register the nCipher CNG CSP, the hardserver must be running and able to communicate with at least one module. This requirement is normally fulfilled during the product installation process. You can check that this requirement is fulfilled by running the **`enquiry`** command-line utility and checking the output for details about the module. For more information, see the User Guide.

Registering the CNG CSP with the CNG Configuration Wizard

We recommend using the CNG Configuration Wizard to register the nCipher CNG CSP. The product installation process places a shortcut to the CNG Configuration Wizard in the Start menu under **`All Programs`** > **`nCipher`**.

Note: You can also use the CNG Configuration Wizard to create new Security Worlds, load existing Security Worlds, generate new OCSs, and configure the set-up parameters of the CNG CSP. For more information, see the User Guide.

To register the CNG CSP with the CNG Configuration Wizard, you must have already created a Security World and chosen a key protection method, either module-protection or OCS-protection. If you chose OCS-protection, you must also have already created an OCS before you can register the nCipher CNG CSP with the CNG Configuration Wizard.

Note: The CNG Configuration Wizard is not suitable for creating complex Security World setups or for creating Security Worlds with the unit. For information about other methods of creating Security Worlds, see the User Guide.

If you use the CNG Configuration Wizard to create a Security World (and, if appropriate, an OCS), the wizard automatically prompts you to register the CNG CSP after you have fulfilled the necessary prerequisites.

You can also use the CNG Configuration Wizard to change an existing configuration at any time by running the wizard as usual and choosing the **`Use the existing security world`** option on the **`Initial setup`** screen.

To register the CNG CSP with the CNG Configuration Wizard after the necessary key-protection prerequisites have been fulfilled:

1. If the wizard is not already running:
   a. Run the wizard by double-clicking its shortcut in the Start menu under **`All Programs`** > **`nCipher`**.
      The wizard displays the welcome window:
      Note: If any module is in the maintenance mode, the wizard displays a warning. In such a case, reset the module into the initialization mode, and rerun the wizard.
   b. Click the **`Next`** button.
      If the prerequisite to create a Security World has been fulfilled, the wizard displays a confirmation screen.
   c. Click the **`Next`** button.
      The wizard displays a screen confirming that your Security World and (if you chose to create an OCS) an OCS have been created:
      Note: If you chose module-protection for your keys, the wizard does not confirm that an OCS has been created.
2. When the wizard has confirmed that it is ready to register the nCipher CNG providers, click the **`Next`** button.
   The wizard registers the nCipher CNG CSP.
   Note: You cannot use the CNG Configuration Wizard to configure the nCipher CNG providers for use as defaults. We recommend that you always use the nCipher CNG providers by selecting them directly with the application that is using CNG.
   When configuration of your nCipher CNG CSP is complete, the wizard displays the confirmation screen.

Registering the CNG CSP with cngregister

You can use the **`cngregister`** command-line utility to register the nCipher CNG CSP manually even if you have not already created a Security World (or, if you choose OCS-protection for your keys, even if you have not already created an OCS).

To register the nCipher CNG CSP with the **`cngregister`** command-line utility, run the command without specifying any options:

```
cngregister
```

Note: You cannot use the **`cngregister`** command-line utility to configure the nCipher CNG providers for use as defaults. we recommend that you always use the nCipher CNG providers by selecting them directly with the application that is using CNG.

For more information about the **`cngregister`** command-line utility, see *cngregister* on page 98.

Unregistering or reregistering the CNG CSP

You can use the **`cngregister`** command-line utility to unregister or reregister the nCipher CNG CSP manually.

To unregister the nCipher CNG CSP, run the command:

```
cngregister -U
```

This command unregisters the CNG CSP, but does not remove the provider DLL files from your system. For information about removing these files, see *Uninstalling or reinstalling the CNG CSP* on page 95.

If any applications or services are using the nCipher CNG providers for key storage or cryptography, unregistering the nCipher CNG CSP can cause system instability.

After unregistering the nCipher CNG CSP, you can reregister it at any time as long as the files have not been uninstalled from your system. To reregister the nCipher CNG CSP on your system, run the command:

```
cngregister
```

Note: You cannot use the **`cngregister`** command-line utility to configure the nCipher CNG providers for use as defaults. We recommend that you always use the nCipher CNG providers by selecting them directly with the application that is using CNG.

For more information about these command-line utilities, see *Utilities for CNG* on page 96.

Uninstalling or reinstalling the CNG CSP

To uninstall the nCipher CNG CSP:

1. To remove any and all dependencies that you have set, run the command:

   ```
   ncsvcdep -x
   ```

   Note: Always run **`ncsvcdep`** as a user with full administrative privileges.
2. Unregister the nCipher CNG CSP on your system by running the command:

   ```
   cngregister -U
   ```

   This command unregisters the CNG CSP, but does not remove the provider DLL files from your system.

3. Uninstall the nCipher CNG DLLs from your system:
   - On 32-bit versions of Windows, run the command:

     ```
     cnginstall -U
     ```

   - On 64-bit versions of Windows, run the command:

     ```
     cnginstall64 -U
     ```

To reinstall the nCipher CNG CSP after you have previously uninstalled it:

1. Reinstall the nCipher CNG CSP files on your system:
   - On 32-bit versions of Windows, run the command:

     ```
     cnginstall -i
     ```

   - On 64-bit versions of Windows, run the command:

     ```
     cnginstall64 -i
     ```

2. Reregister the nCipher CNG CSP on your system by running the command:

   ```
   cngregister
   ```

For more information about these command-line utilities, see *Utilities for CNG* on page 96.

Utilities for CNG

Use the **`nfkmverify`** command-line utility to check the security of all stored keys in the Security World. Use **`nfkminfo`**, **`nfkmcheck`**, and other command-line utilities to assist in this process. For more information about these command-line utilities, see the User Guide.

The following table lists the utilities specific to the nCipher CNG CSP:

x86	x64	Utility description
`cngimport.exe`	**`cngimport.exe`**	This key migration utility is used to migrate Security World, CAPI, and CNG keys to the Security World Key Storage Provider.

x86	x64	Utility description
`cnginstall.exe`	`cnginstall64.exe`	This utility is the nCipher CNG CSP installer. Only use this utility to remove or reinstall the provider DLLs and associated registry entries manually.
`cnglist.exe`	`cnglist.exe`	This utility lists information about CNG CSP.
`cngregister.exe`	`cngregister.exe`	This is the nCipher CNG CSP registration utility. You can use it to unregister and re-register the nCipher providers manually.
`cngsoak.exe`	`cngsoak64.exe`	This utility is the nCipher CNG soak tool. You can use it to evaluate the performance of signing, key exchange, and key generation using a user-defined number of threads.
`ncsvcdep.exe`	`ncsvcdep.exe`	This utility is the service dependency tool. You can configure some service based applications, such as Microsoft Certificate Services and IIS, to use the nCipher CNG CSP. The nShield Service dependency tool allows you to add the nFast Server to the dependency list of such services.

These utilities are located in the **`bin`** directory of your Security World Software installation (for example, *%NFAST_HOME%*`\bin`).

Note: On 64-bit versions of Windows, both the 32-bit and 64-bit versions of the listed utilities are installed. When working on an 64-bit version of Windows, always ensure that you use the 64-bit version of the utility (if one is available).

cngimport

Use **`cngimport`** to migrate keys to the Security World Key Storage Provider. For more information, see *Migrating keys for CNG* on page 89.

cnginstall

The **`cnginstall`** utility is used by the Security World Software installation wizard. You can also use this utility to manually uninstall (or reinstall) the nCipher CNG DLLs and registry entries.

To uninstall the nCipher CNG DLL files, run the command:

```
cnginstall -U
```

This command removes the provider DLL files from your system. It produces output of the form:

```
ncksppt.dll removed.
ncksρsw.dll removed.
ncpp.dll removed.
```

Before you uninstall the nCipher CNG DLL files, ensure that you unregister the nCipher CNG CSP. For more information, see:

- *cngregister* on page 98
- *Unregistering or reregistering the CNG CSP* on page 94.

After unregistering the nCipher CNG CSP, you can reregister it at any time as long as the files have not been uninstalled from your system. To reregister the nCipher CNG CSP on your system, run the command:

```
cngregister
```

For more information about uninstalling and reinstalling the nCipher CNG CSP with **`cnginstall`**, see *Uninstalling or reinstalling the CNG CSP* on page 95.

cngregister

Use **`cngregister`** to unregister the nCipher CNG CSP manually.

To unregister the nCipher CNG CSP, run the command:

```
cngregister -U
```

This command produces output for the form:

```
Unregistered provider 'nCipher Primitive Provider'
Unregistered provider 'nCipher Security World Key Storage Provider'
```

This command unregisters the CNG CSP, but does not remove the provider DLL files from your system. For information about removing these files, see:

- *cnginstall* on page 97
- *Uninstalling or reinstalling the CNG CSP* on page 95.

If any applications or services are using the nCipher CNG CSP for key storage or cryptography, unregistering it can cause system instability.

After unregistering the nCipher CNG CSP, you can reregister it at any time as long as the files have not been uninstalled from your system. To reregister the nCipher CNG CSP on your system, run the command:

```
cngregister
```

Note: You cannot use the **`cngregister`** command-line utility to configure the nCipher CNG providers for use as defaults. We recommend that you always use the nCipher CNG providers by selecting them directly with the application that is using CNG.

cngsoak

Use **cngsoak** to obtain statistics about the performance of the nCipher CNG CSP. Specifically, use **cngsoak** to determine the speed of:

- signing a hash (**cngsoak --sign**)
- encryption (**cngsoak --encrypt**)
- key exchange (**cngsoak --keyx**)
- key generation (**cngsoak --generate**).

The output from **cngsoak** displays information as columns of information. From left to right, these columns display:

- the time in second that **cngsoak** has been running
- the total number of operations completed
- the number of operations completed in last second
- the average number of operations completed each second.

ncsvcdep

Use the **ncsvcdep** utility to ensure that the nShield **nFast Server** service is running before certain services are enabled. For example, Active Directory Certificate Services or Internet Information Services require that the hardserver is running in order to use the nCipher CNG CSP. Failure to set this dependency can lead to system instability.

To list installed services, run the **ncsvcdep** command with the **-l** option:

```
ncsvcdep -l
```

Output from this command has the form:

```
Installed Services (Count - "Display Name" - "Service Name")
0 - "Application Experience" - "AeLookupSvc"
1 - "Application Layer Gateway Service" - "ALG"
2 - "Application Information" - "Appinfo"
3 - "Application Management" - "AppMgmt"
4 - "Windows Audio Endpoint Builder" - "AudioEndpointBuilder"
.
.
108 - "nFast Server" - "nFast Server"
109 - "Active Directory Certificate Services" - "CertSvc"
```

Note: Always run **ncsvcdep** as a user with full administrative privileges.

To set a dependency, run the command:

```
ncsvcdep -a "DependentService"
```

In this command, *DependentService* is the service that has the dependency. The following example shows how to make the Active Directory Certificate Services dependent on the nFast Server:

```
ncsvcdep -a "CertSvc"
Dependency change succeeded.
```

To remove a specific dependency relationship, run **`ncsvcdep`** with the **`-r`** option, for example:

```
ncsvcdep -r "CertSvc"
Dependency change succeeded.
```

To remove all dependencies, run **`ncsvcdep`** with the **`-x`** option:

```
ncsvcdep -x
```

Note: Microsoft Certificate Services require that the **`certsvc`** service is made dependent on the hardserver.

Note: Microsoft Internet Information Services require that the **`http`** service is made dependent on the hardserver.

cnglist

Use **`cnglist`** to display details of CNG providers, keys, and algorithms.

To list details of the CNG providers, run the **`cnglist`** command with the **`--list-providers`** option:

```
cnglist --list-providers
```

Output from this command is of the form:

```
Microsoft Primitive Provider
Microsoft Smart Card Key Storage Provider
Microsoft Software Key Storage Provider
Microsoft SSL Protocol Provider
nCipher Primitive Provider
nCipher Security World Key Storage Provider
```

To list details of the algorithms, run the **`cnglist`** command with the **`--list-algorithms`** option:

```
cnglist --list-algorithms
```

Output from this command has the form:

```
BCryptEnumAlgorithms(BCRYPT_CIPHER_OPERATION):
   Name                           Class      Flags
   AES                            0x00000001 0x0
   RC2                            0x00000001 0x0
   RC4                            0x00000001 0x0
   DES                            0x00000001 0x0
   DESX                           0x00000001 0x0
   3DES                           0x00000001 0x0
   3DES_112                       0x00000001 0x0
BCryptEnumAlgorithms(BCRYPT_HASH_OPERATION):
   Name                           Class      Flags
   SHA1                           0x00000002 0x0
   MD2                            0x00000002 0x0
   MD4                            0x00000002 0x0
   MD5                            0x00000002 0x0
   SHA256                         0x00000002 0x0
   SHA384                         0x00000002 0x0
   SHA512                         0x00000002 0x0
   AES-GMAC                       0x00000002 0x0
   SHA224                         0x00000002 0x0
BCryptEnumAlgorithms(BCRYPT_ASYMMETRIC_ENCRYPTION_OPERATION):
   Name                           Class      Flags
   RSA                            0x00000003 0x0
```

To list details of the algorithms for the Security World Key Storage Provider, run the **cnglist** command with the **--list-algorithms**, **--keystorage**, and **--nc** options:

```
cnglist --list-algorithms --keystorage --nc
```

Output from this command has the form:

```
NCryptEnumAlgorithms(NCRYPT_CIPHER_OPERATION) no supported algorithms
NCryptEnumAlgorithms(NCRYPT_HASH_OPERATION) no supported algorithms
NCryptEnumAlgorithms(NCRYPT_ASYMMETRIC_ENCRYPTION_OPERATION):
   Name                           Class      Operations Flags
   RSA                            0x00000003 0x00000014 0x0
NCryptEnumAlgorithms(NCRYPT_SECRET_AGREEMENT_OPERATION):
   Name                           Class      Operations Flags
   DH                             0x00000004 0x00000008 0x0
   ECDH_P224                      0x00000004 0x00000008 0x0
   ECDH_P256                      0x00000004 0x00000008 0x0
   ECDH_P384                      0x00000004 0x00000008 0x0
   ECDH_P521                      0x00000004 0x00000008 0x0
NCryptEnumAlgorithms(NCRYPT_SIGNATURE_OPERATION):
   Name                           Class      Operations Flags
   RSA                            0x00000003 0x00000014 0x0
   DSA                            0x00000005 0x00000010 0x0
   ECDSA_P224                     0x00000005 0x00000010 0x0
   ECDSA_P256                     0x00000005 0x00000010 0x0
   ECDSA_P384                     0x00000005 0x00000010 0x0
   ECDSA_P521                     0x00000005 0x00000010 0x0
```

To list details of the algorithms for a specific named key storage provider, run the **cnglist** command with the **--list-algorithms** and **--provider=**"*ProviderName*" options:

```
cnglist --list-algorithms --provider="Microsoft Software Key Storage Provider"
```

Output from this command has the form:

```
Microsoft Software Key Storage Provider
NCryptEnumAlgorithms(NCRYPT_CIPHER_OPERATION) no supported algorithms
NCryptEnumAlgorithms(NCRYPT_HASH_OPERATION) no supported algorithms
NCryptEnumAlgorithms(NCRYPT_ASYMMETRIC_ENCRYPTION_OPERATION):
   Name                           Class      Operations Flags
   RSA                            0x00000003 0x00000014 0x0
NCryptEnumAlgorithms(NCRYPT_SECRET_AGREEMENT_OPERATION):
   Name                           Class      Operations Flags
   DH                             0x00000004 0x00000008 0x0
   ECDH_P256                      0x00000004 0x00000018 0x0
   ECDH_P384                      0x00000004 0x00000018 0x0
   ECDH_P521                      0x00000004 0x00000018 0x0
NCryptEnumAlgorithms(NCRYPT_SIGNATURE_OPERATION):
   Name                           Class      Operations Flags
   RSA                            0x00000003 0x00000014 0x0
   DSA                            0x00000005 0x00000010 0x0
   ECDSA_P256                     0x00000005 0x00000010 0x0
   ECDSA_P384                     0x00000005 0x00000010 0x0
   ECDSA_P521                     0x00000005 0x00000010 0x0
```

Chapter 7: nCipherKM JCA/JCE CSP

The nCipherKM JCA/JCE CSP (Cryptographic Service Provider) allows Java applications and services to access the secure cryptographic operations and key management provided by Thales hardware. This provider is used with the standard JCE (Java Cryptographic Extension) programming interface.

To use the nCipherKM JCA/JCE CSP, you must install:

- the **`javasp Java Support (including KeySafe)`** bundle
- the **`jcecsp nCipherKM JCA/JCE provider classes`** component.

For more information about the bundles and components supplied on your Security World Software installation media, see the User Guide.

The following versions of Java have been tested to work with, and are supported by, your Thales Security World Software:

- Java5 (or Java 1.5x)
- Java6 (or Java 1.6x)
- Java7 (or Java 1.7x)
- Java8 (or Java 1.8x).

We recommend that you ensure Java is installed before you install the Security World Software. The Java executable must be on your system path.

If you can do so, please use the latest Java version currently supported by Thales that is compatible with your requirements. Java versions before those shown are no longer supported. If you are maintaining older Java versions for legacy reasons, and need compatibility with current Thales software, please contact Thales support.

To install Java you may need installation packages specific to your operating system, which may depend on other pre-installed packages to be able to work.

Suggested links from which you may download Java software as appropriate for your operating system are:

Operating System	Download site
AIX	http://www.ibm.com/developerworks/systems/library/es-JavaOnAix_install.html
HPUX	https://h20392.www2.hp.com/portal/swdepot/displayProductInfo.do?productNumber=HPUXJAVAHOME
various	http://www.oracle.com/technetwork/java/index.html
various	http://www.oracle.com/technetwork/java/all-142825.html

Note: Detailed documentation for the JCE interface can be found on the Oracle Technology web page https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html.

Note: Softcards are not supported for use with the nCipherKM JCA/JCE CSP in Security Worlds that are compliant with FIPS 140-2 level 3.

Installing the nCipherKM JCA/JCE CSP

To install the nCipherKM JCA/JCE CSP:

1. In the hardserver configuration file, ensure that:
   - **`priv_port`** (the port on which the hardserver listens for local privileged TCP connections) is set to 9001
   - **`nonpriv_port`** (the port on which the hardserver listens for local nonprivileged TCP connections) is set to 9000.

   If you need to change either or both of these port settings, you restart the hardserver before continuing the nCipherKM JCA/JCE CSP installation process. For more information, see the User Guide.
2. Copy the **`nCipherKM.jar`** file to the extensions folder of your local Java Virtual Machine installation from the following directory:
   - Windows: *%NFAST_HOME%*`\java\classes`
   - Unix-based: `/opt/nfast/java/classes`

   The location of the extensions folder depends on the type of your local Java Virtual Machine (JVM) installation:

JVM type	Extensions folder
Java Developer Kit (JDK)	Windows: *%JAVA_HOME%*`\jre\lib\ext` Unix-based: *$JAVA_HOME*`/jre/lib/ext`
Java Runtime Environment (JRE)	Windows: *%JAVA_HOME%*`\lib\ext` Unix-based: *$JAVA_HOME*`/lib/ext`

   In these paths, *%JAVA_HOME%* (Windows) or *$JAVA_HOME* (Unix-based) is the home directory of the Java installation (commonly specified in the **`JAVA_HOME`** environment variable).
3. Add *%JAVA_HOME%*`\bin` (Windows) or *$JAVA_HOME*`/bin` (Unix-based) to your **`PATH`** system variable.
4. Install the unlimited strength JCE jurisdiction policy files that are appropriate to your version of Java.

   The Java Virtual Machine imposes limits on the cryptographic strength that may be used by default with JCE providers. Replace the default policy configuration files with the unlimited strength policy files.

To install the unlimited strength JCE jurisdiction policy files:

a. If necessary, download the archive containing the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files from your Java Virtual Machine vendor's Web site. Be sure to download a file appropriate for your version of Java.

   Note: The Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files are covered and controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. We recommend that you take legal advice before downloading these files from your Java Virtual Machine vendor.

b. Extract the files **`local_policy.jar`** and **`US_export_policy.jar`** from Java Virtual Machine vendor's Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy File archive.

c. Copy the extracted files **`local_policy.jar`** and **`US_export_policy.jar`** into the security directory for your local Java Virtual Machine (JVM) installation:

JVM type	Extensions folder
Java Developer Kit (JDK)	Windows: *%JAVA_HOME%*`/jre\lib\security` Unix-based: *$JAVA_HOME*`/jre/lib/security`
Java Runtime Environment (JRE)	Windows: *%JAVA_HOME%*`\lib\security` Unix-based: *$JAVA_HOME*`/lib/security`

   In these paths, *%JAVA_HOME%* (Windows) or *$JAVA_HOME* (Unix-based) is the home directory of the Java installation (commonly specified in the **`JAVA_HOME`** environment variable).

   Note: Copying the files **`local_policy.jar`** and **`US_export_policy.jar`** into the appropriate folder must overwrite any existing files with the same names.

5. Add the nCipherKM provider to the Java security configuration file **`java.security`** (located in the security directory for your local Java Virtual Machine (JVM) installation).
   The **`java.security`** file contains list of providers in preference order that is used by the Java Virtual Machine to decide from which provider to request a mechanism instance. Ensure that the nCipherKM provider is registered in the first position in this list, as shown in the following example:

```
#
# List of providers and their preference orders (see above):
#
security.provider.1=com.ncipher.provider.km.nCipherKM
security.provider.2=sun.security.provider.Sun
security.provider.3=sun.security.rsa.SunRsaSign
security.provider.4=com.sun.net.ssl.internal.ssl.Provider
security.provider.5=com.sun.crypto.provider.SunJCE
security.provider.6=sun.security.jgss.SunProvider
security.provider.7=com.sun.security.sasl.Provider
```

   Placing the nCipherKM provider first in the list permits the nCipherKM provider's algorithms to override the algorithms that would be implemented by any other providers (except in cases where you explicitly request another provider name).

Note: The nCipherKM provider cannot serve requests required for the SSL classes unless it is in the first position in the list of providers.

Do not change the relative order of the other providers in the list.

Note: If you add the nCipherKM provider as **`security.provider.1`**, ensure that the subsequent providers are re-numbered correctly. Ensure you do not list multiple providers with the same number (for example, ensure your list of providers does not include two instances of **`security.provider.1`**, both **`com.ncipher.provider.km.nCipherKM`** and another provider).

6. Save your updates to the file **`java.security`**.

   When you have installed the nCipherKM JCA/JCE CSP, you must have created a Security World before you can test or use it. For more information about creating a Security World, see the User Guide.

   Note: If you have a Java Enterprise Edition Application Server running, you must restart it before the installed nCipherKM provider is loaded into the Application Server virtual machine and ready for use.

Testing the nCipherKM JCA/JCE CSP installation

After installation, you can test that the nCipherKM JCA/JCE CSP is functioning correctly by running the command:

```
java com.ncipher.provider.InstallationTest
```

Note: For this command to work, you must have added *%JAVA_HOME%* (Windows) or *$JAVA_HOME* (Unix-based) to your **`PATH`** system variable.

If the nCipherKM JCA/JCE CSP is functioning correctly, output from this command has the following form:

```
Installed providers:
1: nCipherKM
2: SUN
3: SunRsaSign
4: SunJSSE
5: SunJCE
6: SunJGSS
7: SunSASL
Unlimited strength jurisdiction files are installed.
The nCipher provider is correctly installed.
nCipher JCE services:
Alg.Alias.Cipher.1.2.840.113549.1.1.1
Alg.Alias.Cipher.1.2.840.113549.3.4
Alg.Alias.Cipher.AES
Alg.Alias.Cipher.DES3
....
```

If the **`nCipherKM`** provider is installed but is not registered at the top of the providers list in the **`java.security`** file, the **`InstallationTest`** command produces output that includes the message:

```
The nCipher provider is installed, but is not registered at
the top of the providers list in the java.security file. See
the user guide for more information about the recommended
system configuration.
```

In such a case, edit the **`java.security`** file (located in the security directory for your local JVM installation) so that the nCipherKM provider is registered in the first position in that file's list of providers. For more information about the **`java.security`** file, see *Installing the nCipherKM JCA/JCE CSP* on page 104.

If the nCipherKM provider is not installed at all, or you have not created a Security World, or if you have not configured ports correctly in the hardserver configuration file, the **`InstallationTest`** command produces output that includes the message:

```
The nCipher provider is not correctly installed.
```

In such case:

- Check that you have configured ports correctly, as described in *Installing the nCipherKM JCA/JCE CSP* on page 104. For more information about hardserver configuration file settings, see the User Guide.
- Check that you have created a Security World. If you have not created a Security World, create a Security World. For more information, see the User Guide.
- If you have already created a Security World, repeat the nCipherKM JCA/JCE CSP installation process as described in *Installing the nCipherKM JCA/JCE CSP* on page 104.

After making any changes to the nCipherKM JCA/JCE CSP installation, run the **`InstallationTest`** command again and check the output.

Whether or not the nCipherKM provider is correctly installed, if the unlimited strength jurisdiction files are not installed or (not correctly installed), the **`InstallationTest`** command produces output that includes the message:

```
Unlimited strength jurisdiction files are NOT installed.
```

Note: The **`InstallationTest`** command can only detect this situation if you are using JRE/JDK version 1.5 or later.

This message means that, because the Java Virtual Machine imposes limits on the cryptographic strength that you can use by default with JCE providers, you must replace the default policy configuration files with the unlimited strength policy files. For information about how to install the unlimited strength jurisdiction files, see *Installing the nCipherKM JCA/JCE CSP* on page 104.

System properties

You can use system properties to control the provider. You set system properties when starting the Java Virtual Machine using a command such as:

```
java -Dproperty=value MyJavaApplication
```

In this example command, *property* represents any system property, *value* represents the value set for that property, and *MyJavaApplication* is the name of the Java application you are starting. You can set multiple system properties in a single command, for example:

```
java -Dprotect=module -DignorePassphrase=true MyJavaApplication
```

The available system properties and their functions as controlled by setting different values for a property are described in the following table:

Property	Function for different values
`JCECSP_DEBUG`	This property is a bit mask for which different values specify different debugging functions; the default value is **`0`**. For details about the effects of setting different values for this property, see *JCECSP_DEBUG property values* on page 109.
`JCECSP_DEBUGFILE`	This property specifies a path to the file to which logging output is to be written. Set this property if the **`JCECSP_DEBUG`** property is set to a value other than the default of **`0`**. For details about the effects of setting different values for this property, see *JCECSP_DEBUG property values* on page 109. In a production environment, we recommend that you disable debug logging to prevent sensitive information being made available to an attacker.
`protect`	This property specifies the type of protection to be used for key generation and nCipherKM KeyStore instances. You can set the value of this property to one of **`module`**, **`softcard:`***IDENT* or **`cardset`**. OCS protection (**`cardset`**) uses the card from the first slot of the first usable hardware security module. To find the logical token hash *IDENT* of a softcard, run the command **`nfkminfo --softcard-list`**.
`module`	This property lets you override the default module and select a specific module to use for module and OCS protection. Set the value of this property as the ESN of the module you want to use.
`slot`	This property lets you override the default slot for OCS-protection and select a specific slot to use. Set this the value of this property as the number of the slot you want to use.

Property	Function for different values
`ignorePassphrase`	If the value of this property is set to `true`, the nCipherKM provider ignores the pass phrase provided in its KeyStore implementation. This feature is included to allow the Oracle or IBM `keytool` utilities to be used with module-protected keys. The `keytool` utilities require a pass phrase be provided; setting this property allows a dummy pass phrase to be used.
`seeintegname`	Setting the value of this property to the name of an SEE integrity key causes the provider to generate SEE application keys. These keys may only be used by an SEE application signed with the named key.
`com.ncipher.provider.announcemode`	The default value for this property is `auto`, which uses firmware auto-detection to disable algorithms in the provider that cannot be supported across all installed modules. Setting the value of this property to `on` forces the provider to advertise all mechanisms at start-up. Setting the value of this property to `off` forces the provider to advertise no mechanisms at start-up.
`com.ncipher.provider.enable`	For the value of this property, you supply a comma-separated list of mechanism names that are to be forced on, regardless of the announce mode selected.
`com.ncipher.provider.disable`	For the value of this property, you supply a comma-separated list of mechanism names that are to be forced off, regardless of the announce mode selected. Any mechanism supplied in the value for the `com.ncipher.provider.disable` property overrides the same mechanism if it is supplied in the value for the `com.ncipher.provider.enable` property.

JCECSP_DEBUG property values

The `JCECSP_DEBUG` system property is a bit mask for which you can set different values to control the debugging functions. The following table describes the effects of different values that you can set for this property:

`JCECSP_DEBUG` value	Function
`0`	If this property has no bits set, no debugging information is reported. This is the default setting.
`1`	If this property has the bit 1 set, minimal debugging information (for example, version information and critical errors) is reported.
`2`	If this property has the bit 2 set, comprehensive debugging information is reported.
`4`	If this property has the bit 3 set, debugging information relating to creation and destruction of memory and module resources is reported.
`8`	If this property has the bit 4 set, `debugFunc` and `debugFuncEnd` generate debugging information for functions that call them.

`JCECSP_DEBUG` value	Function
`16`	If this property has the bit 5 set, **`debugFunc`** and **`debugFuncEnd`** display the values for all the arguments that are passed in to them.
`32`	If this property has the bit 6 set, context information is reported with each debugging message (for example, the **`ThreadID`** and the current time.
`64`	If this property has the bit 7 set, the time elapsed during each logged function is calculated, and information on the number of times a function is called and by which function it was called is reported.
`128`	If this property has the bit 8 set, debugging information for NFJAVA is reported in the debugging file.
`256`	If this property has the bit 9 set, the call stack is printed for every debug message.

To set multiple logging functions, add up the **`JCECSP_DEBUG`** values for the debugging functions you want to set, and specify the total as the value for **`JCECSP_DEBUG`**. For example, if you want to set the debugging to use both function tracing (bit 4) and function tracing with parameters (bit 5), add the **`JCECSP_DEBUG`** values shown in the table for these debugging functions (**`8`** + **`16`** = 24) and specify this total (**`24`**) as the value to use for **`JCECSP_DEBUG`**.

Compatibility

The nCipherKM JCA/JCE CSP supports both module-protected keys and OCS-protected keys. The CSP currently supports 1/*N* OCSs and a single protection type for each nCipherKM JCE KeyStore.

You can use the nCipherKM JCA/JCE CSP with Security Worlds that comply with FIPS 140-2 at either level 2 or level 3.

Note: In a Security World that complies with FIPS 140-2 level 3, it is not possible to import keys generated by other JCE providers.

The nCipherKM JCA/JCE CSP supports load-sharing for keys that are stored in the nCipherKM KeyStore. This feature allows a server to spread the load of cryptographic operations across multiple connected modules, providing greater scalability.

Note: We recommend that you use load-sharing unless you have existing code that is designed to run with multiple modules. To share keys with load-sharing, you must create a 1/*N* OCS with at least as many cards as you have modules. All the cards in the OCS must have the same pass phrase.

Keys generated or imported by the nCipherKM JCA/JCE CSP are not recorded into the Security World until:

1. The key is added to an nCipherKM KeyStore (by using a call to **`setKeyEntry()`** or **`setCertificateEntry()`**).
2. That nCipherKM KeyStore is then stored (by using a call to **`store()`**).

The pass phrase used with the KeyStore must be the pass phrase of the card from the OCS that protects the keys in the KeyStore.

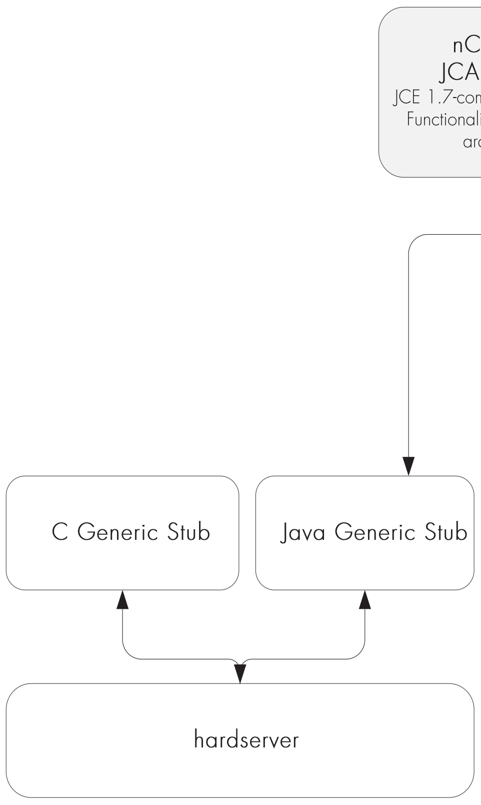
Architecture

The nCipherKM JCA/JCE CSP implements its functionality using two underlying nShield APIs:

- the KM Java library (`kmjava`)
- the Java Generic Stub (`nfjava`).

These libraries relay commands generated by the JCE provider to the underlying hardserver and modules.

Figure 8. nCipherKM JCA/JCE CSP architecture


nCipherKM
JCA/JCE CSP
JCE 1.7-compliant application
Functionality limited by JCE architecture
C Generic Stub
Java Generic Stub
hardserver

Available functions

The module firmware automatically detects which algorithms it can support. These algorithms are advertised when the provider first starts up. The provider conservatively advertises only those mechanisms that are supported by all installed modules in the system.

Note: Certain algorithms are not supported by older versions of firmware. We recommend that you ensure that your module is upgraded to the most recent version of firmware appropriate for your environment.

Cipher	Cipher mode						Padding type							
	CBC	CFB	CTR	ECB	OFB	GCM	ANSI X9.23	ISO 10126	ISO 7816	None	OAEP	PKCS #1	PKCS #5	Zero byte
AESWrap				X						X				

Cipher	Cipher mode						Padding type							
	CBC	CFB	CTR	ECB	OFB	GCM	ANSI X9.23	ISO 10126	ISO 7816	None	OAEP	PKCS #1	PKCS #5	Zero byte
ArcFour														
Blowfish	X	X	X	X	X		X	X	X	X			X	X
CAST256	X	X	X	X	X		X	X	X	X			X	X
Blowfish	X	X	X	X	X		X	X	X	X			X	X
CAST	X	X	X	X	X		X	X	X	X			X	X
DES2	X	X	X	X	X		X	X	X	X			X	X
DES	X	X	X	X	X		X	X	X	X			X	X
DESede	X	X	X	X	X		X	X	X	X			X	X
DESedeWrap	X									X				
Rijndael	X	X	X	X	X	X	X	X	X	X			X	X
RSA				X							X	X		
Serpent	X	X	X	X	X		X	X	X	X			X	X
Twofish	X	X	X	X	X		X	X	X	X			X	X

Note: The Blowfish, CAST, Serpent, and Twofish algorithms are not supported for modules with firmware version 2.33.60 or later.

Algorithm	Key length in bits for generation or signing	Use for ...									
		KeyGenerator	KeyPairGenerator	Signature	Cipher	KeyAgreement	KeyFactory	KeyStore	MAC	MessageDigest	SecureRandom
AESWrap					Y						
Arcfour	8, 16 to 2048	Y			Y						
Blowfish	32, 40 to 448	Y			Y						
CAST	40, 48 to 128	Y			Y						
CAST256	128, 192, 256	Y			Y						
DES	64	Y			Y						
DESede	192	Y			Y						

Algorithm	Key length in bits for generation or signing	Use for ...									
		KeyGenerator	KeyPairGenerator	Signature	Cipher	KeyAgreement	KeyFactory	KeyStore	MAC	MessageDigest	SecureRandom
DES2	128	Y			Y						
DESedeWrap					Y						
DH			Y			Y	Y				
DSA	1024		Y				Y				
ECDH			Y			Y	Y				
ECDSA			Y				Y				
HmacMD5		Y							Y		
HmacRIPEMD160	8, 16 to 2048	Y							Y		
HmacSHA1	8, 16 to 2048	Y							Y		
HmacSHA224	8, 16 to 2048	Y							Y		
HmacSHA256	8, 16 to 2048	Y							Y		
HmacSHA384	8, 16 to 2048	Y							Y		
HmacSHA512	8, 16 to 2048	Y							Y		
HmacTiger	8, 16 to 2048	Y							Y		
MD5										Y	
MD5andSHA1withRSA				Y							
MD5withRSA				Y							
nCipher.sworld								Y			
Rijndael		Y			Y						
RawRSA				Y							
RIPEMD160withRSA				Y							
RIPEMD160withRSAandMGF1	322+			Y							
RND											Y
RSA	512+		Y		Y		Y				
Serpent	8, 16 to 256	Y			Y						
SHA1										Y	
SHA1withDSA				Y							

Algorithm	Key length in bits for generation or signing	Use for ... KeyGenerator	KeyPairGenerator	Signature	Cipher	KeyAgreement	KeyFactory	KeyStore	MAC	MessageDigest	SecureRandom
SHA1withECDSA				Y							
SHA1withRSA				Y							
SHA1withRSAandMGF1	322+			Y							
SHA224										Y	
SHA224withDSA				Y							
SHA224withECDSA				Y							
SHA224withRSA				Y							
SHA224withRSAandMGF1	450+			Y							
SHA256										Y	
SHA256withDSA				Y							
SHA256withECDSA				Y							
SHA256withRSA				Y							
SHA256withRSAandMGF1	514+			Y							
SHA384										Y	
SHA384withDSA				Y							
SHA384withECDSA				Y							
SHA384withRSA				Y							
SHA384withRSAandMGF1	770+			Y							
SHA512										Y	
SHA512withDSA				Y							
SHA512withECDSA				Y							
SHA512withRSA				Y							
SHA512withRSAand MGF1	1026+			Y							
Tiger	8, 16 to 256	Y			Y						

Note: The Blowfish, CAST, Serpent, Twofish, and MD5withRSA algorithms are not supported for modules with firmware version 2.33.60 or later.

The KeyStore API

You can load and store nShield module-protected keys by using the standard KeyStore API. This interface allows access to a KeyStore data file by means of a pass phrase and an **`InputStream`** or **`OutputStream`**.

nShield KeyStore data files contain only the name-space identifier of the keys stored in them; the actual keys are stored in the Security World regardless of the stream used. The name-space identifier is the hash of the root key of the individual KeyStore. The **`ident`** of the KeyStore keys in the Security World begins with this hash and is followed by key-specific characters. This naming hierarchy allows you to identify the relevant key in Security World tools (such as KeySafe) and remove keys from a KeyStore.

Note: To use an existing KeyStore on another machine in the same Security World, copy both its KeyStore data file and the Security World's Key Management Data directory to the other machine.

Initialization

You create a new KeyStore by passing a null **`InputStream`** to the KeyStore load method. When you create a new KeyStore, the nCipherKM provider generates a KeyStore key that is used to sign trusted public certificate entries. The relevant signature is verified when public certificates in the KeyStore are used; this functionality prevents an attacker inserting new certificates into a KeyStore without the protection token that is needed to use the KeyStore key.

By default, the KeyStore protection key is OCS-protected. Ensure that the pass phrase argument used with the KeyStore interface matches the pass phrase of that OCS. When the KeyStore method is called, you must present a card with a matching pass phrase from the required OCS. You can use the **`protect`** system property to change the protection type used for the KeyStore key; for more information about the **`protect`** property, see *System properties* on page 108.

An existing KeyStore file is not overwritten if the KeyStore store method is called on an **`OutputStream`** directed at the same file path. Instead, the KeyStore at the existing path is used to store the keys in the new KeyStore. This operation fails if the pass phrases for the two KeyStores do not match.

Loading and storing keys

We recommend that separate KeyStores are used for separate purposes; for example, you can use one KeyStore to hold private keys and a different KeyStore for Certifying Authorities. With this approach, you need separate OCSs to operate separate KeyStores. However, you can also use different OCSs to protect keys within the same KeyStore.

You require a certificate chain to store private keys. The Virtual Machine JCE implementation enforces this requirement, not the nCipherKM provider.

keytool

You can use either the Oracle **`keytool`** utility or the IBM **`keytool`** utility to read and edit an nShield KeyStore. These utilities are shipped with the Oracle and IBM JVMs. You must specify the correct

nCipher.sworld KeyStore type when you run the **keytool** utility, and you must specify the correct package name for the Oracle or IBM **keytool** utility.

To generate a new key in an OCS-protected KeyStore with the Oracle or IBM **keytool** utility, run the appropriate command:

- Sun Microsystems **keytool** utility:

```
java sun.security.tools.KeyTool -genkey -storetype nCipher.sworld -keyalg RSA -sigalg
SHA1withRSA -storepass KeyStore_passphrase -keystore KeyStore_path
```

- IBM **keytool** utility:

```
java com.ibm.crypto.tools.KeyTool -genkey -storetype nCipher.sworld -keyalg RSA -sigalg
SHA1withRSA -storepass KeyStore_passphrase -keystore KeyStore_path
```

In these example commands, *KeyStore_passphrase* is the pass phrase for the OCS that protects the KeyStore and *KeyStore_path* is the path to that KeyStore.

To generate a new key in a module-protected KeyStore with the Oracle or IBM **keytool** utility, run the appropriate command:

- Sun Microsystems **keytool** utility:

```
java -Dprotect=module -DignorePassphrase=true sun.security.tools.KeyTool -genkey -
storetype nCipher.sworld -keyalg RSA -sigalg SHA1withRSA -keystore KeyStore_path
```

- IBM **keytool** utility:

```
java -Dprotect=module -DignorePassphrase=true com.ibm.crypto.tools.KeyTool -genkey -
storetype nCipher.sworld -keyalg RSA -sigalg SHA1withRSA -keystore KeyStore_path
```

In these example commands, *KeyStore_path* is the path to the KeyStore.

By default, the **keytool** utilities use the **MD5withRSA** signature algorithm to sign certificates used with a KeyStore. This signature mechanism is unavailable on modules with firmware version 2.33.60 or later.

Using keys

Only the nCipherKM provider can use keys stored in an nShield KeyStore because the underlying key material is held separately in the Security World.

You can always store nShield keys in an nShield KeyStore. You can also store keys generated by a third-party provider into an nShield KeyStore if both of the following conditions apply:

- the key type is known to the nCipherKM provider
- the Security World is *not* compliant with FIPS 140-2 level 3.

When you generate an nShield key (or create it from imported key material), that key is associated with an ACL (Access Control List). This ACL prevents the key from being used for operations for which it is unsuited and enforces requirements that certain tokens be presented; for example, the ACL can specify that signing key cannot be used for encryption.

Glossary

Authorized Card List

Controls the use of Remote Administration cards. If the serial number of a card does not appear in the Authorized Card List, it is not recognized by the system and cannot be used. The list only applies to Remote Administration cards.

Access Control List (ACL)

An Access Control List is a set of information contained within a key that specifies what operations can be performed with the associated key object and what authorization is required to perform each of those operations.

Administrator Card Set (ACS)

Part of the Security World architecture, an Administrator Card Set (ACS) is a set of smart cards used to control access to Security World configuration, as well as recovery and replacement operations.

The Administrator Cards containing share in the logical tokens that protect the Security World keys, including $\mathbf{K}_{\mathrm{NSO}}$, the key-recovery key, and the recovery authorization keys. Each card contains one share from each token. The ACS is created using the well-known module key so that it can be loaded onto any nShield module.

See also *Security World*, *Operator Card Set (OCS)*

Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) is a block cipher adopted as an encryption standard by the US government and officially documented as US FIPS PUB 197 (FIPS 197). Originally only used for non-classified data, AES was also approved for use with for classified data in June 2003. Like its predecessor, the Data Encryption Standard (DES), AES has been analyzed extensively and is now widely used around the world.

Although AES is often referred to as *Rijndael* (the cipher having been submitted to the AES selection process under that name by its developers, Joan Daemen and Vincent Rijmen), these are not precisely the same cipher. Technically, Rijndael supports a larger range of block and key sizes (at any multiple of 32 bits, to a minimum of 128 bits and a maximum of 256 bits); AES has a fixed block size of 128 bits and only supports key sizes of 128, 192, or 256 bits.

See also *Data Encryption Standard (DES)* on page 119

CAST

CAST is a symmetric encryption algorithm with a 64-bit block size and a key size of between 40 bits to 128 bits (but only in 8-bit increments).

client identifier: R_{SC}

This notation represents an arbitrary number used to identify a client. In the nCore API, all client identifiers are 20 bytes long.

Data Encryption Standard (DES)

The Data Encryption Standard (DES) is a symmetric cipher approved by NIST for use with US Government messages that are Secure but not Classified. The implementation of DES used in the module has been validated by NIST. DES uses a 64-bit block and a 56-bit key. DES keys are padded to 64 bits with 8 parity bits.

See also *Triple DES* on page 124, *Advanced Encryption Standard (AES)* on page 118

Diffie-Hellman

The Diffie-Hellman algorithm was the first commercially published public key algorithm. The Diffie-Hellman algorithm can only be used for key exchange.

Digital Signature Algorithm (DSA)

Also known as the Digital Signature Standard (DSS), the Digital Signature Algorithm (DSA) is a digital signature mechanism approved by NIST for use with US Government messages that are Secure but not Classified. The implementation of the DSA used by nShield modules has been validated by NIST as complying with FIPS 186.

Digital Signature Standard (DSS)

See *Digital Signature Algorithm (DSA)* on page 119

ECDH

A variant of the Diffie-Hellman anonymous key agreement protocol which uses elliptic curve cryptography.

See also *Diffie-Hellman* on page 119.

ECDSA

Elliptic Curve DSA: a variant of the Digital Signature Algorithm (DSA) which uses elliptic curve cryptography.

See also *Digital Signature Algorithm (DSA)* on page 119, *Diffie-Hellman* on page 119.

encryption: $\{A\}_B$

This notation indicates the result of **A** encrypted with key **B**.

Federal Information Processing Standards (FIPS)

The Federal Information Processing Standards (FIPS) were developed by the United States federal government for use by non-military government agencies and government contractors. FIPS 140 is a series of publications intended to coordinate the requirements and standards for cryptographic security modules, including both their hardware and software components.

All Security Worlds are compliant with FIPS 140-2. By default, Security Worlds are created to comply with FIPS 140-2 at level 2, but those customers who have a regulatory requirement for compliance with FIPS 140-2 at level 3 can also choose to create a Security World that meets those requirements.

For more details about FIPS 140-2, see http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf.

hardserver

The hardserver software controls communication between applications and nShield modules, which may be installed locally or remotely. It runs as a service on the host computer. The behavior of the hardserver is controlled by the settings in the hardserver configuration file.

The hardserver software controls communication between the internal hardware security module and applications on the network. The module hardserver is configured using the front panel on the module or by means of uploaded configuration data. Configuration data is stored on the module and in files in a specially configured file system on each client computer.

hardware security module (HSM)

A hardware security module (commonly referred to as an HSM) is a hardware device used to hold cryptographic keys and software securely.

hash: H(X)

This notation indicates a fixed length result that can be obtained from a variable length input and that can be used to identify the input without revealing any other information about it. The nShield module uses the Secure Hash Algorithm (SHA-1) for its internal security.

identifier hash: $H_{ID}(X)$

An identifier hash is a hash that uniquely identifies a given object (for example, a key) without revealing the data within that object. The module calculates the identity hash of an object by hashing together the object type and the key material. The identity hash has the following properties:

H_{ID} is not modified by any operations on the key (for example, altering the ACL, the application data field, or other modes and flags)

H_{ID} is the same for both public and private halves of a key pair.

Unique data is added to the hash so that a $\mathbf{H}_{ID}$ is most unlikely to be the same as any other hash value that might be derived from the key material.

key blob

A key blob is a key object with its ACL and application data encrypted by a module key, a logical token, or a recovery key. Key blobs are used for the long-term storage of keys. Blobs are cryptographically secure; they can be stored on the host computer's hard disk and are only readable by units that have access to the same module key.

See also *Access Control List (ACL)*.

key object: K_A

This is a key object to be kept securely by the module. A key object may be a private key, a public counterpart to a private key, a key for a symmetric cipher (MAC or some other symmetric algorithm), or an arbitrary block of data. Applications can use this last type to allow the module to protect any other data items in the same way that it protects cryptographic keys. Each key object is stored with an ACL and a 20-byte data block that the application can use to hold any relevant information.

KeyID: ID_{KA}

When a key object KA is loaded within the module's RAM, it is given a short identifier or handle that is notated as **ID**$_{KA}$. This is a transient identifier, not to be confused with the key hash **HID(KA)**.

logical token: K_T

A logical token is a key used to protect key blobs. A logical token is generated on the nShield module and never revealed, except as shares.

MAC: MAC_{KC}

This notation indicates a MAC (Message Authentication Code) created using key **KC**.

module

See *hardware security module (HSM)*.

module key: K_M

A module key is a cryptographic key generated by each nShield module at the time of initialization and stored within the module. It is used to wrap key blobs and key fragments for tokens. Module keys can be shared across several modules to create a larger Security World.

All modules include two module keys:

- module key zero **K**$_{M0}$, a module key generated when the module is initialized and never revealed outside the module.
- **null**, or well-known module key **K**$_{MWK}$.

You can program extra module keys into a module.

See also: *Security World*, *hardware security module (HSM)*.

module signing key: K_{ML}

The module signing key is the module's public key. It is used to issue certificates signed by the module. Each module generates its own unique **K**$_{ML}$ and **K**$_{ML}^{-1}$ values when it is initialized. The private half of this key pair, **K**$_{ML}^{-1}$, is never revealed outside the module.

nShield master feature enable key K_{SA}

Certain features of the module firmware are available as options. These features must be purchased separately from Thales. To use a feature on a specific module, you require a certificate from Thales signed by $\mathbf{K}_{SA}$. These certificates include the electronic serial number for the module.

nShield Remote Administration Card

Smart cards that are capable of negotiating cryptographically secure connections with an HSM, using warrants as the root of trust. nShield Remote Administration Cards can also be used in the local slot of an HSM if required. You must use nShield Remote Administration Cards with Remote Administration.

nShield Security Officer's key: K_{NSO}^{-1}

The notation $\mathbf{K}_{NSO}^{-1}$ indicates the Security Officer's signing key. This key is usually a key to a public-key signature algorithm.

nShield Trusted Verification Device

A smart card reader that allows the card holder to securely confirm the Electronic Serial Number (ESN) of the HSM to which they want to connect, using the display of the device. Thales supplies and the nShield Trusted Verification Device and recommends its use with Remote Administration.

null module key: K_{MWK}

The null module key is used to create tokens that can be loaded onto any module. Such tokens are required for recovery schemes. The null module key is a Triple DES key of a value 01010101. As this value is well known, this module key does not have any security. Key blobs cannot be made directly under the null module key. To make a blob under a token protected by the null module key, the key must have the ACL entry **AllowNullKMToken**.

Operator Card Set (OCS)

Part of the Security World architecture, an Operator Card Set (OCS) is a set of smart cards containing shares of the logical tokens that is used to control access to application keys within a Security World. OCSs are protected using the Security World key, and therefore they cannot be used outside the Security World.

See also: *Security World*, *Administrator Card Set (ACS)*.

Recovery key: K_{RA}

The recovery key is the public key of the key recovery agent.

Remote Access Client, Server and solution

The remote access solution, such as SSH or a remote desktop application, which is used as standard by your organization. Enables you to to carry out Security World administrative tasks from a different location to that of an nShield Connect or nShield Solo.

For example, the remote access solution is used to run security world utilities remotely and to enter passphrases.

Note: Thales does not provide this software.

Remote Administration

An optional Security World feature that enables Remote Administration card holders to present their cards to an HSM located elsewhere. For example, the card holder may be in an office, while the HSM is in a data center. Remote Administration supports the ACS, as well as persistent and non-persistent OCS cards, and allows all smart card operations to be carried out, apart from loading feature certificates.

nShield Remote Administration Client

A GUI or command-line interface that enables you to select an HSM located elsewhere from a list provided by the Remote Administration Service, and associate a card reader attached to your computer with the HSM. Resides on your local Windows or Linux-based computer.

Remote Administration Service

Enables secure communications between an nShield Remote Administration Card and the hardserver that is connected to the appropriate HSM. Listens for incoming connection requests from nShield Remote Administration Clients. Supplies a list of available HSMs to the nShield Remote Administration Client and maintains an association between the relevant card reader and the HSM.

Dynamic Slot

Virtual card slots that can be associated with a card reader connected to a remote computer. Remote Administration Slots are in addition to the local slot of an HSM and any soft card slot that may be available. HSMs have to be configured to support between zero (default) and 16 Remote Administration Slots.

Rijndael

See *Advanced Encryption Standard (AES)* on page 118

salt: X

The random value, or salt, is used in some commands to discourage brute force searching for keys.

Security World

The Security World technology provides an infrastructure for secure lifecycle management of keys. A Security World consists of at least one hardware security module, some cryptographic key and certificate data encrypted by a Security World key and stored on at least one host computer, a set of Administrator Cards used to control access to Security World configuration, recovery and replacement operations, and optionally one or more sets of Operator Cards used to control access to application keys.

See also *Administrator Card Set (ACS)*, *Operator Card Set (OCS)*.

Security World key: K_{MSW}

The Security World key is the module key that is present on all modules in a Security World. Each Security World has a unique Security World key. This key is generated randomly when the Security World is created, and it is stored as a key blob protected by the ACS.

share: K_{Ti}

The notation **K_{Ti}** indicates a share of a logical token. Shares can be stored on smart cards or software tokens. Each share is encrypted under a separate share key.

share key: K_{Si}

A share key is a key used to protect an individual share in a token. Share keys are created from a Security World key, a pass phrase, and a salt value.

Standard nShield Cards

Smart cards used in the local slot of an HSM. Standard nShield cards are not supported for use with Remote Administration.

Standard card reader

A smart card reader for ISO/IEC 7816 compliant smart cards. Thales recommends that standard smart card readers are only used with the nShield Remote Administration Client command-line utility, not the GUI.

Triple DES

Triple DES is a highly secure variant of the Data Encryption Standard (DES) algorithm in which the message is encrypted three times.

See also *Data Encryption Standard (DES)* on page 119, *Advanced Encryption Standard (AES)* on page 118.

Internet addresses

Web site:	http://www.thales-esecurity.com/
Support:	http://www.thales-esecurity.com/support-landing-page
Online documentation:	http://www.thales-esecurity.com/knowledge-base
International sales offices:	http://www.thales-esecurity.com/contact

Addresses and contact information for the main Thales e-Security sales offices are provided at the bottom of the following page.

THALES

www.thales-esecurity.com

About Thales e-Security

Thales e-Security is a leading global provider of trusted cryptographic solutions with a 40-year track record of protecting the world's most sensitive applications and information. Thales solutions enhance privacy, trusted identities, and secure payments with certified, high performance encryption and digital signature technology for customers in a wide range of markets including financial services, high technology, manufacturing, and government. Thales e-Security has a worldwide support capability, with regional headquarters in the United States, the United Kingdom, and Hong Kong. www.thales-esecurity.com

Follow us on:

Americas – Thales e-Security Inc. 900 South Pine Island Road, Suite 710, Plantation, FL 33324 USA • Tel: +1 888 744 4976 or +1 954 888 6200 • Fax: +1 954 888 6211 • E-mail: sales@thalesesec.com
Asia Pacific – Unit 4101 41/F 248, Queen's Road East, Wanchai, Hong Kong, PRC • Tel: +852 2815 8633 • Fax: +852 2815 8141 • E-mail: asia.sales@thales-esecurity.com
Europe, Middle East, Africa – Meadow View House, Long Crendon, Aylesbury, Buckinghamshire HP18 9EQ • Tel: +44 (0)1844 201800 • Fax: +44 (0)1844 208550 • E-mail: emea.sales@thales-esecurity.com