



nShield Security World

PKCS #11 Reference Guide for nShield Security World v13.9.3

12 December 2025

Table of Contents

1. Introduction	1
1.1. Read this guide if.....	2
1.2. Additional useful documentation.....	2
1.3. Security World Software default directories	2
1.4. Utility help options.....	4
1.5. Further information	4
1.6. Security advisories	4
1.7. Contacting Entrust nShield Support.....	5
2. nShield Architecture	6
2.1. Security World Software modules.....	6
2.2. Security World Software server.....	6
2.3. Stubs and interface libraries	7
2.4. Using an interface library	7
2.5. Writing a custom application	8
2.6. Acceleration-only or key management.....	8
3. PKCS #11 Developer libraries	9
3.1. Checking the installation of the nShield PKCS #11 library	9
3.2. PKCS #11 security assurance mechanism.....	9
3.2.1. Key security	10
4. PKCS #11 with load sharing mode.....	11
4.1. Logging in	12
4.2. Session objects.....	12
4.3. Module failure	12
4.4. Compatibility	13
4.5. Restrictions on function calls in load-sharing mode.....	13
5. PKCS #11 with HSM Pool mode	14
5.1. Module failure.....	14
5.2. Module recovery	14
5.3. Restrictions on function calls in HSM Pool mode	14
6. Generating and deleting NVRAM-stored keys with PKCS #11.....	16
6.1. Generating NVRAM-stored keys	16
6.2. Deleting NVRAM-stored keys.....	17
7. PKCS #11 with key reloading	18
7.1. Usage under preload	18
7.1.1. Persistent preload files	19
7.2. Supported function calls	19
7.3. Retrying key reloads.....	19

7.4. Adding new HSMs	19
8. PKCS #11 without load-sharing or HSM Pool modes	21
8.1. K/N support for PKCS #11	21
9. PKCS #11 with preload	23
10. PKCS #11 Security Officer	24
11. nShield-specific PKCS #11 API extensions	25
11.1. C_LoginBegin	25
11.2. C_LoginNext	25
11.3. C_LoginEnd	26
12. Compiling and linking	27
12.1. Windows	27
12.2. Linux	27
13. nShield PKCS #11 library environment variables	29
13.1. CKNFAST_ASSUME_SINGLE_PROCESS	30
13.2. CKNFAST_ASSURANCE_LOG	31
13.3. CKNFAST_CARDSET_HASH	31
13.4. CKNFAST_CONCATENATIONKDF_X963_COMPLIANCE	31
13.5. CKNFAST_DEBUG	31
13.6. CKNFAST_DEBUGDIR	32
13.7. CKNFAST_DEBUGFILE	32
13.8. CKNFAST_DH_LSB	32
13.9. CKNFAST_EDDSA_PUBKEY_FORMAT	32
13.10. CKNFAST_FAKE_ACCELERATOR_LOGIN	33
13.11. CKNFAST_HA_MINIMUM_INTERVAL	33
13.12. CKNFAST_HSM_POOL	33
13.13. CKNFAST_JCE_COMPATIBILITY	34
13.14. CKNFAST_LOADSHARING	34
13.15. CKNFAST_NO_ACCELERATOR_SLOTS	34
13.16. CKNFAST_NO_SYMMETRIC	35
13.17. CKNFAST_NO_UNWRAP	35
13.18. CKNFAST_NONREMOVABLE	35
13.19. CKNFAST_NVRAM_KEY_STORAGE	35
13.20. CKNFAST_OVERRIDE_SECURITY_ASSURANCES	35
13.20.1. all	36
13.20.2. none	37
13.20.3. tokenkeys	37
13.20.4. longterm[=days]	37
13.20.5. explicitness	38
13.20.6. import	38

13.20.7. wrapping_crypt	39
13.20.8. unwrap_kek	39
13.20.9. derive_kek	39
13.20.10. derive_xor	40
13.20.11. derive_concatenate	40
13.20.12. unwrap_rsa_aes_kwp	40
13.20.13. weak_<algorithm>	41
13.20.14. shortkey_<algorithm=bitlength>	41
13.20.15. silent	41
13.20.16. Diagnostic warnings about questionable operations	41
13.21. CKNFAST_SEED_MAC_ZERO	42
13.22. CKNFAST_SESSION_THREADSafe	42
13.23. CKNFAST_SESSION_TO_TOKEN	42
13.24. CKNFAST_SHARE_SESSION_KEYS	42
13.25. CKNFAST_TOKENS_PERSISTENT	43
13.26. CKNFAST_USE_THREAD_UPCALLS	43
13.27. CKNFAST_LOAD_KEYS	44
13.28. CKNFAST_WRITE_PROTECTED	44
13.29. CKNFAST_RELOAD_KEYS	44
14. Objects	45
14.1. Certificate Objects and Data Objects	46
14.2. Key Objects	46
14.3. Card passphrases	46
15. Mechanisms	48
15.1. Footnote 1	55
15.2. Footnote 2	55
15.3. Footnote 3	55
15.4. Footnote 4	55
15.5. Footnote 5	55
15.6. Footnote 6	55
15.7. Footnote 7	56
15.8. Footnote 8	56
15.9. Footnote 9	57
15.10. Footnote 10	57
15.11. Footnote 11	57
15.12. Footnote 12	58
15.13. Footnote 13	58
15.14. Footnote 14	58
15.15. Footnote 15	59

15.16. Footnote 16	59
15.17. Footnote 17	59
15.18. Footnote 18	59
15.19. Footnote 19	59
15.20. Footnote 20	59
16. Vendor annotations on P11 mechanisms	60
16.1. CKM_RSA_PKCS_OAEP	60
16.2. CKM_RSA_PKCS_PSS and CKM_SHA*_RSA_PKCS_PSS	60
17. Vendor-defined mechanisms	62
17.1. CKM_SEED_ECB_ENCRYPT_DATA and CKM_SEED_CBC_ENCRYPT_DATA	62
17.2. CKM_CAC_TK_DERIVATION	62
17.3. CKM_SHA*_HMAC and CKM_SHA*_HMAC_GENERAL	63
17.4. CKM_NC_ECKDF_HYPERLEDGER	64
17.5. CKM_HAS160	65
17.6. CKM_PUBLIC_FROM_PRIVATE	66
17.7. CKM_NC_AES_CMAC	66
17.8. CKM_NC_AES_CMAC_KEY_DERIVATION and CKM_NC_AES_CMAC_KEY_DERIVATION_SCP03	66
17.9. CK_NC_AES_CMAC_KEY_DERIVATION_PARAMS	68
17.10. CKM_COMPOSITE_EMV_T_ARQC, CKM_WATCHWORD_PIN1 and CKM_WATCHWORD_PIN2	68
17.11. CKM_NC_ECIES	68
17.12. CKM_NC_MILENAGE_OPC	70
17.13. CKM_NC_MILENAGE, CKM_NC_MILENAGE_AUTS, CKM_NC_MILENAGE_RESYNC	70
17.13.1. CKM_NC_MILENAGE	71
17.13.2. CKM_NC_MILENAGE_RESYNC	71
17.13.3. CKM_NC_MILENAGE_AUTS (testing only)	72
17.14. CKM_NC_TUAK_TOPC	72
17.15. CKM_NC_TUAK, CKM_NC_TUAK_AUTS, CKM_NC_TUAK_RESYNC	72
17.15.1. CKM_NC_TUAK	73
17.15.2. CKM_NC_TUAK_RESYNC	74
17.15.3. CKM_NC_TUAK_AUTS (testing only)	74
17.16. CKA_NC_KSD_IVS and CKA_NC_KSD_COUNTERS	74
17.16.1. CKA_NC_KSD_COUNTERS selection	75
17.16.2. Setting CKA_NC_KSD_IVS and CKA_NC_KSD_COUNTERS	75
17.16.3. Using C_Wrap CKK_AES and CKA_NC_KSD_IVS	76
17.16.4. Using C_Wrap CKK_RSA and CKA_NC_KSD_IVS	76
18. KISAAlgorithm mechanisms	78

18.1. KCDSA keys	78
18.2. Pre-hashing	78
18.3. CKM_KCDSA_SHA1, CKM_KCDSA_HAS160, CKM_KCDSA_RIPEMD160	79
18.4. CKM_KCDSA_KEY_PAIR_GEN	79
18.5. CKM_KCDSA_PARAMETER_GEN	80
18.6. CKM_HAS160	80
18.7. SEED secret keys	80
18.7.1. CKM_SEED_KEY_GEN	80
18.7.2. CKM_SEED_ECB, CKM_SEED_CBC, CKM_SEED_CBC_PAD	80
18.7.3. CKM_SEED_MAC, CKM_SEED_MAC_GENERAL	81
19. Attributes	82
19.1. CKA_SENSITIVE	82
19.2. CKA_PRIVATE	82
19.3. CKA_EXTRACTABLE	82
19.4. CKA_ENCRYPT, CKA_DECRYPT, CKA_SIGN, CKA_VERIFY	83
19.5. CKA_WRAP, CKA_UNWRAP	83
19.6. CKA_WRAP_TEMPLATE, CKA_UNWRAP_TEMPLATE	84
19.7. CKA_SIGN_RECOVER	85
19.8. CKA_VERIFY_RECOVER	86
19.9. CKA_DERIVE	86
19.10. CKA_ALLOWED_MECHANISMS	86
19.10.1. CKM_CONCATENATE_BASE_AND_KEY	86
19.10.2. CKM_RSA_AES_KEY_WRAP	87
19.11. CKA_MODIFIABLE	87
19.12. CKA_TOKEN	87
19.13. CKA_START_DATE, CKA_END_DATE	87
19.14. CKA_TRUSTED and CKA_WRAP_WITH_TRUSTED	87
19.15. CKA_COPYABLE and CKA_DESTROYABLE	88
19.16. RSA key values	89
19.17. DSA key values	89
19.18. Vendor specific error codes	89
20. Utilities	90
20.1. ckdes3gen	90
20.2. ckinfo	90
20.3. cklist	90
20.4. ckmechinfo	91
20.5. ckmlsa	91
20.6. cksagen	91
20.7. cksotool	91

21. Functions	93
21.1. Choosing functions	93
21.1.1. Generating random numbers and keys	93
21.1.2. Digital signatures	93
21.1.3. Asymmetric encryption	94
21.1.4. Symmetric encryption	94
21.1.5. Message digest	94
21.1.6. Mechanisms	94
21.1.7. Key wrapping	94
21.1.8. Key agreement	95
22. General purpose functions	96
22.1. C_Finalize	96
22.1.1. Notes	96
22.2. C_GetInfo	96
22.3. C_GetFunctionList	96
22.4. C_GetInterface	96
22.5. C_GetInterfaceList	97
22.6. C_Initialize	97
22.6.1. Notes	97
23. Slot and token management functions	98
23.1. C_GetSlotInfo	98
23.2. C_GetTokenInfo	98
23.3. C_GetMechanismList	98
23.4. C_GetMechanismInfo	98
23.5. C_GetSlotList	98
23.5.1. Notes	99
23.6. C_InitToken	99
23.6.1. Notes	99
23.7. C_InitPIN	99
23.7.1. Notes	100
23.8. C_SetPIN	100
23.8.1. Notes	100
24. Standard session management functions	101
24.1. C_OpenSession	101
24.2. C_CloseSession	101
24.3. C_CloseAllSessions	101
24.4. C_GetOperationState	101
24.5. C_GetSessionInfo	101
24.6. C_SetOperationState	102

24.7. C_Login	102
24.8. C_Logout	102
25. nShield session management functions	103
25.1. C_LoginBegin	103
25.2. C_LoginNext	103
25.3. C_LoginEnd	103
26. Object management functions	104
26.1. C_CreateObject	104
26.1.1. CKK_NC_MILENAGERC	104
26.2. C_CopyObject	105
26.3. C_DestroyObject	105
26.4. C_GetObjectSize	105
26.5. C_GetAttributeValue	106
26.6. C_SetAttributeValue	106
26.7. C_FindObjectsInit	106
26.8. C_FindObjects	106
26.9. C_FindObjectsFinal	106
27. Encryption functions	107
27.1. C_EncryptInit	107
27.2. C_Encrypt	107
27.3. C_EncryptUpdate	107
27.4. C_EncryptFinal	107
28. Decryption functions	108
28.1. C_DecryptInit	108
28.2. C_Decrypt	108
28.3. C_DecryptUpdate	108
28.4. C_DecryptFinal	108
29. Message digesting functions	109
29.1. C_DigestInit	109
29.2. C_Digest	109
29.3. C_DigestUpdate	109
29.4. C_DigestFinal	109
30. Signing and MACing functions	110
30.1. C_SignInit	110
30.2. C_Sign	110
30.3. C_SignRecoverInit	110
30.4. C_SignRecover	110
30.5. C_SignUpdate	110
30.6. C_SignFinal	111

31. Functions for verifying signatures and MACs	112
31.1. C_VerifyInit	112
31.2. C_Verify	112
31.3. C_VerifyRecover	112
31.4. C_VerifyRecoverInit	112
31.5. C_VerifyUpdate	112
31.6. C_VerifyFinal	113
32. Dual-purpose cryptographic functions	114
32.1. C_DigestEncryptUpdate	114
32.2. C_DecryptDigestUpdate	114
32.3. C_SignEncryptUpdate	114
32.3.1. Notes	114
32.4. C_DecryptVerifyUpdate	114
32.4.1. Notes	115
33. Key-management functions	116
33.1. C_GenerateKey	116
33.2. C_GenerateKeyPair	117
33.3. C_WrapKey	117
33.4. C_UnwrapKey	117
33.5. C_DeriveKey	117
33.6. C_Decapsulate	117
33.7. C_Encapsulate	118
34. Random number functions	119
34.1. C_GenerateRandom	119
34.2. C_SeedRandom	119
34.2.1. Notes	119
35. Parallel function management functions	120
35.1. C_GetFunctionStatus	120
35.1.1. Notes	120
35.2. C_CancelFunction	120
35.2.1. Notes	120
36. Callback functions	121

1. Introduction

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

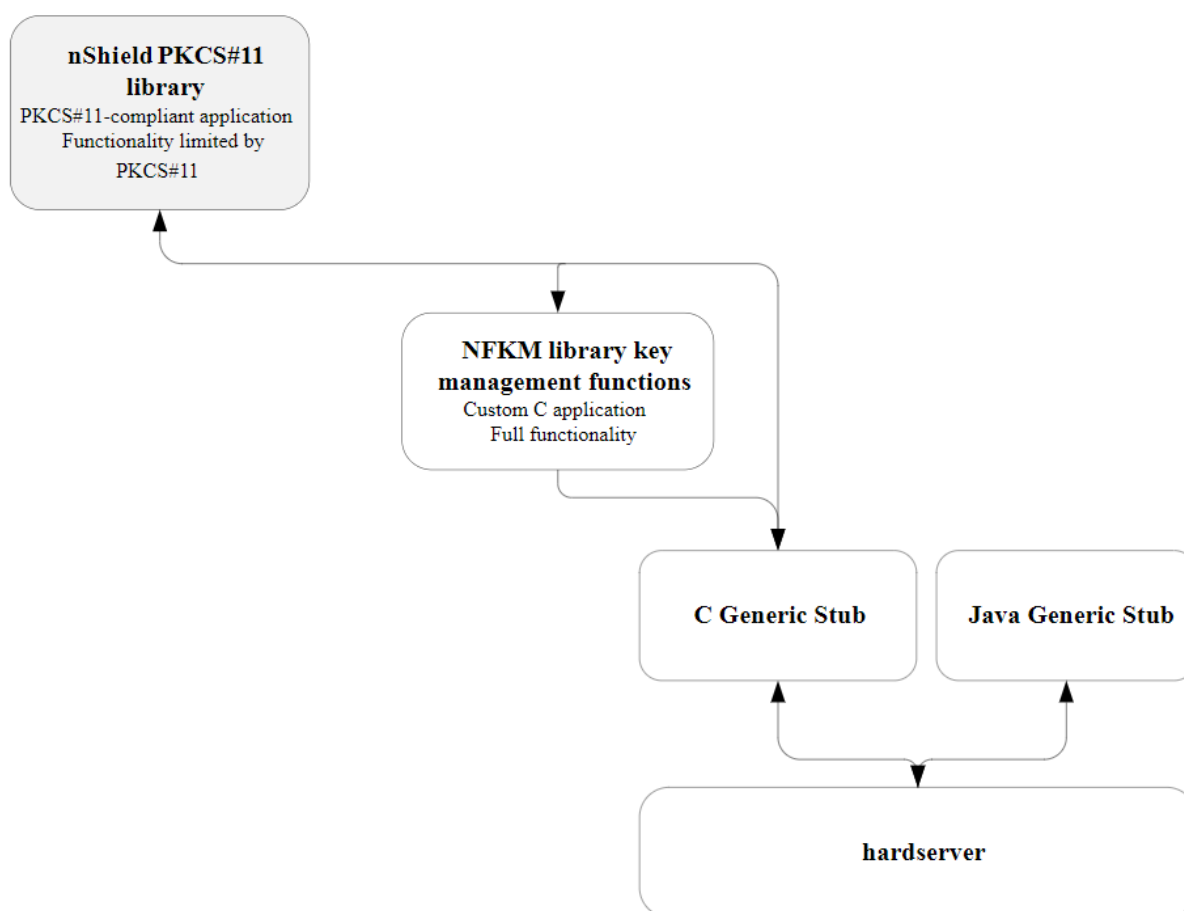
For an introduction to the PKCS #11 user library, see [nShield PKCS #11 library](#). You can find information about the available utilities in the [Utilities](#) reference guide. For information about the environment variables, see [nShield PKCS #11 library environment variables](#).

Before using the nShield PKCS #11 libraries, we recommend that you read <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.2/pkcs11-spec-v3.2.html>



To obtain full use of functions such as `C_Encapsulate()` and `C_Decapsulate()` it is required that applications access these functions by calling `C_GetInterface()` or `C_GetInterfaceList()` as-per the 3.2 version of the PKCS#11 specification.

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



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

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

This toolkit, like the application plug-ins supplied by Entrust, uses the Security World paradigm for key storage. For an introduction to Security Worlds, see [nShield Security World v13.9.3 Management Guide](#).

1.1. Read this guide if...

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

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

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

1.2. Additional useful documentation

Refer to [nShield Security World v13.9.3 Management Guide](#) and [nShield v13.9.3 HSM User Guide](#) for additional information about Security Worlds and nShield HSMs.

1.3. Security World Software default directories

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

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

Directory Name	Environment Variable	Windows Server 2016	Linux
Static Feature Certificates		C:\ProgramData\nCipher\Features	/opt/nfast/kmdata/features/
Log Files	NFAST_LOGDIR	C:\ProgramData\nCipher\Log Files	/opt/nfast/log/



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



Dynamic feature certificates must be stored in the directory stated in the default directories table.

The directory shown for static feature certificates is an example location. You can store those certificates in any directory and provide the appropriate path when using the Feature Enable Tool. However, you must not store static feature certificates in the dynamic features certificates directory. For more information about feature certificates, see [Optional features](#).

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

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

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

NFAST_KMDATA cannot be a symbolic link.

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

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

1.4. Utility help options

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

- h|--help** displays help for the utility
- v|--version** displays the version number of the utility
- u|--usage** displays a brief usage summary for the utility.

1.5. Further information

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

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

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

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

1.6. Security advisories

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



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

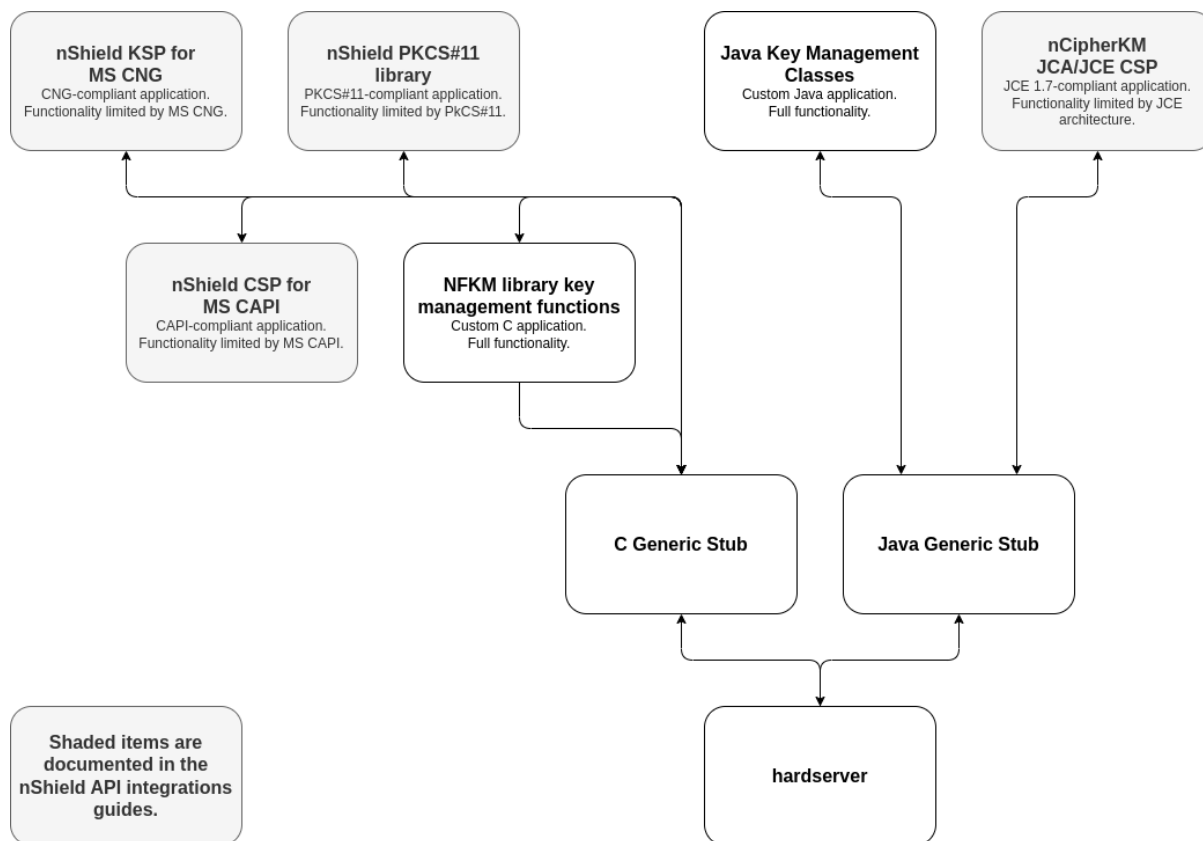
We recommend that you monitor the Announcements & Security Notices section on Entrust nShield, <https://trustedcare.entrust.com/>, where any announcement of nShield Security Advisories will be made.

1.7. Contacting Entrust nShield Support

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

2. nShield Architecture

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



2.1. Security World Software modules

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

2.2. Security World Software server

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

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

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

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

2.3. Stubs and interface libraries

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

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

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

2.4. Using an interface library

Entrust supplies the following interface libraries:

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

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

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

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

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

2.5. Writing a custom application

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

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

2.6. Acceleration-only or key management

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

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

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

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

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

3. PKCS #11 Developer libraries

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

The nShield PKCS #11 libraries:

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

3.1. Checking the installation of the nShield PKCS #11 library

After you have created a Security World, ensure that the nShield PKCS #11 library has been successfully installed with [ckcheckinst](#).

3.2. PKCS #11 security assurance mechanism

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

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

If you decide that some operations that carry a higher security risk are acceptable to you, then you can reconfigure the nShield PKCS #11 library to permit these operations by means of the environment variable `CKNFAST_OVERRIDE_SECURITY_ASSURANCES`. You must think carefully, however, before permitting operations that could compromise the security of cryptographic keys.



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

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

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

3.2.1. Key security

Questionable operations largely relate to the concept of a key being *secure*. A private or secret key is considered insecure if there is some reason for believing that its value may be available outside the HSM. Public keys are never considered insecure; by definition they are intended to be public.

An explicitly insecure PKCS #11 key is one where **CKA_SENSITIVE** is set to false. If an application uses a key that is insecure but **CKA_SENSITIVE** is not set to false, it is possible that the application is using an inadequate concept of key security, and that the library disallows use of that key by default. Use of insecure keys should, by default, be restricted to short-term session keys, and applications should explicitly recognize the insecurity.

4. PKCS #11 with load sharing mode

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

An additional virtual slot may be returned (with the label of **accelerator**), depending on the value given to the variable **CKNFAST_NO_ACCELERATOR_SLOTS** (described in [CKNFAST_NO_ACCELERATOR_SLOTS](#)). Accelerator slots can:

- Be used to support session objects
- Be used to create module-protected keys
- Not be used to create private objects.



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

Whether or not load-sharing mode is enabled is determined by the state of the [CKNFAST_LOADSHARING](#) environment variable.

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

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

The PKCS #11 token is present until you remove the last card belonging to the OCS. When you remove the token, the nShield PKCS #11 library closes any open sessions.

Enabling load-sharing mode also enables high-availability mode. When in high-availability mode, the nShield PKCS #11 library will detect modules being added to, or removed from, the current Security World and, provided that at least one module remains available, applications will not require restarting to adapt to changes in module availability.



The minimum interval that modules are checked when in high-availability mode can be configured by setting [CKNFAST_HA_MINIMUM_INTERVAL](#).



High-availability mode is disabled if **preLoad** is used.



If operating within a FIPS Security World, it is necessary to ensure at least one operator card is available to provide FIPS authorization.

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

4.1. Logging in

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

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

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

If you remove a smart card that belongs to a logged-in token, the nShield PKCS #11 library closes any open sessions and marks the token as being not present (unless the OCS is persistent). Removing a card from a persistent OCS has no effect, and the PKCS #11 token remains present until you log out.

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

4.2. Session objects

Session objects are loaded on all modules.

4.3. Module failure

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

4.4. Compatibility

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

4.5. Restrictions on function calls in load-sharing mode

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

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

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

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



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



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

5. PKCS #11 with HSM Pool mode

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

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

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

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

5.1. Module failure

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

5.2. Module recovery

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

5.3. Restrictions on function calls in HSM Pool mode

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

- `C_LoginBegin`
- `C_LoginNext`

- `C_LoginEnd`
- `C_InitToken`
- `C_InitPIN`
- `C_SetPIN`

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

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

6.1. Generating NVRAM-stored keys

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

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

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

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

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

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

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



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

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

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

6.2. Deleting NVRAM-stored keys

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

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

```
preload --admin=Nv pause
```

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



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

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

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

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

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



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

7. Unset the `CKNFAST_NVRAM_KEY_STORAGE` environment variable.

7. PKCS #11 with key reloading

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

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

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



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

Key reloading is not supported for session keys.

7.1. Usage under preload

PKCS #11 key reloading only reloads keys. It must also operate under a preload session during which preload is reloading tokens that protect the keys used by PKCS #11, in high availability mode. When the PKCS #11 application is using a token-protected key, `preload` should first be run to reload the token while PKCS #11 is reloading the key. For information on running `preload` for PKCS #11 key reloading, see [PKCS #11 with preload](#) and [Preload Utility](#).



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

Either run the PKCS #11 application as a subprocess of preload, or in a separate command window ensuring the preload file set for preload matches the one set for PKCS #11. See [PKCS #11 with preload](#) and [Preload Utility](#)

The application will attempt to reload keys when supported functions are called, see [Sup-](#)

[ported function calls](#).

7.1.1. Persistent preload files

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

7.2. Supported function calls

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

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

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

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

7.3. Retrying key reloads

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

7.4. Adding new HSMs

With key reloading enabled using the `CKNFAST_RELOAD_KEYS` environment variable, the PKCS #11 library can add new HSMs to its internal list of usable modules. HSMs are new if they were not present when PKCS #11 applications were initialized. When key reloading is not

enabled, PKCS #11 applications must be restarted before the new HSMs can be used.

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

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

The nShield PKCS #11 library makes each nShield module appear to your PKCS #11 application as two or more PKCS #11 slots, unless you have set `CKNFAST_NO_ACCELERATOR_SLOTS`.



The entry called `accelerator` cannot be used to create private objects. It can be used to create module-protected keys.

The first slot represents the module itself. This token:

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



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

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

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



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

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



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

8.1. K/N support for PKCS #11

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

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

9. PKCS #11 with preload

You can use the `preload` command-line utility to preload *K/N* OCSs before actually using PKCS #11 applications. The `preload` utility loads the logical token and then passes it to the PKCS #11 utilities.

You must provide any required passphrase for the tokens when using `preload` to load the card set. However, because the application is not aware that the card set has been preloaded, the application operates normally when handling the login activity (including prompting for a passphrase), but the PKCS #11 library will not actually check the supplied passphrase. `preload` must be also used with the `cksotool` utility to perform operations that require the PKCS #11 Security Officer role.

Normally, `preload` uses environment variables to pass information to the program using the preloaded objects, including the PKCS #11 library. Therefore, if the application you are using is one that clears its environment before the PKCS #11 library is loaded, you must set the appropriate values in the `cknfastrc` file (see [nShield PKCS #11 library environment variables](#)). The current environment variables remain usable. The default setting for the `CKN-FAST_LOADSHARING` environment variable changes from specifying load-sharing as disabled to specifying load-sharing as enabled. Moreover, in load-sharing mode, the loaded card set is used to set the environment variable `CKNFAST_CARDSET_HASH` so that only the loaded card set is visible as a slot.

The `NFAST_NFKM_TOKENSFILE` environment variable must also be set in the `cknfastrc` file to the location of the preload file (see [nShield PKCS #11 library environment variables](#)).

A logical token preloaded by `preload` for use with the nShield PKCS #11 library is the only such token available to the application for the complete invocation of the library. You can use more than one HSM with the same card set.

If the loaded card set is non-persistent, then a card must be left in each HSM on which the set has been loaded during the start-up sequence. After a non-persistent card has been removed, the token is not present even if the card is reinserted.

If load-sharing has been specifically switched off, you see multiple slots with the same label.

10. PKCS #11 Security Officer

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



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

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

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

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

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

11. nShield-specific PKCS #11 API extensions

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



The login sequence must occur in the same session.



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

11.1. C_LoginBegin

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

The `pu1K` and `pu1N` 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 pu1K, /* cards required to load logical token*/
             CK_ULONG_PTR pu1N /* Number of cards in set */)
```

11.2. C_LoginNext

`C_LoginNext` is called *K* times until the required number of cards (for the given card set) have been presented. This function checks the Security World info to ensure that the card has changed each time. It also checks for the correct passphrase before loading the card share. `pu1SharesLeft` allows the user application to assess the number of cards loaded to the number of cards required.

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

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

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

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

11.3. C_LoginEnd

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

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



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

If other functions are accidentally called during the log-in cycle, then **slot.loadcardset-state** 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.

12. Compiling and linking

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

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

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

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

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



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

12.1. Windows

All versions are built with Visual Studio 2022. Entrust supplies the following files:

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



Both files are identical.

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

12.2. Linux

Entrust supplies the following libraries:

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

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

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

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

13. nShield PKCS #11 library environment variables

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

- CKNFAST_ASSUME_SINGLE_PROCESS
- CKNFAST_ASSURANCE_LOG
- CKNFAST_CARDSET_HASH
- CKNFAST_CONCATENATIONKDF_X963_COMPLIANCE
- CKNFAST_DEBUG
- CKNFAST_DEBUGDIR
- CKNFAST_DEBUGFILE
- CKNFAST_DH_LSB
- CKNFAST_EDDSA_PUBKEY_FORMAT
- CKNFAST_FAKE_ACCELERATOR_LOGIN
- CKNFAST_HA_MINIMUM_INTERVAL
- CKNFAST_HSM_POOL
- CKNFAST_JCE_COMPATIBILITY
- CKNFAST_LOADSHARING
- CKNFAST_LOAD_KEYS
- CKNFAST_NO_ACCELERATOR_SLOTS
- CKNFAST_NO_SYMMETRIC
- CKNFAST_NO_UNWRAP
- CKNFAST_NONREMOVABLE
- CKNFAST_NVRAM_KEY_STORAGE
- CKNFAST_OVERRIDE_SECURITY_ASSURANCES
- CKNFAST_RELOAD_KEYS
- CKNFAST_SEED_MAC_ZERO
- CKNFAST_SESSION_THREADSafe
- CKNFAST_SESSION_TO_TOKEN
- CKNFAST_SHARE_SESSION_KEYS
- CKNFAST_TOKENS_PERSISTENT
- CKNFAST_USE_THREAD_UPCALLS
- 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`.

Linux

This file must be in the `/opt/nfast/` directory of the client.

Windows

If the `NFAST_HOME` environment variable is not set, or if environment variables are cleared by your application, the file `cknfastrc` must be in the `%NFAST_HOME%` directory of the client.



The `cknfastrc` file should be saved without any suffix (such as `.txt`).

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

```
<variable>=<value>
```



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`.



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.

13.1. 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_SetAttributeValue` or `C_GetAttributeValue` 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.

13.2. CKNFAST_ASSURANCE_LOG

This variable is used to direct all warnings from the Security Assurance Mechanism to a specific log file.

13.3. 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 [nfkminfo](#).

13.4. CKNFAST_CONCATENATIONKDF_X963_COMPLIANCE

Sets the correct use of ECDH derive with concatenate KDF using the ANSI X9.63 specification as per the PKCS#11 standard.



The default is ANSI X9.63 to match that of the PKCS #11 Specification.



ECDH derive with concatenate KDF SP800-56a can use the standard PKCS #11 v3 `CKD_SHA[x]_SP800_KDF` values.

13.5. 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`.

Value	Description
<code>0</code>	None (default setting)

Value	Description
1	Fatal error
2	General error
3	Fix-up error
4	Warnings
5	Application errors
6	Assumptions made by the nShield PKCS #11 library
7	API function calls
8	API return values
9	API function argument values
10	Details
11	Mutex locking detail

13.6. 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.

13.7. CKNFAST_DEBUGFILE

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

13.8. CKNFAST_DH_LSB

If this variable is set the least significant bytes of the result of DH/ECDH key agreement using the `CKM_DH_PKCS_DERIVE`, `CKM_X9_42_DH_DERIVE` or `CKM_ECDH1_DERIVE` mechanisms are taken. This is in line with the PKCS#11 specification. If this variable is not set the most significant bytes will be used. The latter behavior is consistent with Security World software prior to v12.81.

13.9. CKNFAST_EDDSA_PUBKEY_FORMAT

If applications require the `CKA_EC_POINT` output (`C_GetAttributeValue`) as an ASN.1 Bit

String, set this variable to **bits**, otherwise it will be supplied as an ASN.1 Octet String.

This only applies to EDDSA keys.

13.10. CKNFAST_FAKE_ACCELERATOR_LOGIN

If this variable is set, the nShield 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 HSM can enter an arbitrary PIN and obtain the certificate.

CKNFAST_FAKE_ACCELERATOR slots allow the creation of objects with **CKA_PRIVATE=TRUE** in the template even though the login is "fake" and the objects are not private.

- Examining the attributes shows **CKA_PRIVATE** as **FALSE**.
- A search for the object will not find it if the search criteria includes **CKA_PRIVATE=TRUE**.

13.11. CKNFAST_HA_MINIMUM_INTERVAL

When high-availability mode is enabled by **CKNFAST_LOADSHARING**, the minimum interval (in seconds) that modules are checked can be configured by setting **CKNFAST_HA_MINIMUM_INTERVAL**:

- If unset, a default interval of 60 seconds is used.
- If set to **0**, high-availability mode will be disabled.

This check is performed when the nShield PKCS #11 library is used for operations that require use of a module.

The time it takes for the nShield PKCS #11 library to detect modules can vary and is dependent on both the state a module is transitioning from and, for example, whether an application is active and whether remote operator cards are in use.

13.12. CKNFAST_HSM_POOL

HSM Pool mode is determined by the state of the **CKNFAST_HSM_POOL** environment variable.

Set the environment variable to **1**, **y** or **Y** to enable HSM Pool mode for the PKCS #11 application, or set to **0**, **n** or **N** to explicitly disable HSM Pool mode for the PKCS #11 application.

HSM Pool mode takes precedence over load-sharing mode. HSM Pool mode only supports module protected keys so do not use **CKNFAST_NO_ACCELERATOR_SLOTS** to disable the accelerator.

ator slot.

13.13. CKNFAST_JCE_COMPATIBILITY

This property is included to allow the saving of objects when using Java PKCS#11 providers.

It is possible, using `C_CopyObject`, to change a key's `CKA_TOKEN` value from `CK_FALSE` to `CK_TRUE`. This requires the `CKNFAST_JCE_COMPATIBILITY` environment variable to be set to `1`. The original key's `CKA_TOKEN` value will remain unchanged.

13.14. CKNFAST_LOADSHARING

Load-sharing mode 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` and ensure that the `CKNFAST_HSM_POOL` environment variable is not set. The virtual slot behavior then operates.

Enabling load-sharing mode also enables high-availability mode, see: `CKNFAST_HA_MINIMUM_INTERVAL`.



High-availability mode is disabled if `preload` is used.



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).

13.15. CKNFAST_NO_ACCELERATOR_SLOTS

If this variable is set, the nShield 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.



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

13.16. CKNFAST_NO_SYMMETRIC

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

13.17. CKNFAST_NO_UNWRAP

If this variable is set, the nShield 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.

13.18. CKNFAST_NONREMOVABLE

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



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.

13.19. CKNFAST_NVRAM_KEY_STORAGE

When this environment variable is set, the PKCS #11 library generates only keys in non-volatile memory (NVRAM). You must also ensure this environment variable is set in order to delete NVRAM-stored keys.

13.20. 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`
- `wrapping_crypt`
- `unwrap_kek`
- `derive_kek`
- `derive_xor`
- `derive_concatenate`
- `unwrap_rsa_aes_kwp`
- `weak_<algorithm>`
- `shortkey_<algorithm>=<bitlength>`
- `silent.`

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

Linux

```
CKNFAST_OVERRIDE_SECURITY_ASSURANCES="<parameter1>;<parameter2>=<value3>"
```

Windows

```
set CKNFAST_OVERRIDE_SECURITY_ASSURANCES=<parameter1>;<parameter2>=<value3>
```

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

```
CKNFAST_OVERRIDE_SECURITY_ASSURANCES=<parameter1>;<parameter2>=<value3>
```

Unknown parameters generate a warning; see [Diagnostic warnings about questionable operations](#).

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

13.20.1. 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.

13.20.2. 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.

13.20.3. 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 HSM 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.

13.20.4. 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.



A need to set this variable usually means that some important keys that should be protected by the HSM'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.

13.20.5. explicitness

The `explicitness` parameter permits applications to create insecure keys without explicitly recognizing that they are insecure. An insecure key is a key that is deemed sensitive, but can be wrapped and extracted from the HSM by any untrusted key. A secure key must have the `CKA_WRAP_WITH_TRUSTED` attribute.



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.

13.20.6. import

The `import` parameter allows keys that are to be imported into the HSM'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.



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.

13.20.7. wrapping_crypt

The `wrapping_crypt` parameter allows you to create keys with insecure combinations of wrap/unwrap and encrypt/decrypt operations.

By default, when `wrapping_crypt` is not supplied as a parameter for `CKNFAST_OVERRIDE_SECURITY_ASSURANCES`, trying to create a key with either `CKA_UNWRAP=true` or `CKA_WRAP=true` and `CKA_DECRYPT=true` or `CKA_ENCRYPT=true` will fail with `CKR_TEMPLATE_INCONSISTENT`.

Combinations such as wrap+encrypt or unwrap+encrypt are prohibited because for some mechanisms (e.g. counter mode), encrypt and decrypt are the same operation, so allowing encrypt is functionally the same as allowing decrypt.

13.20.8. unwrap_kek

When a key is transferred into the HSM 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).

13.20.9. 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).

13.20.10. 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).

13.20.11. 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.

13.20.12. unwrap_rsa_aes_kwp

The `unwrap_rsa_aes_kwp` parameter only applies to firmware version 13.3 or earlier. It is not needed in later versions.

The `C_UnwrapKey` operation with `CKM_RSA_AES_KEY_WRAP` imports the temporary AES key with an nCore API ACL that permits unwrapping of the wrapped target key by the temporary AES key. When using the `C_UnwrapKey` operation with only a user supplied template (`pTemplate`) it is possible to create this ACL such that it permits a one-time unwrap of only the wrapped target key. When the RSA unwrapping key has `CKA_UNWRAP_TEMPLATE` set it is necessary to construct the ACL when the RSA key is created in order to setup the partitioning guarantees from the `CKA_UNWRAP_TEMPLATE`. The intended wrapped target keys are unknown at this time, which means the ACL must permit a one-time unwrap of any key.

The Security Assurance Mechanism (SAM) considers this scenario insecure by default and therefore the use of the `C_UnwrapKey` operation with `CKM_RSA_AES_KEY_WRAP` is disabled when the RSA unwrapping key has `CKA_UNWRAP_TEMPLATE` set. When the `unwrap_rsa_aes_kwp` parameter is set the SAM enables the `C_UnwrapKey` operation with `CKM_RSA_AES_KEY_WRAP` in this scenario. The RSA unwrapping key must also explicitly allow the `CKM_RSA_AES_KEY_WRAP` mechanism via `CKA_ALLOWED_MECHANISMS` in addition to setting the `unwrap_rsa_aes_kwp` (or `all`) parameter; otherwise, the `C_UnwrapKey` operation will remain disabled when the RSA unwrapping key has `CKA_UNWRAP_TEMPLATE` set.

13.20.13. 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.

13.20.14. 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.

13.20.15. 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.

13.20.16. 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, these messages are sent to `stderr`. On Windows platforms, they are also always 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.



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`.

13.21. CKNFAST_SEED_MAC_ZERO

Set this variable to use zero padding for the Korean SEED MAC mechanisms (`CK_SEED_MAC` and `CKM_SEED_MAC_GENERAL`). If this variable is not set, or is set to `n`, then the SEED MAC mechanisms will use the default PKCS #5 padding scheme.

13.22. 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.

13.23. CKNFAST_SESSION_TO_TOKEN

This environment variable controls whether session keys can be copied to token keys using the nShield PKCS #11 library. If you generate persistent keys using a JCE PKCS #11 provider, such as SunPKCS11 or IBMPKCS11Impl, set this variable.

If `CKNFAST_SESSION_TO_TOKEN` is set (the default), then `C_CopyObject` may be used to copy a session key to a token key, that is, to convert a session key to token key.

- If `CKNFAST_SESSION_TO_TOKEN` is enabled, all keys are created with Key Generation Certificates.
- If `CKNFAST_SESSION_TO_TOKEN` is disabled, Session Keys are generated without certificates.

Unsetting this `CKNFAST_SESSION_TO_TOKEN` allows faster generation of session keys, but disables the ability to convert a session key to a token key.

13.24. CKNFAST_SHARE_SESSION_KEYS

This variable can take a list of one or more semicolon (;) separated values to improve performance through loadsharing when session keys are used. See [CKNFAST_LOADSHARING](#).

Loadsharing improves performance and adds resilience in the case of module failure. However, if the key is used only a few times, the overhead of sharing it may be greater than the performance benefit. If a key will be used many times or if it has a long lifespan, sharing is recommended.

- `all` (default)
- `copy`
- `derive`
- `generate`
- `import`
- `none`
- `unwrap`

If the origin of the session key matches a selected category, then the key is automatically shared to all HSMs when it is created.

13.25. 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.



Use of the nShield PKCS #11 library to create tokens is deprecated, because it can only create 1/1 tokens in FIPS 140 Level 2 Security Worlds. Use one of the command-line utilities to create OCSs.

13.26. 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_usencthread`s 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.

13.27. CKNFAST_LOAD_KEYS

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

13.28. 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.



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

13.29. CKNFAST_RELOAD_KEYS

Set this variable to enable PKCS #11 key reloading. See [PKCS #11 with key reloading](#).

Key reloading requires load sharing-mode to operate, and enables it automatically if `CKNFAST_LOADSHARING` is not set.

14. Objects

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

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

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

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

The following table lists the protection for different types of PKCS #11 token objects:

	Smart card Slot	Accelerator Slot
Private Token Object	Operator Card Set	not supported
Public Token Object	Security World	Security World
Public key	well known HSM key	well known HSM key

Operator Card Set

The object is stored as an nShield key blob encrypted by the OCS key. You must log in to this OCS before you can load this object.

security world

The object is stored as an nShield key blob encrypted by the Security World key. This object can be loaded on to any HSM in the Security World. The nShield PKCS #11 library only allows access if a card from this OCS is present.

well-known module key

Public keys are encrypted under a well-known HSM key. This encryption is for programming convenience only and does not provide security. These keys can be loaded on any nShield HSM.

14.1. Certificate Objects and Data Objects

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

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

14.2. Key Objects

The following restrictions apply to keys:

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

14.3. Card passphrases

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

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

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

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

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



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

15. Mechanisms

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



Some mechanisms may be restricted from use in Security Worlds conforming to FIPS 140 Level 3.

Mechanism	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive Key	Encapsulate & Decapsulate
CKM_AES_CBC_ENCRYPT_DATA	—	—	—	—	—	—	Y	—
CKM_AES_CBC_PAD	Y	—	—	—	—	Y	—	—
CKM_AES_CBC	Y	—	—	—	—	Y ¹	—	—
CKM_AES_CMAC_GENERAL	—	Y	—	—	—	—	—	—
CKM_AES_CMAC	—	Y	—	—	—	—	—	—
CKM_AES_CTR	Y	—	—	—	—	X	—	—
CKM_AES_ECB_ENCRYPT_DATA	—	—	—	—	—	—	Y	—
CKM_AES_ECB	Y	—	—	—	—	Y ¹	—	—
CKM_AES_GCM	Y	—	—	—	—	Y ¹³	—	—
CKM_AES_KEY_GEN	—	—	—	—	Y	—	—	—
CKM_AES_KEY_WRAP	—	—	—	—	—	Y	—	—
CKM_AES_KEY_WRAP_PAD ²	Y	—	—	—	—	Y	—	—
CKM_AES_KEY_WRAP_KEY_WRAP	Y	—	—	—	—	Y	—	—
CKM_AES_MAC_GENERAL	—	Y	—	—	—	—	—	—
CKM_AES_MAC	—	Y	—	—	—	—	—	—
CKM_ARIA_CBC ¹⁶	Y	—	—	—	—	Y ¹⁷	—	—
CKM_ARIA_CBC_PAD ¹⁶	Y	—	—	—	—	—	—	—

Mechanism	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive Key	Encapsulate & Decapsulate
CKM_ARIA_ECB ¹⁶	Y	—	—	—	—	Y ¹⁷	—	—
CKM_ARIA_KEY_GEN ¹⁶	—	—	—	—	Y	—	—	—
CKM_ARIA_MAC ¹⁶	—	Y	—	—	—	—	—	—
CKM_ARIA_MAC_GENERAL ¹⁶	—	Y	—	—	—	—	—	—
CKM_CONCATENATE_BASE_AND_KEY	—	—	—	—	—	—	Y ³	—
CKM_DES_CBC_ENCRYPT_DATA	—	—	—	—	—	—	Y	—
CKM_DES_CBC_PAD	Y	—	—	—	—	Y	—	—
CKM_DES_CBC	Y	—	—	—	—	Y	—	—
CKM_DES_ECB_ENCRYPT_DATA	—	—	—	—	—	—	Y	—
CKM_DES_ECB	Y	—	—	—	—	Y	—	—
CKM_DES_KEY_GEN	—	—	—	—	Y	—	—	—
CKM_DES_MAC_GENERAL	—	Y	—	—	—	—	—	—
CKM_DES_MAC	—	Y	—	—	—	—	—	—
CKM_DES2_KEY_GEN	—	—	—	—	Y	—	—	—
CKM_DES3_CBC_ENCRYPT_DATA	—	—	—	—	—	—	Y	—
CKM_DES3_CBC_PAD	Y	—	—	—	—	Y	—	—
CKM_DES3_CBC	Y	—	—	—	—	Y ¹	—	—
CKM_DES3_ECB_ENCRYPT_DATA	—	—	—	—	—	—	Y	—
CKM_DES3_ECB	Y	—	—	—	—	Y ¹	—	—
CKM_DES3_KEY_GEN	—	—	—	—	Y	—	—	—
CKM_DES3_MAC_GENERAL	—	Y	—	—	—	—	—	—
CKM_DES3_MAC	—	Y	—	—	—	—	—	—
CKM_DH_PKCS_DERIVE	—	—	—	—	—	—	Y	—

Mechanism	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive Key	Encapsulate & Decapsulate
CKM_DH_PKCS_KEY_PAIR_GEN	—	—	—	—	Y	—	—	—
CKM_DSA_KEY_PAIR_GEN	—	—	—	—	Y	—	—	—
CKM_DSA_PARAMETER_GEN	—	—	—	—	Y	—	—	—
CKM_DSA_SHA1	—	Y	—	—	—	—	—	—
CKM_DSA	—	Y ⁴	—	—	—	—	—	—
CKM_EC_EDWARDS_KEY_PAIR_GEN	—	—	—	—	Y ¹⁹	—	—	—
CKM_EC_KEY_PAIR_GEN	—	—	—	—	Y ⁶	—	—	—
CKM_EC_MONTGOMERY_KEY_PAIR_GEN	—	—	—	—	Y ⁵	—	—	—
CKM_ECDH1_DERIVE	—	—	—	—	—	—	Y ⁷	—
CKM_ECDSA_SHA1	—	Y	—	—	—	—	—	—
CKM_ECDSA_SHA224	—	Y	—	—	—	—	—	—
CKM_ECDSA_SHA256	—	Y	—	—	—	—	—	—
CKM_ECDSA_SHA384	—	Y	—	—	—	—	—	—
CKM_ECDSA_SHA512	—	Y	—	—	—	—	—	—
CKM_ECDSA_SHA3_224	—	Y	—	—	—	—	—	—
CKM_ECDSA_SHA3_256	—	Y	—	—	—	—	—	—
CKM_ECDSA_SHA3_384	—	Y	—	—	—	—	—	—
CKM_ECDSA_SHA3_512	—	Y	—	—	—	—	—	—
CKM_EDDSA	—	Y ^{4,8}	—	—	—	—	—	—
CKM_ECDSA	—	Y ⁴	—	—	—	—	—	—
CKM_GENERIC_SECRET_KEY_GEN	—	—	—	—	Y	—	—	—
CKM_HASH_ML_DSA ²⁰	—	Y ⁴	—	—	—	—	—	—

Mechanism	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive Key	Encapsulate & Decapsulate
CKM_HASH_ML_D-SA_SHA256 ²⁰	—	Y ⁴	—	—	—	—	—	—
CKM_HASH_ML_D-SA_SHA512 ²⁰	—	Y ⁴	—	—	—	—	—	—
CKM_HASH_ML_D-SA_SHAKE128 ²⁰	—	Y ⁴	—	—	—	—	—	—
CKM_HASH_ML_D-SA_SHAKE256 ²⁰	—	Y ⁴	—	—	—	—	—	—
CKM_MD5_HMAC_GENERAL	—	Y	—	—	—	—	—	—
CKM_MD5_HMAC	—	Y	—	—	—	—	—	—
CKM_MD5	—	—	—	Y	—	—	—	—
CKM_ML_DSA_KEY_PAIR_GEN ²⁰	—	—	—	—	Y	—	—	—
CKM_ML_DSA ²⁰	—	Y ⁴	—	—	—	—	—	—
CKM_NC_ECIES	—	—	—	—	—	Y ⁹	—	—
CKM_NC_MD5_HMAC_KEY_GEN	—	—	—	—	Y	—	—	—
CKM_NC_MILENAGE	—	Y ^{4,15}	—	—	—	—	—	—
CKM_NC_MILENAGE_AUTS	—	Y ^{4,15}	—	—	—	—	—	—
CKM_NC_MILENAGE_RESYNC	—	Y ^{4,15}	—	—	—	—	—	—
CKM_NC_MILENAGE_OPC	—	—	—	—	—	—	Y	—
CKM_NC_MILENAGE_GEOP_KEY_GEN	—	—	—	—	Y	—	—	—
CKM_NC_MILENAGER-C_KEY_GEN	—	—	—	—	Y	—	—	—
CKM_NC_MILENAGESUBSCRIBER_KEY_GEN	—	—	—	—	Y	—	—	—
CKM_NC_TUAK	—	Y ^{4,15}	—	—	—	—	—	—
CKM_NC_TUAK_AUTS	—	Y ^{4,15}	—	—	—	—	—	—

Mechanism	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive Key	Encapsulate & Decapsulate
CKM_NC_TUAK_RESYNC	—	Y ^{4,15}	—	—	—	—	—	—
CKM_NC_TUAK_TOPC	—	—	—	—	—	—	Y	—
CKM_NC_TUAKSUB- SCRIBER_KEY_GEN	—	—	—	—	Y	—	—	—
CKM_NC_TUAK- TOP_KEY_GEN	—	—	—	—	Y	—	—	—
CKM_P- BE_MD5_DES_CBC	—	—	—	—	Y	—	—	—
CKM_RIPEMD160	—	—	—	Y	—	—	—	—
CKM_RSA_9796	—	Y ⁴	Y ⁴	—	—	—	—	—
CKM_R- SA_AES_KEY_WRAP	—	—	—	—	—	Y ¹⁴	—	—
CKM_RSA_PKCS_KEY_- PAIR_GEN	—	—	—	—	Y	—	—	—
CKM_RSA_PKCS_OAEP	Y	—	—	—	—	Y	—	—
CKM_RSA_PKCS_PSS ¹¹	Y	Y	—	—	—	—	—	—
CKM_RSA_PKCS	Y ⁴	Y ⁴	Y ⁴	—	—	Y	—	—
CKM_RSA_X_509	Y ⁴	Y ⁴	Y ⁴	—	—	X	—	—
CKM_RSA_X9_31_KEY_- PAIR_GEN	—	—	—	—	Y	—	—	—
CKM_SHA_1_HMAC_GEN- ERAL	—	Y ¹⁰	—	—	—	—	—	—
CKM_SHA_1_HMAC	—	Y ¹⁰	—	—	—	—	—	—
CKM_SHA_1	—	—	—	Y	—	—	—	—
CKM_SHA1_RSA_PKCS- PSS ¹¹	—	Y	—	—	—	—	—	—
CKM_SHA1_RSA_PKCS	—	Y	—	—	—	—	—	—
CKM_SHA224_HMAC_- GENERAL	—	Y ¹⁰	—	—	—	—	—	—
CKM_SHA224_HMAC	—	Y ¹⁰	—	—	—	—	—	—
CKM_SHA224_RSA_PKCS- PSS ¹¹	—	Y	—	—	—	—	—	—

Mechanism	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive Key	Encapsu- late & Decapsu- late
CKM_SHA224_RSA_P- KCS	—	Y	—	—	—	—	—	—
CKM_SHA224	—	—	—	Y	—	—	—	—
CKM_SHA256_HMAC_- GENERAL	—	Y ¹⁰	—	—	—	—	—	—
CKM_SHA256_HMAC	—	Y ¹⁰	—	—	—	—	—	—
CKM_SHA256_RSA_PKC- S_PSS ¹¹	—	Y	—	—	—	—	—	—
CKM_SHA256_RSA_P- KCS	—	Y	—	—	—	—	—	—
CKM_SHA256	—	—	—	Y	—	—	—	—
CKM_SHA384_HMAC_- GENERAL	—	Y ¹⁰	—	—	—	—	—	—
CKM_SHA384_HMAC	—	Y ¹⁰	—	—	—	—	—	—
CKM_SHA384_RSA_PKC- S_PSS ¹¹	—	Y	—	—	—	—	—	—
CKM_SHA384_RSA_P- KCS	—	Y	—	—	—	—	—	—
CKM_SHA384	—	—	—	Y	—	—	—	—
CKM_SHA512_HMAC_- GENERAL	—	Y ¹⁰	—	—	—	—	—	—
CKM_SHA512_HMAC	—	Y ¹⁰	—	—	—	—	—	—
CKM_SHA512_RSA_PKC- S_PSS ¹¹	—	Y	—	—	—	—	—	—
CKM_SHA512_RSA_P- KCS	—	Y	—	—	—	—	—	—
CKM_SHA512	—	—	—	Y	—	—	—	—
CKM_SHA3_224	—	—	—	Y	—	—	—	—
CKM_SHA3_224_RSA_P- KCS_PSS ¹¹	—	Y	—	—	—	—	—	—
CKM_SHA3_224_RSA_P- KCS	—	Y	—	—	—	—	—	—
CKM_SHA3_256	—	—	—	Y	—	—	—	—

Mechanism	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive Key	Encapsulate & Decapsulate
CKM_SHA3_256_RSA_P-KCS_PSS ¹¹	—	Y	—	—	—	—	—	—
CKM_SHA3_256_RSA_P-KCS	—	Y	—	—	—	—	—	—
CKM_SHA3_384	—	—	—	Y	—	—	—	—
CKM_SHA3_384_RSA_P-KCS_PSS ¹¹	—	Y	—	—	—	—	—	—
CKM_SHA3_384_RSA_P-KCS	—	Y	—	—	—	—	—	—
CKM_SHA3_512	—	—	—	Y	—	—	—	—
CKM_SHA3_512_RSA_P-KCS_PSS ¹¹	—	Y	—	—	—	—	—	—
CKM_SHA3_512_RSA_P-KCS	—	Y	—	—	—	—	—	—
CKM_SP800_108_COUNTER_KDF ¹⁸	—	—	—	—	—	—	Y	—
CKM_XOR_BASE_AND_DATA	—	—	—	—	—	—	Y ¹²	—
CKM_ML_KEM_KEY_PAIR_GEN	—	—	—	—	Y	—	Y	—
CKM_ML_KEM	—	—	—	—	—	—	—	Y

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

In the table above:

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

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

15.1. Footnote 1

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

15.2. Footnote 2

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

15.3. Footnote 3

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

15.4. Footnote 4

Single-part operations only.

15.5. Footnote 5

`CKA_EC_PARAMS` is a DER-encoded PrintableString `curve25519`. This will be a byte array with the following values:

```
CK_BYTE curve25519[] = { 0x13, 0x0a, 0x63, 0x75, 0x72, 0x76,
                          0x65, 0x32, 0x35, 0x35, 0x31, 0x39 };
```

15.6. Footnote 6

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

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

curves.

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

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

15.7. Footnote 7

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

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

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

For the **pPublicData*** parameter, a raw octet string value (as defined in section A.5.2 of ANSI X9.62) and DER-encoded ECPoint value (as defined in section E.6 of ANSI X9.62 or, in the case of **CKK_EC_MONTGOMERY**, RFC 7748) are now accepted.

15.8. Footnote 8

The **Ed25519**, **Ed25519ph**, **Ed448** and **Ed448ph** signature schemes are supported.

The **Ed25519** scheme requires no **CK_EDDSA_PARAMS** to be passed.

The **Ed448** signature scheme requires **CK_EDDSA_PARAMS** to have the following set:

- `phFlag` to `CK_FALSE`
- `ulContextDataLen` to `0`.

The `Ed25519ph` and `Ed448ph` signature schemes require `CK_EDDSA_PARAMS` to have the following set:

- `phFlag` to `CK_TRUE`
- `ulContextDataLen` to `0`.

15.9. Footnote 9

Wrap secret keys only.

15.10. Footnote 10

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

15.11. Footnote 11

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

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

Mechanism	Salt-length
SHA-1	160-bit
SHA-224	224-bit
SHA-256	256-bit
SHA-384	384-bit
SHA-512	512-bit
SHA3-224	224-bit

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

15.12. Footnote 12

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

15.13. Footnote 13

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

15.14. Footnote 14

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

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

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



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

For more information about the SAMs, see [PKCS #11 security assurance mechanism](#). For more information about the **CKA_ALLOWED_MECHANISMS** attribute, see [Attributes](#).

15.15. Footnote 15

Sign only.

15.16. Footnote 16

Use of these mechanisms requires the **KISAAlgorithms** feature to be enabled.

15.17. Footnote 17

Wraps secret keys only.

15.18. Footnote 18

CKM_SP800_COUNTER_KDF restrictions:

- Supported in firmware versions v13.5 and later.
- The **CK_SP800_108_BYTE_ARRAY** field is limited to two repetitions, or three if one of them is a single zero byte.
- The PRF is restricted to SHA-224, SHA-256, SHA-384, SHA-512, or AES CMAC.
- The **ulWidthInbits** for the counter and dkm formats must be 8, 16, or 32.
- Only one key can be derived, so the **ulAdditionalDerivedKeys** must be 0.

15.19. Footnote 19

CKA_EC_PARAMS is a DER-encoded PrintableString **edwards25519** or **edwards448**. These will be byte arrays with the following values:

```
CK_BYTE edwards25519[] = { 0x13, 0x0c, 0x65, 0x64, 0x77, 0x61, 0x72,  
                           0x64, 0x73, 0x32, 0x35, 0x35, 0x31, 0x39 };  
CK_BYTE edwards448[] = { 0x13, 0x0c, 0x65, 0x64, 0x77, 0x61,  
                         0x72, 0x64, 0x73, 0x34, 0x34, 0x38 };
```

15.20. Footnote 20

Use of these mechanisms requires a firmware version of v13.8 or greater and the **PostQuantum** feature to be enabled.

16. Vendor annotations on P11 mechanisms

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

16.1. CKM_RSA_PKCS_OAEP

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

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

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

16.2. CKM_RSA_PKCS_PSS and CKM_SHA*_RSA_PKCS_PSS

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

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

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

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

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

17. Vendor-defined mechanisms

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



Some mechanisms may be restricted from use in Security Worlds conforming to FIPS 140 Level 3. See [Cryptographic algorithms](#) for more information.

17.1. CKM_SEED_ECB_ENCRYPT_DATA and CKM_SEED_CBC_ENCRYPT_DATA

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

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

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

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

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

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

17.2. CKM_CAC_TK_DERIVATION

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

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

- `CKA_TKC1`
- `CKA_TKC2`
- `CKA_TKC3`.

These attributes are all in the `CKA_VENDOR_DEFINED` range.

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

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

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

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

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

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

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

17.3. CKM_SHA*_HMAC and CKM_SHA*_HMAC_GENERAL

This version of the library supports the following mechanisms:

- `CKM_SHA_1_HMAC`
- `CKM_SHA_1_HMAC_GENERAL`
- `CKM_SHA224_HMAC`

- `CKM_SHA224_HMAC_GENERAL`
- `CKM_SHA256_HMAC`
- `CKM_SHA256_HMAC_GENERAL`
- `CKM_SHA384_HMAC`
- `CKM_SHA384_HMAC_GENERAL`
- `CKM_SHA512_HMAC`
- `CKM_SHA512_HMAC_GENERAL`

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

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

- `CKK_SHA_1_HMAC`
- `CKK_SHA224_HMAC`
- `CKK_SHA256_HMAC`
- `CKK_SHA384_HMAC`
- `CKK_SHA512_HMAC`

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

- `CKM_NC_MD5_HMAC_KEY_GEN`
- `CKM_NC_SHA_1_HMAC_KEY_GEN`
- `CKM_NC_SHA224_HMAC_KEY_GEN`
- `CKM_NC_SHA256_HMAC_KEY_GEN`
- `CKM_NC_SHA384_HMAC_KEY_GEN`
- `CKM_NC_SHA512_HMAC_KEY_GEN`

17.4. CKM_NC_ECKDF_HYPERLEDGER

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

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

```
typedef struct CK_ECKDF_HYPERLEDGERCLIENT_PARAMS {
```

```

CK_OBJECT_HANDLE hKeyDF_Key;
CK_MECHANISM_TYPE HMACMechType;
CK_MECHANISM_TYPE TCertEncMechType;
CK_ULONG ulEksize;
CK_BYTE_PTR pEncTCertData;
CK_ULONG ulEvsize;
CK_ULONG ulEndian;
} CK_ECKDF_HYPERLEDGERCLIENT_PARAMS

```

Where:

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

The function is then called as follows:

```

C_DeriveKey(
    hSession,
    &mechanism_hyperledger,
    EnrollPriv_Key,
    TCertPriv_Key_template,
    NUM(TCertPriv_Key_template,
    &TCertPriv_Key);

```

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

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

17.5. CKM_HAS160

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

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

17.6. CKM_PUBLIC_FROM_PRIVATE

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

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



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



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

17.7. CKM_NC_AES_CMAC

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

In a similar way to other AES MAC mechanisms, **CKM_NC_AES_CMAC** takes a plaintext type of any length of bytes, and returns a **M_Mech_Generic128MAC_Cipher** standard byte block.

CKM_NC_AES_CMAC is a standard FIPS 140 Level 3 approved mechanism, and is only usable with **CKK_AES** key types.

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

17.8. CKM_NC_AES_CMAC_KEY_DERIVATION and CKM_NC_AES_CMAC_KEY_DERIVATION_SCP03

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

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

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

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

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

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

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

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

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

Attribute	If the attributes for the <i>original</i> keys are...	The attribute for the <i>derived</i> 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

17.9. CK_NC_AES_CMAC_KEY_DERIVATION_PARAMS

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

The fields of the structure have the following meanings:

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

17.10. CKM_COMPOSITE_EMV_T_ARQC, CKM_WATCHWORD_PIN1 and CKM_WATCHWORD_PIN2

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

17.11. CKM_NC_ECIES

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

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

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

Where:

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

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

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

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

CK_ECDH1_DERIVE_PARAMS ecdh1_params = { CKD_SHA256_KDF };
CK_MECHANISM agreement_mech = {
    CKM_ECDH1_DERIVE,
    &ecdh1_params,
    sizeof(CK_ECDH1_DERIVE_PARAMS)
};

CK_MECHANISM symmetric_mech = { CKM_XOR_BASE_AND_DATA };
CK_MAC_GENERAL_PARAMS mac_params = { 16 };
CK_MECHANISM mac_mech = {
    CKM_SHA256_HMAC_GENERAL,
    &mac_params,
    sizeof(CK_MAC_GENERAL_PARAMS)
};

CK_NC_ECIES_PARAMS ecies_params = {
    &agreement_mech,
    &symmetric_mech,
    0,
    &mac_mech,
    256
};

CK_MECHANISM ecies_mech = {
    CKM_NC_ECIES,
    &ecies_params,
    sizeof(CK_NC_ECIES_PARAMS)
};

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

17.12. CKM_NC_MILENAGE_OPC

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

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

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

- **CKM_NC_MILENAGESUBSCRIBER_KEY_GEN**
- **CKM_NC_MILENAGEOP_KEY_GEN**

17.13. CKM_NC_MILENAGE, CKM_NC_MILENAGE_AUTS, CKM_NC_MILENAGE_RESYNC

3GPP mechanisms for 5G mobile networks as defined by ETSI TS 135 206. Used with **C_SignInit** and **C_Sign** function calls. The parameters for these mechanisms are defined in the following structure:

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

ulMilenageFlags can consist of the following flags:

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

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

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

17.13.1. CKM_NC_MILENAGE

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

The following example shows how to use **CKM_NC_MILENAGE**:

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

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

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

CK_MILENAGE_SIGN_PARAMS milenage_params;
milenage_params.ulMilenageFlags = CKF_NC_MILENAGE_OP_OBJECT | CKF_NC_MILENAGE_OPC;
milenage_params.hSecondaryKey = opc_key;
memcpy(&(milenage_params.sqn), sqn, 6);
memcpy(&(milenage_params.amf), amf, 2);

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

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

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

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

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

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

17.13.2. CKM_NC_MILENAGE_RESYNC

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

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

17.13.3. CKM_NC_MILENAGE_AUTS (testing only)

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

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

17.14. CKM_NC_TUAK_TOPC

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

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

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

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

- `CKM_NC_TUAKSUBSCRIBER_KEY_GEN`
- `CKM_NC_TUAKTOP_KEY_GEN`

17.15. CKM_NC_TUAK, CKM_NC_TUAK_AUTS, CKM_NC_TUAK_RESYNC

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

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

The ulTuakFlags can consist of the following flags:

```
#define CKF_NC_TUAK_TOPC          0x00000001 /* secondary key is TOPC (not TOP) */
#define CKF_NC_TUAK_TOP_OBJECT  0x00000004 /* secondary key is supplied by object handle */
```

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

17.15.1. CKM_NC_TUAK

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

The following example shows how to use **CKM_NC_TUAK**:

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

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

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

CK_TUAK_SIGN_PARAMS tuak_params;
tuak_params.ulTuakFlags = CKF_NC_TUAK_TOP_OBJECT | CKF_NC_TUAK_TOPC;
tuak_params.hSecondaryKey = topc_key;
tuak_params.ulIterations = 1; /* 1 or 2 */
tuak_params.ulResLen = 32; /* 4, 8, 16 or 32 */
```

```

tuak_params.ulMacALen = 32;           // 8, 16 or 32
tuak_params.ulCkLen = 32;            // 16 or 32
tuak_params.ulIkLen = 32;            // 16 or 32
memcpy(&(tuak_params.sqn), sqn, 6);
memcpy(&(tuak_params.amf), amf, 2);

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

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

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

```

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

17.15.2. CKM_NC_TUAK_RESYNC

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

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

17.15.3. CKM_NC_TUAK_AUTS (testing only)

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

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

17.16. CKA_NC_KSD_IVS and CKA_NC_KSD_COUNTERS

CKA_NC_KSD_IVS can be used with **C_Wrap()** to provide one or more encrypted counter val-

ues to use with the `Mech_KSDEncryptAES` and `Mech_KSDEncrypt` nCore mechanisms when using the `CKM_AES_CBS` and `CKM_WRAP_RSA_CRT_COMPONENTS` PKCS#11 mechanisms.



Neither of these attribute values are considered sensitive information.

17.16.1. CKA_NC_KSD_COUNTERS selection

`CKA_NC_KSD_IVS` and `CKA_NC_KSD_COUNTERS` work as a pair of attributes. Each attribute value is a byte array that is a series of 16 byte values.

When using `C_Wrap`, if the supplied mechanism parameter value matches exactly the Nth value in the key's `CKA_NC_KSD_IVS` attribute, then the wrapping operation will use the Nth value from `CKA_NC_KSD_COUNTERS` as the encrypted counter.

17.16.2. Setting CKA_NC_KSD_IVS and CKA_NC_KSD_COUNTERS

When a new `CKK_AES` key is generated by `C_GenerateKey()` with `CKA_WRAP` set to `CK_TRUE` in the new key template, you may also include `CKA_NC_KSD_IVS`. With this attribute present in the key template, the PKCS#11 library will decrypt each of these 16-byte values using the newly created key to create the `CKA_NC_KSD_COUNTERS` attribute. You may specify one or more 16-byte encrypted counter strings as the template attribute value for `CKA_NC_KSD_IVS` when calling `C_GenerateKey()`, for example:

```
#include "pkcs11extra.h"

CK_OBJECT_CLASS class_secret = CKO_SECRET_KEY;
CK_KEY_TYPE key_aes = CKK_AES;
CK_BBOOL b_true = CK_TRUE;

CK_BYTE iv_list[] = {
    0x42, 0x00, 0x10, 0x0d, 0xcd, 0x0a, 0xf1, 0x00,
    0x20, 0x39, 0xaa, 0x0f, 0xc0, 0xd1, 0x02, 0x17
};

CK_ATTRIBUTE new_key[] = {
    { CKA_CLASS, &class_secret, sizeof(class_secret) },
    { CKA_KEY_TYPE, &key_aes, sizeof(key_aes) },
    { CKA_WRAP, &b_true, sizeof(b_true) },
    { CKA_KSD_IVS, iv_list, 16 },
};
```

The `new_key[]` template above will generate a key usable for wrapping using `DeriveMech_KSDEncryptAES` and `DeriveMech_KSDEncrypt` in nCore and store the IV counter values as `CKA_NC_KSD_COUNTERS`.



You may also set values for `CKA_NC_KSD_IVS` and `CKA_NC_KSD_COUNTERS` attributes on existing keys if these have been computed externally.

When doing so, both attributes must be the same length and a multiple of 16 bytes.

17.16.3. Using C_Wrap CKK_AES and CKA_NC_KSD_IVS

Supplying a mechanism parameter to `C_Wrap()` when using `CKM_AES_CBC`. If the `CKK_AES` key has both the `CKA_NC_KSD_IVS` and `CKA_NC_KSD_COUNTERS` attributes. The mechanism parameter value is used to select the counter from `CKA_NC_KSD_COUNTERS` as described in [CKA_NC_KSD_COUNTERS selection](#) and is used as the counter value for the nCore `DeriveMech_KSDEncryptAES` wrapping mechanism. For example:

```
CK_BYTE iv_list[] = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
CK_MECHANISM wrapmech = { CKM_AES_CBC, iv_list, 16 };
```

The ciphertext returned will be the result of wrapping the key using the `DeriveMech_KSDEncryptAES` nCore mechanism. The IV and MAC values produced by nCore are removed from the result.

17.16.4. Using C_Wrap CKK_RSA and CKA_NC_KSD_IVS

`C_Wrap()` can wrap RSA private keys with the `CKM_WRAP_RSA_CRT_COMPONENTS` mechanism using a `CKK_AES` wrapping key that has the `CKA_NC_KSD_COUNTERS` attribute set.

If a mechanism parameter is supplied, the `DeriveMech_KSDEncrypt` nCore mechanism is used and the counter value is selected as described in [CKA_NC_KSD_COUNTERS selection](#).

```
CK_BYTE iv_list[] = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
CK_MECHANISM wrapmech = { CKM_WRAP_RSA_CRT_COMPONENTS, iv_list, 16 };
```

The resulting ciphertext will be the result of `DeriveMech_KSDEncrypt` in nCore, with the IV and MAC values removed.

The returned encrypted RSA components are of equal length and in the following order:

- ENC(p)
- ENC(q)
- ENC(dmp1)

- ENC(dmqr1)
- ENC(iqmp)

18. KISAAlgorithm mechanisms

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

18.1. KCDSA keys

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

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

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

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

- `CKM_SHA_1`
- `CKM_HA160`
- `CKM_R1PEMD160`

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.

18.2. Pre-hashing

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

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

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

the message added after this.

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

18.3. CKM_KCDSA_SHA1, CKM_KCDSA_HA160, 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.

18.4. CKM_KCDSA_KEY_PAIR_GEN

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

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

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

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

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

18.5. CKM_KCDSA_PARAMETER_GEN



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

Use this mechanism to create a **CKO_DOMAIN_PARAMETERS** object. This is referred to as a **KCD-SAComm** 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.

18.6. CKM_HAS160

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

18.7. SEED secret keys

18.7.1. CKM_SEED_KEY_GEN

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

18.7.2. CKM_SEED_ECB, CKM_SEED_CBC, CKM_SEED_CBC_PAD

These mechanisms are the standard mechanisms to be used when encrypting and decrypt-

ing 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.

18.7.3. CKM_SEED_MAC, CKM_SEED_MAC_GENERAL

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

19. Attributes

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

19.1. CKA_SENSITIVE

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

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

See also `CKA_UNWRAP_TEMPLATE`.

19.2. CKA_PRIVATE

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

You must set `CKA_PRIVATE` to:

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

19.3. CKA_EXTRACTABLE

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

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

19.4. CKA_ENCRYPT, CKA_DECRYPT, CKA_SIGN, CKA_VERIFY

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

19.5. CKA_WRAP, CKA_UNWRAP

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

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

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

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

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

19.6. CKA_WRAP_TEMPLATE, CKA_UNWRAP_TEMPLATE

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

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

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

Nested occurrences of **CKA_WRAP_TEMPLATE** or **CKA_UNWRAP_TEMPLATE** are not supported.

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

CKA_MODIFIABLE (TRUE)

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

CKA_MODIFIABLE (FALSE)

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

CKA_SENSITIVE (TRUE)

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



For security reasons, Entrust recommends that you include **CKA_SENSITIVE=TRUE** in the template. This is because the restrictions imposed by **CKA_UNWRAP_TEMPLATE** are enforced at the module level. Keys with **CKA_SENSITIVE=FALSE** have low security, especially if **CKA_EXTRACTABLE=TRUE**.

CKA_SENSITIVE (FALSE)

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

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

19.7. CKA_SIGN_RECOVER

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

19.8. CKA_VERIFY_RECOVER

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

19.9. CKA_DERIVE

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

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

19.10. CKA_ALLOWED_MECHANISMS

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

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

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

19.10.1. CKM_CONCATENATE_BASE_AND_KEY

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

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

19.10.2. CKM_RSA_AES_KEY_WRAP

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

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

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

- `CKA_TOKEN` with a value of `CK_TRUE`
- `CKA_UNWRAP_TEMPLATE`

The `C_CopyObject` operation returns `CKR_ACTION_PROHIBITED` for such keys.

19.11. CKA_MODIFIABLE

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

See also `CKA_UNWRAP_TEMPLATE`.

19.12. CKA_TOKEN

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

19.13. CKA_START_DATE, CKA_END_DATE

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

19.14. CKA_TRUSTED and CKA_WRAP_WITH_TRUSTED

`CKA_TRUSTED` and `CKA_WRAP_WITH_TRUSTED` guard against a key being wrapped and removed from the HSM by an untrusted wrapping key. A key with a `CKA_WRAP_WITH_TRUSTED` attribute can only be wrapped by a wrapping key with a `CKA_TRUSTED` attribute. A trusted key can only

be given a `CKA_TRUSTED` attribute by the PKCS #11 Security officer.

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

The `CKA_TRUSTED` attribute stores on a wrapping key a certificate signed by the PKCS #11 Security Officer. This certificate can then be used to authenticate a wrapping operation.

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

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



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

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

19.15. CKA_COPYABLE and CKA_DESTROYABLE

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

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

- `C_GenerateKey` and `C_GenerateKeyPair`
- `C_CreateObject`
- `C_SetAttributeValue`
- `C_CopyObject`

Existing and new objects have both attributes set to true by default. When changing an

attribute, `CKA_COPYABLE` cannot be changed from false to true.

19.16. RSA key values

`CKA_PRIVATE_EXPONENT` is not used when importing an RSA private key using `C_CreateObject`. However, it must be in the template, since the PKCS #11 standard requires it. All the other values are required.

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

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

19.17. DSA key values

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

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

19.18. Vendor specific error codes

Security World Software defines the following vendor specific error codes:

`CKR_FIPS_TOKEN_NOT_PRESENT`

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

`CKR_FIPS_MECHANISM_INVALID`

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

`CKR_FIPS_FUNCTION_NOT_SUPPORTED`

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

20. Utilities

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

20.1. ckdes3gen

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

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

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

20.2. ckinfo

```
ckinfo [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`.

20.3. cklist

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

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

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

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

20.4. ckmechinfo

```
ckmechinfo
```

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

20.5. ckmldsa

```
ckmldsa [-p PIN] [-s SLOT] [-c CONTEXT-STRING] [ template options ] [ hedge variant selection ]
```

The `ckmldsa` utility is an example showing MLDSA key pair generation and use. This is intended as a programmer's example only and not for general use.

By default, the utility prompts for a passphrase. You can supply a passphrase on the command line with the `-p` (or `--pin-for-testing`) option. If no passphrase has been set, you can specify: `-p ""` (or `--pin-for-testing ""`). The passphrase is displayed in the clear on the command line, so this option is appropriate only for testing.

20.6. ckrsagen

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

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

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

20.7. cksotool

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

The `cksotool` utility can be used to create and manage the PKCS #11 Security Officer (SO). The SO consists of a token and an RSA key, and is necessary to be able to perform any oper

ations that require a Security Officer as defined by the PKCS #11 specification. The utility can be used to view the current state of the SO using the `-i` or `--info` option, which provides details of the existence and validity of the underlying token and key.

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

21. Functions

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



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

21.1. Choosing functions

Some PKCS #11 applications enable you to choose which functions you want to perform on the PKCS #11 token and which functions you want to perform in your application.

The following paragraphs in this section describe the functions that an nShield HSM can provide.

21.1.1. Generating random numbers and keys

The nShield HSM includes a hardware random number generator. A hardware random number generator provides greater security than the pseudo-random number generators provided by host computers. Therefore, always use the nShield HSM to generate random numbers and keys.

21.1.2. Digital signatures

The nShield PKCS #11 library can use the nShield HSM to sign and verify messages using the following algorithms:

- DSA
- RSA
- DES3_MAC
- AES
- ECDSA (if the appropriate feature is enabled)
- MLDSA (if the appropriate feature is enabled and a supported firmware version is used)

An nShield hardware security module is specifically optimized for public key algorithms, and therefore it will provide significant acceleration for DSA, RSA and ECDSA signature generation and verification. You should always choose to perform asymmetric signature generation and verification with an nShield HSM.

21.1.3. Asymmetric encryption

The nShield PKCS #11 library can use an nShield HSM to perform asymmetric encryption and decryption with the RSA algorithm.

The nShield HSM is specifically optimized for asymmetric algorithms, so you should always choose to perform asymmetric operations with the nShield HSM.

21.1.4. Symmetric encryption

The nShield PKCS #11 library can use the nShield HSM to perform symmetric encryption with the following algorithms:

- DES
- Triple DES
- AES

Because of limitations on throughput, these operations can be slower on the nShield HSM than on the host computer. However, although the nShield HSM may be slower than the host under a light load, you may find that under a heavy load the advantage gained from off-loading the symmetric cryptography (which frees the host CPU for other tasks) means that you achieve better overall performance.

21.1.5. Message digest

The nShield PKCS #11 library can perform message digest operations with MD5, SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512 algorithms. However, for reasons of throughput, the library performs these operations on the host computer.

21.1.6. Mechanisms

The mechanisms currently supported by the nShield PKCS #11 library, including some vendor-supplied mechanisms, are listed in [Mechanisms](#).

21.1.7. Key wrapping

The nShield PKCS #11 library can use an nShield HSM to wrap (encrypt) a private or secret key, or to unwrap (decrypt) a wrapped key.

21.1.8. Key agreement

To exchange keys using key encapsulation the nShield PKCS #11 library supports `CKM_ML_KEM` for use with `C_Encapsulate()` and `C_Decapsulate()`.

22. General purpose functions

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

22.1. C_Finalize

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_Finalize</code>	Yes	Without modifications	2.40, 3.2

22.1.1. Notes

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

22.2. C_GetInfo

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_GetInfo</code>	Yes	Without modifications	2.40, 3.2

22.3. C_GetFunctionList

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_GetFunctionList</code>	Yes	Without modifications	2.40

22.4. C_GetInterface

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_GetInterface</code>	Yes	Without modifications	3.2

22.5. C_GetInterfaceList

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_GetInterfaceList</code>	Yes	Without modifications	3.2

22.6. C_Initialize

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_Initialize</code>	Yes	Without modifications	2.40, 3.2

22.6.1. Notes

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

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

23. Slot and token management functions

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

23.1. C_GetSlotInfo

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_GetSlotInfo</code>	Yes	Without modifications	2.40, 3.2

23.2. C_GetTokenInfo

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_GetTokenInfo</code>	Yes	Without modifications	2.40, 3.2

23.3. C_GetMechanismList

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_GetMechanismList</code>	Yes	Without modifications	2.40, 3.2

23.4. C_GetMechanismInfo

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_GetMechanismInfo</code>	Yes	Without modifications	2.40, 3.2

23.5. C_GetSlotList

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_GetSlotList</code>	Yes	Without modifications	2.40, 3.2

23.5.1. Notes

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

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

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

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

23.6. C_InitToken

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_InitToken	Yes	Without modifications	2.40, 3.2

23.6.1. Notes

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

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

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

23.7. C_InitPIN

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_InitPin	Yes	Without modifications	2.40, 3.2

23.7.1. Notes

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

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

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

23.8. C_SetPIN

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_SetPin</code>	Yes	Without modifications	2.40, 3.2

23.8.1. Notes

The card passphrase may be any value.

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

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

24. Standard session management functions

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

24.1. C_OpenSession

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_OpenSession</code>	tbc	Without modifications	2.40, 3.2

24.2. C_CloseSession

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_CloseSession</code>	Yes	Without modifications	2.40, 3.2

24.3. C_CloseAllSessions

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_CloseAllSessions</code>	Yes	Without modifications	2.40, 3.2

24.4. C_GetOperationState

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_GetOperationState</code>	Yes	Without modifications	2.40, 3.2

24.5. C_GetSessionInfo

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_GetSessionInfo</code>	Yes	Without modifications	2.40, 3.2

24.6. C_SetOperationState

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_SetOperationState</code>	Yes	Without modifications	2.40, 3.2

24.7. C_Login

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_Login</code>	Yes	Without modifications	2.40, 3.2

24.8. C_Logout

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_Logout</code>	Yes	Without modifications	2.40, 3.2

25. nShield session management functions

The following are nShield-specific calls for *K/N* card set support and are extensions of the PKCS #11 specification.

25.1. C_LoginBegin

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_LoginBegin</code>	Yes	N/A	N/A

25.2. C_LoginNext

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_LoginNext</code>	Yes	N/A	N/A

25.3. C_LoginEnd

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_LoginEnd</code>	Yes	N/A	N/A

26. Object management functions

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

26.1. C_CreateObject

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_CreateObject</code>	Yes	Without modifications	2.40, 3.2

26.1.1. CKK_NC_MILENAGERC

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

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

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

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

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

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

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

```

}

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

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

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

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

```

26.2. C_CopyObject

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_CopyObject	Yes	Without modifications	2.40, 3.2

26.3. C_DestroyObject

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_DestroyObject	Yes	Without modifications	2.40, 3.2

26.4. C_GetObjectSize

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_GetObjectSize	Yes	Without modifications	2.40, 3.2

26.5. C_GetAttributeValue

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_GetAttributeValue</code>	Yes	Without modifications	2.40, 3.2

26.6. C_SetAttributeValue

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_SetAttributeValue</code>	Yes	Without modifications	2.40, 3.2

26.7. C_FindObjectsInit

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_FindObjectsInit</code>	Yes	Without modifications	2.40, 3.2

26.8. C_FindObjects

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_FindObjects</code>	Yes	Without modifications	2.40, 3.2

26.9. C_FindObjectsFinal

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_FindObjectsFinal</code>	Yes	Without modifications	2.40, 3.2

27. Encryption functions

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

27.1. C_EncryptInit

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_EncryptInit</code>	Yes	Without modifications	2.40, 3.2

27.2. C_Encrypt

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_Encrypt</code>	Yes	Without modifications	2.40, 3.2

27.3. C_EncryptUpdate

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_EncryptUpdate</code>	Yes	Without modifications	2.40, 3.2

27.4. C_EncryptFinal

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_EncryptFinal</code>	Yes	Without modifications	2.40, 3.2

28. Decryption functions

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

28.1. C_DecryptInit

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_DecryptInit</code>	yes	Without modifications	2.40, 3.2

28.2. C_Decrypt

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_Decrypt</code>	yes	Without modifications	2.40, 3.2

28.3. C_DecryptUpdate

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_DecryptUpdate</code>	yes	Without modifications	2.40, 3.2

28.4. C_DecryptFinal

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_DecryptFinal</code>	yes	Without modifications	2.40, 3.2

29. Message digesting functions

The following functions are performed on the host computer:

29.1. C_DigestInit

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_DigestInit</code>	Yes	Without modifications	2.40, 3.2

29.2. C_Digest

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_Digest</code>	Yes	Without modifications	2.40, 3.2

29.3. C_DigestUpdate

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_DigestUpdate</code>	Yes	Without modifications	2.40, 3.2

29.4. C_DigestFinal

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_DigestFinal</code>	Yes	Without modifications	2.40, 3.2

30. Signing and MACing functions

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

30.1. C_SignInit

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_SignInit</code>	Yes	Without modifications	2.40, 3.2

30.2. C_Sign

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_Sign</code>	Yes	Without modifications	2.40, 3.2

30.3. C_SignRecoverInit

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_SignRecoverInit</code>	Yes	Without modifications	2.40, 3.2

30.4. C_SignRecover

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_SignRecover</code>	Yes	Without modifications	2.40, 3.2

30.5. C_SignUpdate



Not all mechanisms support `C_SignUpdate` or `C_VerifyUpdate`. If such a mechanism is used, these functions will return `CKR_OPERATION_NOT_INITIALIZED`.

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_SignUpdate</code>	Yes	Without modifications	2.40, 3.2

30.6. C_SignFinal

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_SignFinal</code>	Yes	Without modifications	2.40, 3.2

31. Functions for verifying signatures and MACs

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

31.1. C_VerifyInit

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_VerifyInit</code>	Yes	Without modifications	2.40, 3.2

31.2. C_Verify

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_Verify</code>	Yes	Without modifications	2.40, 3.2

31.3. C_VerifyRecover

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_VerifyRecover</code>	Yes	Without modifications	2.40, 3.2

31.4. C_VerifyRecoverInit

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_VerifyRecoverInit</code>	Yes	Without modifications	2.40, 3.2

31.5. C_VerifyUpdate



Not all mechanisms support `C_SignUpdate` or `C_VerifyUpdate`. If such a mechanism is used, these functions will return `CKR_OPERATION_NOT_INITIALIZED`.

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_VerifyUpdate</code>	Yes	Without modifications	2.40, 3.2

31.6. C_VerifyFinal

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_VerifyFinal</code>	Yes	Without modifications	2.40, 3.2

32. Dual-purpose cryptographic functions

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

32.1. C_DigestEncryptUpdate

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_DigestEncryptUpdate	Yes	Without modifications	2.40, 3.2

32.2. C_DecryptDigestUpdate

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_DecryptDigestUpdate	Yes	Without modifications	2.40, 3.2

32.3. C_SignEncryptUpdate

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_SignEncryptUpdate	Yes	Without modifications	2.40, 3.2

32.3.1. Notes

This function is supported for:

- CKM_SHA1_RSA_PKCS
- CKM_MD5_RSA_PKCS

32.4. C_DecryptVerifyUpdate

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_DecryptVerifyUpdate	Yes	Without modifications	2.40, 3.2

32.4.1. Notes

This function is supported for:

- CKM_SHA1_RSA_PKCS
- CKM_MD5_RSA_PKCS

33. Key-management functions



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

In Security World v13.3.2 and later, you can set the `CKNFAST_LOADSHARING` environment variable to enable load sharing for the work allocation for key-management functions:

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

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

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

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

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

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

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

33.1. C_GenerateKey

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_GenerateKey</code>	Yes	Without modifications	2.40, 3.2

33.2. C_GenerateKeyPair

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_GenerateKeyPair</code>	Yes	Without modifications	2.40, 3.2

33.3. C_WrapKey

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_WrapKey</code>	Yes	Without modifications	2.40, 3.2

33.4. C_UnwrapKey

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_UnwrapKey</code>	Yes	Without modifications	2.40, 3.2

33.5. C_DeriveKey

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_DeriveKey</code>	Yes	Without modifications	3.2

33.6. C_Decapsulate

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_Decapsulate</code>	Yes	Without modifications	3.2

33.7. C_Encapsulate

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
C_Encapsulate	Yes	Without modifications	3.2

34. Random number functions

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

34.1. C_GenerateRandom

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_GenerateRandom</code>	Yes	Without modifications	2.40, 3.2

34.2. C_SeedRandom

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_SeedRandom</code>	Yes	Without modifications	2.40, 3.2

34.2.1. Notes

The `C_SeedRandom` function returns `CKR_RANDOM_SEED_NOT_SUPPORTED`.

35. Parallel function management functions

35.1. C_GetFunctionStatus

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_GetFunctionStatus</code>	Yes	Without modifications	2.40, 3.2

35.1.1. Notes

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

35.2. C_CancelFunction

Function	Supported in Security World	Performs as in PKCS #11 spec	PKCS #11 spec version
<code>C_CancelFunction</code>	Yes	Without modifications	2.40, 3.2

35.2.1. Notes

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

36. Callback functions

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