Application Notes

Security World Keys

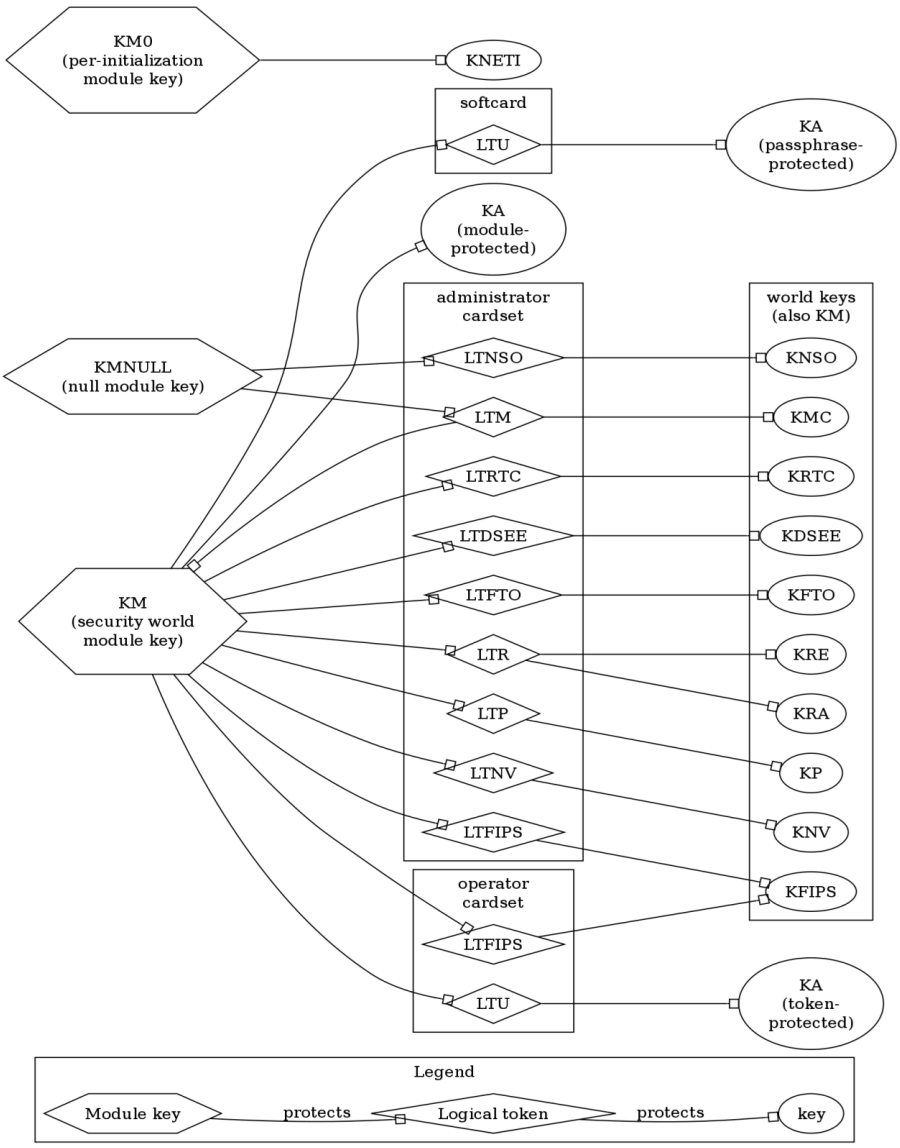
10 February 2026

Table of Contents

1. Introduction

This document discusses the cryptographic keys that make up a security world.

The diagram below summarizes the ways in which keys are protected. For more details see the following chapters.



2. Definitions

2.1. Definitions

A *key* here means any cryptographic key, anywhere, in the abstract.

An *application key* is a key that can be loaded into the module with an ACL and accessed via a key handle. This includes the security world administrator keys, most of which would not normally be used directly by customer applications.

Confusingly it also includes some objects that are not really keys at all, such as template keys for key derivation.

A *module key* is a symmetric key that is persistent within the module. Normally there is no corresponding key handle.

A *module private key* is an asymmetric key that is persistent within the module. Normally there is no corresponding key handle although it is usually possible to get a key handle to the public half of such keys.

A *security world administrator key* is an application key or module key used in securing or administering the security world.

A *logical token* is a key stored either in a cardset or softcard and used to protect application keys or administrator keys.

3. Module Keys

3.1. Module Keys

A module key:

- is a DES3 or AES key
- is not an application key
- does not have an ACL
- is persistent across module reset (but not reinitialization)
- is identifiable by the hash of the key material (usually; not true for null module keys)
- is given a name starting "KM"
- can be used to protect a logical tokens and keys

Some module keys are constructed entirely within the module, for example `KM0`. Others are loaded (normally only during initialization) via the `SetKM` command.

Module keys are erased by `Cmd_InitialiseUnit` and `Cmd_InitialiseUnitEx`. Moreover the persistent storage within the HSM used for module keys is encrypted under a key unique to the HSM and installed firmware, putting them beyond use after an upgrade.

3.1.1. KM0

`KM0` is:

- an AES key (formerly DES3)
- generated by `Cmd_InitialiseUnit` and `Cmd_InitialiseUnitEx` (destroying the old `KM0` upon reinitialization; so `KM0` blobs are not suitable for long-term key protection)

`KM0` appears in the slot 0 in the internal array of module keys (and therefore the first slot in the list of module key hashes returned by `Cmd_GetKMList`), hence the name.

If an application key is to be blobbed under any module key *other* than `KM0`, then `AllowNonKm0` must appear in its ACL.

`KM0` is an AES key even if the module is in a DES3 security world.

3.1.2. Null module keys

`KMNULL` is:

- for a given key type, constant
- well-known
  - for DES3, the key material is `0x010101···`
  - for AES, the key material ix `0x000000···`
- identifiable in ACLs by a 'special' hash
  - for DES3, the hash is `0x000000···`
  - for AES, the hash is `0x010000···`
- built into the module firmware
- used to 'protect' logical tokens that need to be read during module initialization (`LTNSO` and `LTM`), since no other confidentiality key is available at this point
- not capable of providing any confidentiality

If an application key is to be blobbed under a logical token protected by a null module key, `AllowNullKMToken` must appear in its ACL. Normally this would only be used for `KNSO` and KM, since no other module key is available when they are to be created or loaded.

3.1.3. KMWK

The underlying null module key values are also used as `KMWK`, the "well known module key". Which of them is used depends on the ciphersuite.

- In `DLf1024s160mDES3`, `DLf1024s160mRijndael` and `DLf3072s256mRijndael`, `KMWK` is the DES3 null module key (with nCore key hash of this key is `1d572201be533ebc89f30fdd8f3fac6ca3395bf0`)
- In other worlds `KMWK` is the AES module key (with nCore key hash `c2be99fe1c77f1b75d48e2fd2df8dffc0c969bcb`)

`KMWK` is installed as a module key, exempting it from the requirement for `AllowNullKMToken` in ACLs.

To load `KMWK` into the module a (constant!) blob of `KMWK` under itself is used.

3.1.4. Module Key API Calls

- `Cmd_SetKM` allows an application key to be stored as a module key. The application key must have the `UseAsKM` permission in its ACL. The sworld implementation uses this com mand during world creation and world loading to install KM as a module key.
- `Cmd_RemoveKM` removes a module key from persistent storage. `KM0` cannot be removed.
- `Cmd_GetKMList` retrieves a list of the hashes of all module keys.

- `Cmd_InitialiseUnit` and `Cmd_InitialiseUnitEx` erase all module keys (including `KM0`).

Example

```
>>> print conn.transact(nfpython.Command(['GetKMList', 0, 1]))
Reply.cmd= GetKMList
     .status= OK
     .flags= 0x0
     .reply.flags= KNSOset
     .hknso= 787720a1 e305a5af 6c8564af 3f6bfe55 122a733d
     .hkms[0]= df4d2a87 e0eca861 a1063bd1 dcaa9b1a 87bf83f6
     .hkms[1]= 1d572201 be533ebc 89f30fdd 8f3fac6c a3395bf0
     .hkms[2]= 8fd82ce3 2e58f8e4 b559dddb 62de8cf6 5e66ce58
```

In the output:

- `hkms[0]` is `KM0` (see KM0).
- `hkms[1]` is `KMWK` (see KM0). This ends up in this position due to the order things happen in security world creation. Applications shouldn't assume that this is always true.
- `hkms[2]` is KM in this example, see KM (Security World Module Key). Again the numbering is a result of the order in which security world creation is performed; applications shouldn't assume that it is always true.

Compare this with the output of `nfkminfo`:

```
$ nfkminfo
World

generation 2
state 0x1fa70000 Initialised Usable Recovery PINRecovery !ExistingClient RTC NVRAM
  FTO !AlwaysUseStrongPrimes SEEDebugForAll
n_modules 1
hknso 787720a1e305a5af6c8564af3f6bfe55122a733d
hkm 8fd82ce32e58f8e4b559dddb62de8cf65e66ce58* (type Rijndael)
hkmwk 1d572201be533ebc89f30fdd8f3fac6ca3395bf0*
hkre a7ae9935f04f4680c3046a02c7fa838dd3ffd3ed
...
```

4. Module Private Keys

4.1. Module Private Keys

These aren't module keys in the sense described previously (for instance, they are not enumerated by `Cmd_GetKMList`), but they are persistent keys private to the module.

4.1.1. KLF and KLF2 (Module Long-Term Signing Keys)

These are the module's long-term signing keys. They are generated in the factory and fixed for the lifetime of the module (i.e. they survive reinitialization).

`KLF` is a 1024-bit DSA keys and `KLF2` is a P-521 ECDSA key.

Current HSMs use `KLF2`, `KLF` is deprecated. nShield 5s, nShield 5c, and later HSMs use `KLF2`.

The `HasKLF` and `HasKLF2` bits in the `EnquiryDataFour` flags indicate whether these keys are present in the module.

A *warrant* is a certificate make in the factory which binds `KLF`(or `KLF2`) to a module's ESN. It can be used as part of a communication protocol to assure a peer that it is talking to a particular module.

`Cmd_GetKLF` can be used to get a handle to the public half of KLF. `Cmd_GetModuleState` and `Cmd_SignModuleState` can be used to retrieve the public half of `KLF` or `KLF2`.

KLF is obsolete and no longer supported from nShield 5.

4.1.2. KML (Module Per-Initialization Signing Key)

This is a module signing key. It is generated during module initialization (and therefore destroyed when the module is reinitialized). It is a 1024-bit or 3072-bit DSA key, depending on the security world ciphersuite.

The KML type depends on the Security World cipher suite. It can be one of the following:

- DSA key (1024-bit or 3072-bit)
- ECDSA key (on NIST P-256 or P-521)

In Security World v13.7 or later, the KML type can be selected at world creation and/or loading time using the `--kml-type` option with `new-world`.

Cmd_GetKML can be used to get a handle to its public half, and Cmd_GetModuleState and Cmd_SignModuleState can be used to retrieve the public half.

It is used to sign key generation certificates. It is in turn signed by KMC, in CertKMLaESN.

The hkml member of a NFKM_ModuleInfo structure contains the hash of KML, if the module is usable (i.e. in the right security world). The blobpubkml member of an NFKM_WorldInfo or NFKM_Key structure contains the public key blob for the KML of the module that the world or key was generated on.

Examples

This example shows the retrieval of the public half of KLF.

```
>>> print conn.transact(nfpython.Command(['GetModuleState',0,1,'attribs_present',['KLF']]))
Reply.cmd= GetModuleState
     .status= OK
     .flags= 0x0
     .reply.state.attribs[0].tag= KLF
                            .value.hklf= 761403dc de8195f1 e26e3a2d 7d666a76 30229e25
                                  .klfpub.type= DSAPublic
                                         .data.dlg.p= 0xe56e[...]
                                                  .q= 0xbac6[...]
                                                  .g= 0x1af9[...]
                                                  .y= 0x5032[...]
                                  .mech_i= DSA
```

This example shows the retrieval of the public half of KML:

```
>>> print conn.transact(nfpython.Command(['GetModuleState',0,1,'attribs_present',['KML']]))
Reply.cmd= GetModuleState
     .status= OK
     .flags= 0x0
     .reply.state.attribs[0].tag= KML
                            .value.hkml= ec0acf3e 74f93c87 17b64c40 8e6525f8 7960ff6b
                                  .kmlpub.type= DSAPublic
                                         .data.dlg.p= 0xcc2d[...]
                                                  .q= 0xfccc[...]
                                                  .g= 0xa870[...]
                                                  .y= 0xa7f1[...]
                                  .mech_i= DSAhSHA256
```

5. Security World Administrator Keys

5.1. Security World Administrator Keys

A security world administrator key:

- is an application key
- has one of several possible types, dependent on its role and the security world's cipher suite
- is blobbed under a logical token and stored in the `world` file

With the exception of `KFIPS` all these keys are blobbed under logical tokens found only on the administrator cardset, so this cardset must be exposed to use them. For the asymmetric keys their public halves are available from the `world` file.

5.1.1. KNSO (nCipher Security Officer's Key)

This is an *asymmetric integrity* key blobbed under `LTNSO`.

It used for:

- Signing world-binding certificates
- Signing delegation certificates
- Authorization trump operations groups in ACLs of other keys.

Its ACL has four groups:

- A main use group allowing `UseAsCertificate` (for authorizing other operations) and `MakeBlob`. This has a time limit, configurable at world creation time (for example, using `new-world --nso-timeout=…`)
- A certificate signing group. This allows "just enough" `Sign` operations for world creation, and nothing more.
- A certificate use group. This allows "just enough" `UseAsCertificate` operations for world creation, and nothing more. These operations are consumed during the blobbing of the other administrator keys, via their trump operations groups (which require `KNSO` authorization).
- A single-use group allowing `KNSO` itself to be blobbed.

The certificate use group and the single-use group are restricted by use limits, rather than logical token hashes, because the logical token hashes are not available at this point in world creation.

It is variously known as:

- The security officer's key
- The security world root key
- The administrator key
- The master key (in very old documentation)
- The nCipher Security Officer's key

In an `NFKM_WorldInfo` structure, `hknso` is the hash of `KNSO` and `nsotimeout` the time limit on the main use group.

5.1.2. KM (Security World Module Key)

This is a *symmetric security* key blobbed under `LTM`. During world initialization it is installed as a module key; in this role it has the following uses:

- Protection of `LTR`, `LTP`, `LTNV`, `LTRTC`, `LTDSEE`, `LTFTO` and `LTFIPS` on administrator cardsets.
- Protection of `LTU` and `LTFIPS` on operator cardsets.
- Blobbing of module-protected application keys.

It is sometimes referred to as `KMSW` (the security world module key).

Its ACL has two groups:

- A main use group, allowing its use as a module key.
- A trump operations group.

  Note that trump operations group are present for administrator keys even in non-recoverable worlds. See Secret Key ACLs for further discussion of trump operations groups.

In an `NFKM_WorldInfo` structure, `hkm` is the hash of `KM`.

5.1.3. KMC (Module Certification Key)

This is an *asymmetric integrity* key blobbed under `LTM`. It is used to sign world-binding certificates.

Its ACL has two groups:

- A main use group allowing `Sign`.
- A trump operations group.

In an `NFKM_WorldInfo` structure, `hkmc` is the hash of `KMC` and `blobpubkmc` the public key blob.

5.1.4. KRE (Recovery Encryption Key)

This is an *asymmetric security* key, only present in recoverable worlds. It is blobbed under `LTR`. It used to encrypt and decrypt recovery blobs for application keys.

Its ACL has two groups:

- A main use group allowing `UseAsBlobKey`.
- A trump operations group.

In an `NFKM_WorldInfo` structure, `hkre` is the hash `of` KRE and `blobpubkre` the public key blob.

5.1.5. KRA (Recovery Authorization Key)

This is an *asymmetric integrity* key, only present in recoverable worlds. It is blobbed under `LTR`. It is not used in the current implementation.

The intention is to use it to supply authorization for recovery operations, a function currently performed by `KNSO`. There is no current plan to bring it into use.

Its ACL has two groups:

- A main use group allowing `UseAsCertificate`.
- A trump operations group.

In an `NFKM_WorldInfo` structure, `hkra` is the hash of `KRA`.

5.1.6. KP (Passphrase Recovery Key)

This is an *asymmetric security* key, only present in worlds supporting PIN (passphrase) recovery. It is blobbed under `LTP`. It is used to encrypt and decrypt operator card and softcard passphrases.

In a few places this key is referred to as `KRP` rather than `KP`. This may be considered an error.

Its ACL has two groups:

- A main use group allowing `Decrypt`.
- A trump operations group.

In an `NFKM_WorldInfo` structure, `hkp` is the hash of `KP` and `blobpubkp` the public key blob.

5.1.7. KFIPS (FIPS Authorization Key)

This is an *asymmetric integrity* key, only present in strict-FIPS world. It is blobbed under `LTFIPS`, which can be assembled from other administrator cards and operator cards. Its primary purpose is to authorize restricted operations in strict-FIPS worlds via delegation from `KNSO`, though it is also used to sign world-binding certificates.

Its ACL has two groups:

- A main use group allowing `UseAsCertificate` and `Sign`, plus `MakeBlob` under any 1/64 logical token with `KM`.
- A trump operations group.

In an `NFKM_WorldInfo` structure, `hkfips` is the hash of `KFIPS` and `blobpubkfips` the public key blob.

5.1.8. Delegation Keys

The delegation keys are:

- `KNV`, blobbed under `LTNV` and authorizing NVRAM operations
- `KRTC`, blobbed under `LTRTC` and authorizing RTC setting
- `KDSEE`, blobbed under `LTDSEE` and authorizing SEE debugging
- `KFTO`, blobbed under `LTFTO` and authorizing foreign token open

They are all *asymmetric integrity* keys and authorize their respective operations via delegation from `KNSO`. They are only present for worlds supporting the relevant operations.

Their ACLs have two groups:

- A main use group allowing `Sign` and `UseAsCertificate`.
- A trump operations group.

In an `NFKM_WorldInfo` structure, `hknv`, `hkrtc`, `hkdsee` and `hkfto` are the hashes of the delegation keys, if present. `blobpubknv`, `blobpubkrtc`, `blobpubdsee` and `blobpubkfto` are the public key blobs.

5.1.9. Key Types

The key types and mechanisms used in each role are as follows:

DLf1024s160mDES3

Symmetric integrity	DES3
Symmetric security	DES3
Working blob	DES3wSHA1
Temporary blob	DES3wSHA1
Asymmetric integrity	1024-bit DSA
Signature	DSA
Asymmetric security	1024-bit RSA
Recovery blob	RSApPKCS1

DLf1024s160mRijndael

Symmetric integrity	256-bit AES
Symmetric security	256-bit AES
Working blob	BlobCryptv2kHasheRijndaelCBC0hSHA1mSHA1HMAC
Temporary blob	BlobCryptv2kHasheRijndaelCBC0hSHA512mSHA512HMAC
Asymmetric integrity	1024-bit DSA
Signature	DSA
Asymmetric security	1024-bit RSA
Recovery blob	BlobCryptv2kRSAeRijndaelCBC0hSHA512mSHA512HMAC

DLf3072s256mRijndael

Symmetric integrity	256-bit AES
Symmetric security	256-bit AES
Working blob	BlobCryptv2kHasheRijndaelCBC0hSHA1mSHA1HMAC
Asymmetric integrity	3072-bit DSA
Signature	DSAhSHA256
Asymmetric security	3072-bit RSA
Recovery blob	BlobCryptv2kRSAeRijndaelCBC0hSHA512mSHA512HMAC

DLf3072s256mAEScSP800131Ar1

Symmetric integrity	256-bit AES
Symmetric security	256-bit AES
Working blob	BlobCryptv3kNoneeAESCBC0dCTRCMACmSHA256HMAC
Temporary blob	BlobCryptv3kNoneeAESCBC0dCTRCMACmSHA512HMAC
Asymmetric integrity	3072-bit DSA
Signature	DSAhSHA256
Asymmetric security	3072-bit RSA
Recovery blob	BlobCryptv3kRSAOAEPeAESCBC0dCTRCMACmSHA512HMAC

ECp521mAES

Symmetric integrity	256-bit AES
Symmetric security	256-bit AES
Working blob	BlobCryptv3kNoneeAESCBC0dCTRCMACmSHA256HMAC
Temporary blob	BlobCryptv3kNoneeAESCBC0dCTRCMACmSHA512HMAC
Asymmetric integrity	ECDSA P-521
Signature	ECDSAhSHA512
Asymmetric security	ECDH P-521
Recovery blob	BlobCryptv4kECDHeAESmCTRhSHA256mSHA256HMAC

5.1.10. Administrator Key API Calls

The C API is declared in `nfkm.h`:

- `KFIPS` support is provided by `NFKM_fips140auth()`, `NFKM_freefips140auth()` and `NFKM_newkey_makeauth()`. Other `NFKM_…` functions take a `FIPS` handle where necessary. In general applications would be expected to use this API rather than supplying `KFIPS` and the delegation certificate manually, though there is no reason they can't.
- The `NFKM_initworld_…` functions are responsible for creating these keys and blobbing them under their respective logical tokens, and `NFKM_loadworld_…` for installing an existing `KM` when indoctrinating a module into an existing security world.
- `NFKM_replaceocs_…` loads `KRE` so it can load recovery blobs and `KNSO` so it can make new key blobs.
- `NFKM_replaceacs_…` loads keys for subsequent reblobbing under a new set of logical

tokens.

- `NFKM_recoverpin_…` loads `KP` so it can decrypt PIN recovery blobs.
- `NFKM_loadadminkeys_…` loads arbitrary administrator keys.
- `NFKM_softcard_recoverpp()` expects to be supplied a handle to `KP` so it can decrypt PIN recovery blobs.

In addition these operations are available as card-loading library logics, which is how most command line tools and the Connect's front panel UI access them.

In Python:

- `nfkm.getfipsauth()` and `nfkm.makefipscert()` provide `KFIPS` support.
- There is no direct analogue to `NFKM_initworld_…` et al.
- Card-loading library logics are available via the `rqcard` module.

Examples

`nfkminfo` can be used to reveal which administrator keys are present in a security world and what their hashes are:

```
$ nfkminfo
World
generation 2
state 0x1fa70000 Initialised Usable Recovery PINRecovery !ExistingClient RTC NVRAM FTO !AlwaysUseStrongPrimes
SEEDebugForAll
n_modules 1
hknso 787720a1e305a5af6c8564af3f6bfe55122a733d
hkm 8fd82ce32e58f8e4b559dddb62de8cf65e66ce58 (type Rijndael)
hkmwk 1d572201be533ebc89f30fdd8f3fac6ca3395bf0
hkre a7ae9935f04f4680c3046a02c7fa838dd3ffd3ed
hkra 9d1e3c81989fb4db3df7a35df8d021721ff24064
hkmc 1527edda4cd26c7a8354e070bc026ec1ca01d68a
hkp 96ee8fc6732eb43c069dd555490dc2ff6a5b8e7c
hkrtc 1682e9ae78914ba594ae00da4115af51a596157b
hknv edde09e010bf98cca529416e8209565309ca7a59
hkdsee 24b4c4195f80b7d692877effe7bbd8423d3b11e9
hkfto dd30b3831985ebec7469f357e923d56aef514d7d
hkmnull 0100000000000000000000000000000000000000
ex.client none
...
```

To extra the current module configuration instead, use `Cmd_GetKMList`; see Module key API calls for an example.

From Python it is possible to retrieve the key hashes of individual administrator keys:

```
>>> nfkm.getinfo(conn)["hkre"]
"""a7ae9935 f04f4680 c3046a02 c7fa838d d3ffd3ed"""
```

6. Logical Tokens

6.1. Logical Tokens

A logical token:

- is a key
- is not an application key
- does not have an ACL (although individual shares do have ACLs)
- may be DES3 or AES
- can be split into shares and written to smartcards
- can be used to protect keys
- is protected (on smartcards) by a module key
- is denoted `LTsomething`

Logical tokens are written to smartcards as follows:

- The logical token and associated metadata are encrypted under the chosen module key.
- The ciphertext is split into shares.
- For each share a unique share key is derived from the module key, the passphrase (if any) and some additional information.
- Each share is encrypted under its share key.
- Each share is written to a smartcard.

6.1.1. Administrator Cardset Logical Tokens

An administrator cardset holds the following logical tokens:

Token	Module Key	Purpose	Threshold
`LTNSO`	`KMNULL`	Protects `KNSO`	`acs_k`
`LTM`	`KMNULL`	Protects `KM`	`t_m`
`LTR`	`KM`	Protects `KRE`	`t_r`
`LTP`	`KM`	Protects `KP`	`t_p`
`LTNV`	`KM`	Protects `KNV`	`t_nv`
`LTRTC`	`KM`	Protects `KRTC`	`t_rtc`

Token	Module Key	Purpose	Threshold
LTDSEE	KM	Protects KDSEE	t_dsee
LTFTO	KM	Protects KFTO	t_fto
LTFIPS	KM	Protects KFIPS	1

LTNSO and LTM always require a full quorum of administrator cards to reassemble.

LTFIPS only ever requires a single administrator card to reassemble. It is only present in strict-FIPS worlds.

The other logical tokens by default require a full quorum but this can be configured at world creation time.

In an NFKM_WorldInfo structure, the quorums of each logical token are given by the members shown in the table above.

6.1.2. Operator Cardset Logical Tokens

An operator cardset holds the following logical tokens:

Token	Module Key	Purpose
LTU	KM	Protects application keys
LTFIPS	KM	Protects KFIPS

LTU requires a full quorum of the operator cardset to reassemble. It is used to protect cardset-protected keys.

As above, LTFIPS only ever requires a single operator card to reassemble. It is only present in strict-FIPS worlds.

6.1.3. Softcard Logical Tokens

A softcard is a software emulation of a single smartcard. The single share of the token is stored in a file. Its contents can be supplied to the module with Cmd_InsertSoftToken and retrieved with Cmd_RemoveSoftToken.

This is used to implement passphrase-protected keys by storing a 1/1 logical token (called LTU) to it, protected by KM. Application keys are then protected by the softcard's LTU.

6.1.4. Logical Token API Calls

- `Cmd_FormatToken` prepares a smartcard for storage of logical token shares.
- `Cmd_GenerateLogicalToken` constructs a new logical token. It would normally be followed by calls to `Cmd_WriteShare` to store token shares to smartcards.
- `Cmd_EraseShare` erases a logical token share from a smartcard.
- `Cmd_LoadLogicalToken` starts reassembly of a logical token. It would normally be followed by calls to `Cmd_ReadShare` to read successive token shares.
- `Cmd_GetSlotList` identifies the token shares on smartcards.
- `Cmd_GetShareACL` retrieves the ACL of an individual token share.
- `Cmd_GetLogicalTokenInfo` and `Cmd_GetLogicalTokenInfoEx` return information about a logical token (including partially assembled ones).
- `Cmd_ChangeSharePIN` changes the PIN (passphrase) on a token share.

7. Application Keys

7.1. Application Keys

Although a user (customer) application has a great deal of flexibility about how it manages its keys, the security world API provides a standard approach.

Within a security world application keys are often denoted KA. Within a given application they should be given a more descriptive name!

7.1.1. Secret Key ACLs

If the standard tools are used, then private and symmetric keys have (some of) the groups described in the following sections:

7.1.1.1. Main Use Group

Normally all keys have a *main use group* allowing the operations relevant to their purpose. For application key it would usually allow the following:

- Whichever of Sign, Verify, Encrypt, Decrypt, SignModuleCert, UseAsCertificate, UseAsLoaderKey, and UseAsBlobKey are appropriate to the key type and application.

  UseAsLoaderKey is used for SEE confidentiality keys; it allows decryption but only when loading an SEE machine or userdata.

- GetAppData.
- Safe operations such as DuplicateHandle, ReduceACL and GetACL.

Example

```
groups[0].flags= none 0x00000000
         .n_limits= 0
         .n_actions= 1
         .actions[ 0].type= OpPermissions
         .details.oppermissions.perms= DuplicateHandle UseAsCertificate GetAppData ReduceACL Decrypt UseAsBlobKey
Sign
                                       GetACL SignModuleCert 0x0000b52b
         .certifier absent
         .certmech absent
         .moduleserial absent
```

Encrypt and Verify are missing because this is the ACL for the private half of an asymmetric key. UseAsBlobKey is present because this bit controls both creation and loading of blobs.

The effect is that authorized users of the key, for example, as identified by ability to present an operator card set quorum, can use the key to perform cryptographic operations (sign, decrypt, and so on) but not change its protection.

In some cases this group is split into two. See Use Limits And SEE Application Keys.

7.1.1.2. Working Blob Group

Most keys have a *working blob group* to allow them to be persisted. For application keys this will allow `MakeBlob` under `KM` (for module keys) or the logical token that will protect the key. This is a single-use group.

Example

```
.groups[1].flags= none 0x00000000
          .n_limits= 1
          .limits[0].type= Global
                    .details.global.id= 20 bytes 423141fd d65aa9d7 5982be14 2bd1c572 a7475f94
                                .max= 0x00000001 1
          .n_actions= 1
          .actions[0].type= MakeBlob
                    .details.makeblob.flags= AllowKmOnly AllowNonKm0 kmhash_present 0x00000007
                                  .kmhash= 20 bytes 8fd82ce3 2e58f8e4 b559dddb 62de8cf6 5e66ce58
                                  .kthash absent
                                  .ktparams absent
                                  .blobfile absent
          .certifier absent
          .certmech absent
          .moduleserial absent
```

`kmhash (0x8fd82ce3···)` constrains the module key that will protect this key.

In contrast the ACL for a token-protected key identifies the logical token under which it may be blobbed. Here only the action is shown, the rest of the group has the same structure as above:

```
.actions[0].type= MakeBlob
           .details.flags= AllowNonKm0|kmhash_present|kthash_present|ktparams_present
                   .kmhash= 8fd82ce3 2e58f8e4 b559dddb 62de8cf6 5e66ce58
                   .kthash= 608631c1 699af879 5ff36393 f597155a 0a53af05
                   .ktparams.flags= AllTokensRemovable|AllButOneRemovable
                            .sharesneeded= 1
                            .sharestotal= 1
                            .timelimit= 0
```

Here `kmhash (0x8fd8···)` constrains the module key associated with the logical token that will protect this key. `kthash (0x6086···)` constrains the hash of the logical token itself.

The values below `ktparams` represent the least secure token parameters acceptable, for example, the minimum sharesneeded and the maximum `sharestotal`.

This group is only used during key generation. Permission groups with a global use limit are dropped when the key is reloaded (via `Cmd_LoadBlob`).

7.1.1.3. Trump Operations Group

Recoverable keys, and most of the security world administrator keys have a *trump operations group*, allowing:

- The safe permissions: `DuplicateHandle`, `GetACL`, `ReduceACL`.
- The following additional permissions: `GetAppData`, `SetAppData` and `ExpandACL`.
- Except in strict-FIPS worlds, `ExportAsPlain`.
- `MakeBlob` and `MakeArchiveBlob` actions.

This group requires `KNSO` authorization.

Example

```
.groups[2].flags= certifier_present FreshCerts 0x00000003
          .n_limits= 0
          .n_actions= 3
          .actions[0].type= OpPermissions
                     .details.oppermissions.perms= DuplicateHandle ExportAsPlain GetAppData SetAppData ReduceACL
ExpandACL GetACL 0x0000207d
          .actions[1].type= MakeBlob
                     .details.makeblob.flags= AllowKmOnly AllowNonKm0 AllowNullKmToken 0x00000023
                                    .kmhash absent
                                    .kthash absent
                                    .ktparams absent
                                    .blobfile absent
          .actions[2].type= MakeArchiveBlob
                     .details.makearchiveblob.flags= none 0x00000000
                                    .mech= Any
                                    .kahash absent
                                    .blobfile absent
          .certifier= 20 bytes 787720a1 e305a5af 6c8564af 3f6bfe55 122a733d
          .certmech absent
          .moduleserial absent
```

`0x787720a1···` is the hash of `KNSO` for this security world.

The effect of this group is to allow changes to the key's protection and application data, and (except in strict-FIPS worlds) export of its key material, authorized by an administrator cardset quorum.

7.1.1.4. Recovery blob group

Recoverable keys have a *recovery blob group* allowing `MakeArchiveBlob` under `KRE`. This is a single-use group, only used during key generation.

Example

```
.groups[3].flags= none 0x00000000
          .n_limits= 1
          .limits[0].type= Global
                    .details.global.id= 20 bytes 5989ba51 acfda26b e09d735a 389621fb f260b2e4
                              .max= 0x00000001 1
          .n_actions= 1
          .actions[0].type= MakeArchiveBlob
                     .details.makearchiveblob.flags= kahash_present 0x00000001
                                          .mech= BlobCryptv2kRSAeRijndaelCBC0hSHA512mSHA512HMAC
                                          .kahash= 20 bytes a7ae9935 f04f4680 c3046a02 c7fa838d d3ffd3ed
                                          .blobfile absent
          .certifier absent
          .certmech absent
          .moduleserial absent
```

`0xa7ae9935···` is the hash of `KRE`.

Note that module-protected keys are always recoverable, i.e., they have a trump operations group.

- Module-protected keys are protected by encryption under `KM`.
- `KM` is in turn protected by encryption under `LTM`.
- `KM` also has a trump operations group, necessary to support Administrator Card Set replacement.

The consequence of these facts is that control of an Administrator Card Set quorum is suffi cient to recover the plaintext of a module-protected key, even in the absence of the trump ops or recovery blob groups on that key. Only softcard- and OCS-protected keys can be non-recoverable.

7.1.2. Public Key ACLs

The standard ACL for a public key is much simpler. There is just one group with no limits or certification requirements, containing a permissive main use action and a blobbing action allowing any module key to be used. In practice, `KMWK` will be used.

Example

```
.groups[0].flags= none 0x00000000
          .n_limits= 0
          .n_actions= 2
          .actions[0].type= OpPermissions
                     .details.oppermissions.perms= DuplicateHandle ExportAsPlain GetAppData SetAppData ReduceACL
ExpandACL
                                                   Encrypt Verify UseAsBlobKey GetACL 0x000026fd
          .actions[1].type= MakeBlob
                    .details.makeblob.flags= AllowKmOnly AllowNonKm0 AllowNullKmToken 0x00000023
                                  .kmhash absent
                                  .kthash absent
```

```
                                .ktparams absent
                                .blobfile absent
        .certifier absent
        .certmech absent
        .moduleserial absent
```

7.1.3. Use Limits And SEE Application Keys

ACL construction can specify the following restrictions on the use of secret keys:

- Time limits. This is done with a `UseLim_Time` limit.
- Use count limits. This is done with a `UseLim_Auth` limit.
- Restriction to SEE machines. This is done by requiring certification from the SEE integrity key that will sign the SEE machine(s).

If any of these restrictions are present then then the main use permission group is split in two.

The first is similar to an ordinary main use group but only allows the `DuplicateHandle`, `ReduceACL` and `GetACL` permissions.

The second allows the remaining permissions (`GetAppData` and the permissions specific to the key type), and carries the restrictions.

An example group requiring authorization from a SEE integrity key:

```
.groups[ 1].flags= certmech_present 0x00000004
           .n_limits= 0
           .n_actions= 1
           .actions[0].type= OpPermissions
                      .details.oppermissions.perms= UseAsCertificate GetAppData Encrypt Decrypt Verify
UseAsBlobKey Sign
                                                    SignModuleCert 0x0000978a
           .certifier absent
           .certmech.hash= 20 bytes 2864a558 3012768e bf20da8a 4bfc7ee0 ceeee9cb
                    .mech= Any
           .moduleserial absent
```

`0x2864a558···` is the hash of the SEE integrity key.

SEE applications would present authorization in the form of a `SEECert` certificate, either hard-coding the signing key's hash or using `Cmd_GetWorldSigners` to retrieve it.

The time limit and use limit can be found in the `timelimit` and `pa_uselimit` members of an `NFKM_Key` structure. The identifier of the SEE integrity key can be found in the `identseeinteg` member.

7.1.4. Non-Volatile Use Limits (Key Counting)

This feature allows the use of a key to be counted and for a maximum number of uses to be enforced.

Counting is implemented by identifying a region of an NVRAM file that will contain the counter. Other file types, such as smartcards, cannot be used. This means that key counting is only implemented within the context of a single module. If the key is used on multiple modules then the counts must be added up separately.

"Blocks" of uses may be reserved in advance, allowing a trade-off between performance and accuracy.

Counting is implemented as a use limit on a permission group. A typical configuration would split the main use group in two (as above), with the "harmless" permissions, such as `GetACL`, uncounted and the important ones such as `Sign`, `Decrypt`, counted.

The limit may contain an upper bound on the number of uses of the key. If the requirement is just counting then this limit can be set to $2^{64}-1$.

The `mscapi` and `caping` components support key counting. The standard ACL generation code does not provide any standard way to add the extra use limits, though this could be done manually with an ACL they construct.

An example use limit allowing any number of uses but requiring exact counting:

```
.type= NonVolatile 4
.details.nonvolatile.flags= none 0x00000000
                    .file= (name) 11 bytes
                    .range.first= 0x00000000 0
                          .last= 0x00000007 7
                    .maxlo= 0xffffffff
                    .maxhi= 0xffffffff
                    .prefetch= 0x00000000 0
```

The name of the NVRAM file is chosen by the host. The convention used is that the first byte of the name is `1` for a `key-exchange` key, `2` for a `signature` key (RSA or DSA) and `3` for a `caping` key. (These are numeric values, not ASCII digits.)

7.1.5. NVRAM Keys (Keys In The Box)

This feature allows encrypted key blobs to be stored in the module's NVRAM rather than in the host's key management data directory.

To write a key blob to an NVRAM file, `Cmd_MakeBlob` requires extra parameters:

- `file.file` must identify the NVRAM file to write to.
- `file.kacl` must be a `KeyType_DKTemplate` key giving the ACL for the file. The ACL must permit blobs to be loaded from it.

The real key blob is written to the file and the blob field in the reply is just a pointer to the file, which `Cmd_LoadBlob` knows how to interpret.

To require that a key can *only* be written to an NVRAM file, the `MakeBlob` and `MakeArchive-Blob` actions in its ACL must specify additional restrictions:

- `blobfile.devs` must restrict key to being written to NVRAM files.
- `blobfile.aclhash` must specify the hash of the NVRAM file's ACL. This is best computed by temporarily constructing the `DKTemplate` key that will eventually be used to supply the ACL to the file and retrieving its key hash, rather than by attempting to hash the ACL directly.

Example `MakeBlob` action:

```
.actions[ 0].type= MakeBlob
            .details.makeblob.flags= AllowKmOnly AllowNonKm0 kmhash_present blobfile_present 0x00000047
                             .kmhash= 20 bytes 8fd82ce3 2e58f8e4 b559dddb 62de8cf6 5e66ce58
                             .kthash absent
                             .ktparams absent
                             .blobfile.flags= devs_present aclhash_present 0x00000003
                                      .devs= NVMem PhysToken 0x00000003
                                      .aclhash= 20 bytes 7e8d1b59 fd7e2de0 eb383c52 2468aed0 20010018
.certifier absent
.certmech absent
.moduleserial absent
```

Example NVRAM file ACL:

```
ACL.groups[0].flags= 0x0
             .limits= empty
            .actions[0].type= NVMemOpPerms
                      .details.perms= GetACL|LoadBlob
            .actions[1].type= FileCopy
                      .details.flags= 0x0
                               .to= NVMem|PhysToken
                               .from= NVMem|PhysToken
.groups[1].flags= certifier_present|FreshCerts
            .limits= empty
            .actions[0].type= NVMemOpPerms
.details.perms= Read|Write|Free|GetACL|LoadBlob|Resize
            .certifier= 787720a1 e305a5af 6c8564af 3f6bfe55 122a733d
```

Note the presence of `LoadBlob` in the permissions. The `FileCopy` action in the first group permits backup to a smartcard.

The name of the NVRAM file is chosen by the host. The convention is that the first byte of the name is 'b' (0x62) for the working blob and 'r' (0x72) for a recovery blob, with the

remaining 10 bytes being the first half of the key hash.

7.1.6. Application Key Generation APIs

The nCore commands are as follows:

- `Cmd_GenerateKey` and `Cmd_GenerateKeyPair` generate symmetric and asymmetric keys, respectively. They require certification from `KNSO` (via `KFIPS`) in a strict-FIPS security world.
- `Cmd_Import` imports a symmetric key or one half of an asymmetric key. The import of secret keys (symmetric keys and private halves of asymmetric keys) is forbidden in strict-FIPS security worlds.
- `Cmd_DeriveKey` can be used to combine keys to construct new ones, for instance merging keys via XOR or decrypting one key with another.

Supporting C APIs are declared in `nfkm.h:`

- `NFKM_newkey_makeaclx()` can be used to construct ACLs for new application keys. Unless the key has unusual requirements this should be used instead of attempting to construct the ACL manually. It may be convenient to construct the ACL with this function and then modify it.
- `NFKM_newkey_makeauth()` can be used to supply the `KFIPS` delegation when generating keys. General –purpose code can call it without caring whether it is in a strict-FIPS world or not since in a non-strict-FIPS world it does nothing.
- `NFKM_newkey_makeblobsx()` can be used to create working and recovery blobs for newly generated keys, storing the blob(s) in an `NFKM_Key` (for later storage to the key manage ment data directory). It assumes a reasonably standard ACL was used.
- `NFKM_newkey_writecert()` can be used to store key generation certificates for newly generated keys.
- `NFKM_recordkey()` writes an `NFKM_Key` structure to disk.

For NVRAM keys:

- The `NKFM_NKF_NVMemBlob` and `NFKM_NKF_NVMemBlobX` flags cause `NFKM_newkey_makeaclx()` to generate an ACL which requires the key to be written to an NVRAM file. The former uses a default ACL for the file (which allows backup to a smartcard), the latter allow the callers to specify the hash of the ACL.
- The same flags cause `NFKM_newkey_makeblobsx()` to write the working and recovery blobs to NVRAM files. As above the it is possible either to use a default ACL or to supply one (as a `DKTemplate` key). In either case, a handle to `KNV` must be supplied to autho rize writing to NVRAM.

The Python APIs are analogous:

- `nfkm.makeacl()` and `nfkm.makeaclandflags()` to construct ACLs.
- `nfkm.makeblobs()` to create blobs.
- `nfkm.writecert()` to store key generation certificates.
- `nfkm.recordkey()` writes the key data (including blobs) to disk.

7.1.7. Application Key Generation Example

This example shows:

- Identifying a usable module. This step is necessary if any attached module may not be fully usable, either because it is not in operational mode or if it is not in the security world used by the application.
- Constructing the key generation command, including automatic ACL construction. Don't construct ACLs manually unless you have special requirements or do not care about the security of the key.
- Generating blobs and module certificates for the key.
- Storing the key to the key management data directory.

```
#! /opt/nfast/python/bin/python
import nfkm,time

# nCore connectivity
conn=nfkm.connection()

# Find a usable module
for moduleinfo in nfkm.getinfo(conn).modules:
    if moduleinfo.state == 'Usable':
        module=moduleinfo.module

# Construct key generation command
cmd=nfkm.Command()
cmd.cmd='GenerateKeyPair'
cmd.args.flags='Certify'
cmd.args.module=module
cmd.args.params.type='DSAPrivate'
cmd.args.params.params.lenbits=1024

# Get sworld library to construct ACLs
cmd.args.aclpriv,privflags=nfkm.makeaclandflags(conn,['Sign'])
cmd.args.aclpub=nfkm.makeacl(conn,['Verify'], pubkey=True)

# Generate the key
print cmd,"\n"
reply = conn.transact(cmd)
print reply,"\n"

# Fill in the key information structure
key=nfkm.makeblobs(conn, privflags, reply.reply.keypriv, reply.reply.keypub)
key=nfkm.writecert(conn, module,
                   reply.reply.keypriv, reply.reply.certpriv, key)
key.appname="simple"
```

```
key.ident="example"
key.gentime=int(time.time())
print key,"\n"

# Save the key
nfkm.recordkey(conn,key)
```

For a roughly analogous example in C, see the `mkaeskey.c` example from CipherTools.

8. KNETI

8.1. KNETI

KNETI is the key used by a network HSM (for example, nShield Connect) to identify itself securely to a client, or by a client equipped with an nToken to identify itself securely to a network HSM.

A client's `KNETI`:

- is a 3072-bit DSA key (1024 bits for older firmware)
- is blobbed under `KM0` (see KM0)

The use of `KM0` to blob `KNETI`, combined with the desctruction of `KM0` when the module is reinitialized, explains why a hardserver sometimes logs the message "Failed to load kneti". The blob it is attempting to load is protected by an old `KM0`, which is no longer available. This only occurs with full (non-nToken) modules, since an nToken cannot normally be reinitialized.

A network HSM's `KNETI`:

- is a 3072-bit DSA key (1024 bits for older firmware)
- is protected by the network HSM's enclosure

9. Blobs

9.1. Blobs

9.1.1. Blob Concepts

A *blob* or *key blob* consists of a key, its ACL and application data encrypted in some way. A variety of encryption schemes are used depending on the type of security world and the purpose of the blob.

The most frequently used blobs are *working blobs*. These are for day-to-day use: an applica tion may start by loading one or more keys from their working blobs. Working blobs may be encrypted either under a module key or a logical token (i.e. an OCS or softcard). The module key or logical token functions both to keep the application key confidential and to authorize its use. These blobs always use the `Module` or `Token` format and creating them is authorized by `Act_MakeBlob`.

Even public keys are encrypted in working blobs, but the well-known module key is used, so no confidentiality is provided.

Recovery blobs encrypt the key (and ACL, etc.) under an asymmetric key (usually an RSA key). Since encryption uses the public key, `KRE` need not be loaded during the generation of recoverable application keys, but must be loaded to recover them. In the recovery process confidentiality and authorization are partially split: `KRE` provides confidentiality and the authorization to use open permission groups in the keys' ACL, while `KNSO` provides authoriza tion to use the 'trump operations' group. These blobs use the `Indirect` format (in DES3 worlds) or UserKey (in AES worlds) and creating them is authorized by `Act_MakeArchive-Blob`.

Temporary blobs are used during ACS replacement. These blobs use the `Direct` format (in DES3 worlds) or UserKey (in AES worlds) and creating them is authorized by `Act_-MakeArchiveBlob`.

The mechanisms used are discussed in the sections below.

9.1.2. AES Worlds

In AES worlds, key blobs are made using one of the 'new' blob encryption mechanisms:

Mechanism	Key encapsulation	KDF	PRF	Encryption	MAC
BlobCryptv2kHasheRijndael-CBC0hSHA1mSHA1HMAC	Symmetric	Proprietary	SHA-1	AES-CBC	HMAC-SHA1
BlobCryptv2kR-SAeRijndael-CBC0hSHA1mSHA1HMAC	RSA SVP	Proprietary	SHA-1	AES-CBC	HMAC-SHA1
BlobCryptv2kD-HeRijndael-CBC0hSHA1mSHA1HMAC	DH	Proprietary	SHA-1	AES-CBC	HMAC-SHA1
BlobCryptv2kHasheDES3CBC0hSHA1mSHA1HMAC	None	Proprietary	SHA-1	DES3-CBC	HMAC-SHA1
BlobCryptv2kR-SAeDES3CBC0hSHA1mSHA1HMAC	RSA SVP	Proprietary	SHA-1	DES3-CBC	HMAC-SHA1
BlobCryptv2kD-HeDES3CBC0hSHA1mSHA1HMAC	DH	Proprietary	SHA-1	DES3-CBC	HMAC-SHA1
BlobCryptv2kHasheRijndael-CBC0hSHA256mSHA256HMAC	Symmetric	Proprietary	SHA-256	AES-CBC	HMAC-SHA256
BlobCryptv2kR-SAeRijndael-CBC0hSHA256mSHA256HMAC	RSA SVP	Proprietary	SHA-256	AES-CBC	HMAC-SHA256
BlobCryptv2kD-HeRijndael-CBC0hSHA256mSHA256HMAC	DH	Proprietary	SHA-256	AES-CBC	HMAC-SHA256
BlobCryptv2kHasheRijndael-CBC0hSHA512mSHA512HMAC	Symmetric	Proprietary	SHA-512	AES-CBC	HMAC-SHA512

Mechanism	Key encapsulation	KDF	PRF	Encryption	MAC
BlobCryptv2kR-SAeRijndael-CBC0hSHA512mSHA512HMAC	RSA SVP	Proprietary	SHA-512	AES-CBC	HMAC-SHA512
BlobCryptv2kD-HeRijndael-CBC0hSHA512mSHA512HMAC	DH	Proprietary	SHA-512	AES-CBC	HMAC-SHA512
BlobCryptv3-kNoneeAESCBC0dCTRC-MACmSHA256H-MAC	Symmetric	SP800-108	AES-CMAC	AES-CBC	HMAC-SHA256
BlobCryptv3kR-SAOAE-PeAESCBC0dCTRCMACmSHA256HMAC	RSA OAEP	SP800-108	AES-CMAC	AES-CBC	HMAC-SHA256
BlobCryptv3-kNoneeAESCBC0dCTRC-MACmSHA512H-MAC	Symmetric	SP800-108	AES-CMAC	AES-CBC	HMAC-SHA512
BlobCryptv3kR-SAOAE-PeAESCBC0dCTRCMACmSHA512HMAC	RSA OAEP	SP800-108	AES-CMAC	AES-CBC	HMAC-SHA512
BlobCryptv4kECDHeAESmC-TRhSHA256mSHA256HMAC	ECIES	SP800-56Ar2	SHA-256	AES-CTR	HMAC-SHA256
BlobCryptv4kECDHeAESmC-TRhSHA512mSHA512HMAC	ECIES	SP800-56Ar2	SHA-512	AES-CTR	HMAC-SHA512

The blobbing key might be a module key, a logical token or an application key.

In blobcryptv2 and later, the blob itself is encrypted and then protected with a MAC under single-use session keys derived from the blobbing key (in the symmetric case) or encrypted under it (asymmetric), rather than using the blobbing key directly.

Application key blobs are stored in the privblob member of the NFKM_Key structure, or pubblob for public keys. Recovery blobs are stored in the privblobrecov member of the NFKM_Key structure.

9.1.2.1. Module Key Blobs

For module-protected keys, the blobbing key is KM; for public keys it is KMWK. The mechanism is selected automatically (and will be the 'working' mechanism listed above). The key to be blobbed must have a MakeBlob action in its ACL with the following characteristics:

- It must include the AllowKmOnly flag.
- If the module key is not KM0, it must include the AllowNonKM0 flag.
- The flags must not include AllowNullKMToken. This flag only applies to logical token blobs.
- If kmhash is present in the ACL, it must match the hash of the module key.

9.1.2.2. Logical Token Blobs

For application keys the blobbing key is the LTU from an OCS or softcard; for security world administrator keys it is a logical token from the ACS. The mechanism is selected automatically (and will be the 'working' mechanism listed above). The key to be blobbed must have a MakeBlob action in its ACL with the following characteristics:

- If the logical token is associated with a null module key, the flags must include AllowNullKMToken.
- If the logical token is associated with any module key other than KM0, the flags must include AllowNonKM0.
- If kmhash and/or kthash are present in the ACL, they must match the hash of the module and/or logical token respectively.
- If ktparams is present in the ACL, it must be compatible with the characteristics of the logical token (see below).

A logical token is compatible with a TokenParams structure (in an ACL) if all the following are true:

- All the flags present in the structure are present in the logical token
- The logical token has at least as big a quorum (sharesneeded) as given in the structure.
- The logical token has no more total shares than given in the structure.
- If the structure specifies a time limit, the token has a time limit which is no higher than the one in the structure.

9.1.2.3. Application Key Blobs

The blobbing key is a second application key, which must have the `UseAsBlobKey` permission. This is used for recovery blobs, using `KRE`. The mechanism is specified in the command (and in a security world, will be the 'recovery' mechanism listed above, or during ACS recovery the 'temporary' mechanism).

The key to be blobbed must have a `MakeArchiveBlob` action in its ACL with the following characteristics:

- Either a mechanism of `Mech_Any`, or the same mechanism requested in the command.
- If `kahash` is present in the ACL, it must match the hash of the application key.
- If `blobfile` is present in the ACL, it must be compatible with the `blobfile` in the command.

9.1.3. DES Worlds

Triple-DES worlds should no longer be used.

9.1.4. Public Key Blobs

In `DLf1024s160mDES3`, `DLf1024s160mRijndael` and `DLf3072s256mRijndael` worlds, public key blobs always use the DES3 KMWK and the old blobbing scheme. In `DLf3072s256mAESc-SP800131Ar1` worlds they use the AES KMWK and the working blob mechanism for that world.

9.1.5. PIN (Passphrase) Recovery Blobs

PIN recovery blobs consist of the hash of the PIN, encrypted using an RSA key. These are not key blobs and the module has no special knowledge about their format – they are simply encrypted messages created by the NFKM library. So the term "blob" is a bit misleading in this context.

- In `DLf3072s256mRijndael` and `DLf3072s256mAEScSP800131Ar1` worlds, OAEP is used.
- In `ECp521mAES` worlds, ECIES is used.