Application Notes

Key Attestation Format

07 October 2024

Table of Contents

This appnote describes the design information for constructing and verifying key attestations based on KLF2 warrants. For information on generating and verifying nShield key attestation bundles using the `nfkmattest` command line tool, see Introduction.

The current version of this appnote assumes the use of nShield libraries for the processing of any relevant data. If required, the same can be achieved using corresponding third-party cryptographic functions and custom format marshaling.

1. Construction

A key attestation bundle is a JSON object containing all the required data to verify that a key has been generated by an nShield HSM. It is the standard used by the Key Attestation Verifier and is encouraged for any other implementations for verifying nShield keys. All field names, data types and requirements are listed in the table below. For a worked example for retrieving the required fields using Python, nShield libraries and an HSM connection, see Construction example.

All nShield data types (e.g. M_KeyData) are marshaled to byte strings. DDDS objects (e.g. warrants) are marshaled to byte strings. Byte strings are encoded with URL-safe base64 encoding, from RFC4648.

There are no ordering constraints on bundle representation.

1.1. Bundle fields

Field	Type	Encoding	Presence	Description
pubkeydata	M_KeyData	marshal+base64	Always	Public key material in nCore format (including any domain parameters)
kcmsg	M_ModCertMsg	marshal+base64	Always	The key generation certificate body (i.e. ModCertType_KeyGen)
kcsig	M_CipherText	marshal+base64	Always	The signature on the key generation certificate under KML
modstatemsg	M_ModCertMsg	marshal+base64	Always	A module state certificate (i.e. ModCertType_StateCert). See below for further requirements.
modstatesig	M_CipherText	marshal+base64	Always	The signature on the module state certificate under KLF2.

Field	Type	Encoding	Presence	Description
warrant	DDDS list	marshal+base64	Always	The DDDS encoding of the generating HSM's KLF2 warrant.
root	string	none	Always	The name of the warranting root used in this certificate. This will always be KWARN-1 for nShield HSMs.
knsopub	M_KeyData	marshal+base64	Persistent keys	KNSO public key (needed to validate trump ops groups and actions)
hkre	M_KeyHashEx	marshal+base64	Recoverable keys	Hash of KRE (needed to validate recovery blobbing action)
hkra	M_KeyHashEx	marshal+base64	Recoverable keys	Hash of KRA (needed to reconstruct world binding certs)
hkfips	M_KeyHashEx	marshal+base64	Persistent keys in FIPS worlds	Hash of KFIPS (needed to reconstruct world binding certs)
hkmc	M_KeyHashEx	marshal+base64	Persistent keys	Hash of KMC (needed to reconstruct world binding certs)
hkm	M_KeyHashEx	marshal+base64	Persistent keys	Hash of KM (needed to validate module configuration)
CertKMaKMCbKNSO	M_CipherText	marshal+base64	Persistent keys in non-FIPS worlds	Signature on world binding cert
CertKMaKMCaKFIPSbKNSO	M_CipherText	marshal+base64	Persistent keys in FIPS worlds	Signature on world binding cert
CertKREaKRAbKNSO	M_CipherText	marshal+base64	Recoverable keys	Signature on world binding cert

Field	Type	Encoding	Presence	Description
ciphersuite	string	none	Persistent keys	Ciphersuite name for security world from the NFKM_CipherSuite enumeration (e.g. DLf3072s256mAESc SP800131Ar1)

1.2. Persistent and recoverable keys

'Recoverable' has its normal nShield meaning:

- The key's ACL has MakeArchiveBlob action permitting blobbing under KRE (recovery key).
- The key's ACL has a 'trump ops' permission group, requiring KNSO (security officer's key) authorization and permitting any action (generally including ExpandACL and/or ExportAsPlain).

'Persistent' has a special meaning here, not connected to the idea of cardsets being persistent. A persistent key has at least one blobbing action in its ACL, meaning it can be saved to some kind of storage and used long after generation. A non-persistent key has no blobbing actions. Note that recoverable keys are always persistent.

2. Construction example

The code below will showcase how to obtain the required data for the bundle in Python for a simple key with ident `test`. All formatting and encoding functions used are defined in a further section below for readability.

For setup, the `nfkm` and `nfwar` libraries are imported and a connection with the hardserver is established:

```
import nfkm
import nfwar
conn = nfkm.connection(needworldinfo=True)

esn = "ABCD-ABCD-ABCD"
appname = "simple"
ident = "test"
bundle = {}
```

The root name and warrant data can be obtained from the warrant file:

```
warrant = nfwar.Warrant()
warrant.load(f"/opt/nfast/kmdata/warrants/{esn}")

bundle["root"] = warrant._root
bundle["warrant"] = format_bytes(warrant.warrant)
```

Some of the required fields can be retrieved from the key object:

```
key = nfkm.findkey(conn, appname, ident)

bundle["pubkeydata"] = format_pubkey(key.pubblob)
bundle["kcmsg"] = format_bytes(key.kcmsg)
bundle["kcsig"] = format_signature(key.kcsig)
bundle["modstatemsg"] = format_bytes(key.modstatemsg)
bundle["modstatesig"] = format_signature(key.modstatesig)
```

You can also retrieve some fields from the world info:

```
world = nfkm.getinfo(conn)

bundle["ciphersuite"] = str(world.suite)
key_hash_fields = ["hkre", "hkra", "hkfips", "hkmc", "hkm"]
for field_name in key_hash_fields:
  key_hash = world[field_name]
  # skip missing hash, e.g. hkfips in non-FIPS worlds or hkre for non-recoverable keys
  if key_hash != nfkm.Hash():
    bundle[field_name] = format_hash(key_hash)
```

The remaining fields can be extracted from the world file:

```
worldfile = open("/opt/nfast/kmdata/local/world", "rb").read()
```

```
worldfile, _ = nfkm.unmarshal(worldfile, nfkm.KeyMgmtFile)

bundle["knsopub"] = format_pubkey(get_kmf_entry(worldfile, "BlobPubKNSO"))
for cert_name in ["CertKMaKMCbKNSO", "CertKMaKMCaKFIPSbKNSO", "CertKREaKRAbKNSO"]:
  cert_value = get_kmf_entry(worldfile, cert_name)
  # skip missing cert, e.g. CertKMaKMCaKFIPSbKNSO in non-FIPS worlds
  if cert_value is not None:
    bundle[cert_name] = format_signature(cert_value)
```

2.1. Warrant for nShield 5 modules

For nShield 5 modules, the warrant doesn't get installed on the filesystem but instead exists only on the module. It can be requested using the command line function `retrievewarrants` or directly from the module:

```
for module in world.modules:
  if module.esn == esn and module.state == "Usable":
    cmd = nfkm.Command(
      ['GetModuleState', 0, module.module, 'attribs_present', ["WarrantKLF2"]]
    )
    reply = conn.transact(cmd)
    attribs_list = reply.reply.state
    if len(attribs_list.attribs) == 1:
      warrant_attrib = attribs_list.attribs[0]
      if (isinstance(warrant_attrib, nfkm.ModuleAttrib) and
          warrant_attrib.tag == "WarrantKLF2"):
        bundle["warrant"] = format_bytes(bytes(warrant_attrib.value.warrant))
```

2.2. Module state certificate requirements

The module state certificate always needs the ESN and KML or KMLEx attributes. For persistent keys the certificate also needs the KNSO or KNSOEx attributes and the KMList or ModKeyInfoEx attributes. Prior to Security World v13.5, the module state certificate would be generated too early and wouldn't contain KNSO.

The certificate can be regenerated as long as the right module is available and hasn't been reinitialized.

```
# retrieve correct HKML from incomplete modstatemsg
original_hkml = None
modstatemsg, _ = nfkm.unmarshal(modstatemsg, nfkm.ModCertMsg)
for attrib in modstatemsg.data.state.attribs:
  if attrib.tag == "KML":
    original_hkml = nfkm.KeyHashEx(["SHA1Hash", attrib.value.hkml])
  if "KMLEx" in nfkm.ModuleAttribTag.words and attrib.tag == "KMLEx":
    original_hkml = attrib.value.hk

# regenerate modstatemsg and modstatesig for correct module
for module in world.modules:
  if (module.esn == esn and module.state == "Usable" and
      isinstance(original_hkml, nfkm.ByteBlock) and
      module.hkml == original_hkml.data.hash):
    cmd = nfkm.Command(["SignModuleState", 0, module.module, 0, "KLF2"])
```

```
    cert = conn.transact(cmd).reply.cert
    bundle["modstatemsg"] = format_bytes(cert.modcertmsg)
    bundle["modstatesig"] = format_signature(cert.signature)
```

2.3. Formatting functions

```
import base64
import choosealg

def format_bytes(value):
  return str(base64.urlsafe_b64encode(value), "ASCII")

def format_signature(sig):
  if not isinstance(sig, nfkm.CipherText):
    if isinstance(sig, bytes):
      sig = str(sig, "ASCII")
    sig = str(sig, "ASCII").split(" ")
    sig = nfkm.CipherText(sig)
  sig = nfkm.marshal(sig)
  return format_bytes(sig)

def format_hash(hash_):
  if isinstance(hash_, nfkm.Hash):
    hash_ = nfkm.KeyHashEx(["SHA1Hash", hash_])
  return format_bytes(nfkm.marshal(hash_))

def unpack_blob(blob_data):
  public_blob_keys = {
    nfkm.Hash("c2be99fe1c77f1b75d48e2fd2df8dffc0c969bcb"):
      nfkm.KeyData(["Rijndael", "00" * 32]),
    nfkm.Hash("1d572201be533ebc89f30fdd8f3fac6ca3395bf0"):
      nfkm.KeyData(["DES3", "01" * 24]),
  }
  blob_data, _ = nfkm.unmarshal(blob_data, nfkm.BlobData)
  blobkey = public_blob_keys[blob_data.exdata.hkm]
  if blob_data.imech & 0x80000000:
    mech = blob_data.imech & 0x7FFFFFFF
    cipher_text = nfkm.CipherText([mech, blob_data.ka_data])
  else:
    cipher_text = nfkm.CipherText(["DES3wSHA1", blob_data.ka_data, blob_data.datalen])
  plain_text = choosealg.decrypt(blobkey, cipher_text)
  bcb, _ = nfkm.unmarshal(plain_text.data.data, nfkm.BlobCryptBlock)
  key, _ = nfkm.unmarshal(bcb.key, nfkm.KeyData)
  return key

def format_pubkey(k):
  if isinstance(k, nfkm.ByteBlock):
    k = unpack_blob(k)
  return format_bytes(nfkm.marshal(k))

def get_kmf_entry(kmf, entry_type):
  entry_type = nfkm.KeyMgmtEntType(entry_type).getvalue()
  for entry in kmf.entries:
    if entry.type == entry_type:
      return entry.data
  return None
```

3. Verification

Verifying an attestation bundle requires a series of steps to, at a minimum, check the chain of trust between a key and its generating HSM. All verifications mentioned can be performed by any cryptographic provider as long as the required algorithms are supported. For a worked example for verification using Python and nShield libraries, see Verification example.

3.1. Overview

3.1.1. Approaches

There are two possible approaches to verifying an attestation.

The first approach just verifies that the key was created in an nShield HSM. It says nothing about the policy controlling use of the key. If the reason for verification is to meet a formal requirement for keys generated in an HSM, and there is an appropriate level of trust between the key owner and the verifier, this approach is quite adequate.

The second approach does extensive checks on the policy controlling the use of the key, i.e. its Access Control List (ACL). This may be more appropriate if there is only limited trust between the key owner and the verifier. The nShield Key Attestation Verifier implements this second approach but may be unnecessary overhead for custom verifiers.

For the first approach the following verification steps are required:

- WV1 (warrant verification)
- MSCV1 (module state certificate verification)
- MSCV2 (extracting KML from the verified module state certificate)
- KGCV1 (key generation certificate verification)
- KGCV2 (public key binding)
- CSRL1 (CSR binding, if a CSR is in use)

The following steps are not required in the first approach:

- MSCV3, MSCV4, MSCV5
- WBCV1, WBCV2, WBCV3, WBCV4, WBCV5
- ACLV1, ACLV3, ACLV4
- WB1, WB2, WB3, WB5, WB6
- RB1, RB2, RB3, RB5

- KV1, KV2, KV3

For the second approach, all steps are required (except possibly CSRL1, KV1, KV2 and KV3, depending on the situation and the verifier's requirements).

3.1.2. Key Hashes

All key hashes (denoted H(...) below) are nCore key hashes. At present they are always SHA-1 hashes but in the future wider hashes will be supported. For further information on the format, see nCore Key Hash.

3.1.3. Certificates

A number of certificates are used in the verification process.

- a warrant (see nShield Warrants for details) provides the KLF2 and ESN (given trust in the root key)
- a module state certificate provides the KML, the module's HKNSO and list of installed module keys (given trust in KLF2)
- CertKMaKMCbKNSO or CertKMaKMCaKFIPSbKNSO allow validation of KM and KMC (given trust in KNSO)
- CertKREaKRAbKNSO allows validation of KRE and KRA (given trust in KNSO)
- a key generation certificate gives the application key ACL, given trust in KML

Note that the material supplied for the world binding certificates (i.e. Cert*) is the signatures only. The certificate bodies are reconstructed during the verification process. These certificates don't provide any key hashes directly, but they do allow validation of key hashes supplied by other fields from the attestation bundle.

3.1.4. Ciphersuites and recovery mechanisms

The following recovery mechanisms correspond to the given ciphersuites.

ciphersuite	recovery mechanism
CipherSuite_DLf1024s160mDES3	Mech_RSApPKCS1
CipherSuite_DLf1024s160mRijndael	Mech_BlobCryptv2kRSAeRijndaelCBC0hSHA512mSHA512HMAC
CipherSuite_DLf3072s256mRijndael	Mech_BlobCryptv2kRSAeRijndaelCBC0hSHA512mSHA512HMAC

ciphersuite	recovery mechanism
CipherSuite_DLf3072s256mAEScSP800131Ar1	Mech_BlobCryptv3kRSAOAEPeAESCBC0dCTRCMACmSHA512HMAC

3.2. Unpacking

The first step is to unpack the bundle from JSON. There is no signature checking here, the JSON bundle is just a container for certificates, key hashes, etc. If the implementation language has a distinction between 'safe' and 'unsafe' JSON decoders, the safe one should be used.

3.3. Warrant verification

Step WV1: Verify warrant under configured root key, yielding the KLF2 and ESN. If verification fails, exit with an error. Details on the structure and verification of a warrant can be found in Warrant format. It is essential to decide what root key will be trusted. Normally this would be KWARN-1 for nShield HSMs.

3.4. Module state certificate verification

The module state certificate is verified under KLF2. Having verified it following needs to be extracted:

- the module's ESN (the ESN attribute)
- the module's KML (the KML or KMLEx attributes)

If present the following can be retrieved:

- the module's HKNSO (the KNSO or KNSOEx attributes). This must be consistent with the knsopub field of the bundle if that is present.
- the module's module key list (the KMList or ModKeyInfoEx attributes). The hkm field must be among this list if it is present.

Step MSCV1: Verify modstatemsg and modstatesig under KLF2. If verification fails, exit with an error.

Step MSCV2: Extract ESN, KML and (optionally) HKNSO and module key list. If they are not present, exit with an error.

Step MSCV3: Check ESN from the module state certificate matches ESN from the warrant.

If not, exit with an error.

Step MSCV4: Check H(knsopub) matches HKNSO, if knsopub present. If knsopub present but they don't match, exit with an error.

Step MSCV5: Check hkm appears in module key list, if hkm present. If hkm is present but can't be found, exit with an error.

Attestation bundles generated by standard nShield tooling will always include knsopub, so a verifier might make the simplifying assumption that it is always present. The only situation in which it would be appropriate to omit from an attestation bundle is if no part of the key's ACL depends (even indirectly) on KNSO, which would rule out any blobbing actions.

3.5. World binding certificate verification

If any of the world binding certificates (fields Cert*) are present, they must be verified under KNSO.

It is expected that only one of CertKMaKMCbKNSO and CertKMaKMCaKFIPSbKNSO will be present. Verification is only possible if knsopub is present. To do this, it's necessary to reconstruct the certificate subject.

Each certificate consists of a raw bytes representation of a header string and a series of key hashes. At the time of writing, these are be M_KeyHash representations. The header and keys depend on the certificate type and ciphersuite:

- CertKMaKMCbKNSO:
  - ciphersuite DLf1024s160mDES3:

    `"Module keys\0" || H(KNSO) || H(KM) || H(KMC)`

  - ciphersuite DLf1024s160mRijndael:

    `"Module keys: KM type Rijndael\0" || H(KNSO) || H(KM) || H(KMC)`

  - other ciphersuites:

    `"Module keys: suite = <ciphersuite>\0" || H(KNSO) || H(KM) || H(KMC)`

    (e.g. `"Module keys: suite = DLf3072s256mRijndael\0" || H(KNSO) || H(KM) || H(KMC)`)

- CertKMaKMCaKFIPSbKNSO:
  - ciphersuite DLf1024s160mDES3:

    `"Module setup, FIPS3\0" || H(KNSO) || H(KM) || H(KMC) | H(KFIPS)`

    - ciphersuite DLf1024s160mRijndael:

      `"Module setup, FIPS3; KM type Rijndael\0" || H(KNSO) || H(KM) || H(KMC) | H(KFIPS)`

    - other ciphersuites:

      `"Module setup, FIPS3; suite = <ciphersuite>\0" || H(KNSO) || H(KM) || H(KMC) | H(KFIPS)`

      (e.g. `"Module setup, FIPS3; suite = DLf3072s256mRijndael\0" || H(KNSO) || H(KM) || H(KMC) | H(KFIPS)`)

- CertKREaKRAbKNSO: `"Card Recovery\0" || H(KNSO) || H(KRE) || H(KRA)`

Step WBCV1: Verify CertKMaKMCbKNSO under knsopub if it the certificate is present. If any dependency (including knsopub) is absent or verification fails, exit with an error.

Step WBCV2: Verify CertKMaKMCaKFIPSbKNSO under knsopub if it the certificate is present. If any dependency (including knsopub) is absent or verification fails, exit with an error.

Step WBCV3: Verify CertKREaKRAbKNSO under knsopub if it the certificate is present. If any dependency (including knsopub) is absent or verification fails, exit with an error.

At this point some of the key hashes are trusted:

- From steps MSCV1/2 the KML must have come from the HSM identified by ESN
- From steps MSCV1/2 the given module was configured to trust HKSNO
- From step MSCV4 the known knsopub corresponds to that HKNSO
- From step MSCV5 the known hkm was installed as a module key in this module

Step WBCV4: If CertKREaKRAbKNSO is present (and verified) then the hkre field can be trusted. Otherwise, delete the hkre field.

Step WBCV5: If CertKMaKMCaKFIPSbKNSO or CertKMaKMCbKNSO are present (and verified) then the hkm field can be trusted. Otherwise, delete the hkm field.

The further known facts at this stage:

- hkm must be the KM of the same security world as KNSO
- The module was properly configured (it uses the given security world's KNSO and has KM installed as a module key)
- hkre is the given security world's KRE

3.6. Key generation certificate verification

Step KGCV1: Verify kcmsg and kcsig under KML (as recovered from the module state certificate). If verification fails, exit with an error.

This yields the application key hash, its key generation parameters and its generation-time ACL.

Step KGCV2: Check that the key hash in kcmsg matches H(pubkeydata). If they do not match, exit with an error.

3.7. ACL validation

An nShield ACL consists of a collection of permission groups. In turn each permission group contains a collection of actions.

A permission group may also be marked as requiring authorization from a certifier, meaning a key identified by its nCore key hash. A permission group may contain one or more use limits, applying constraints to how often the actions within it may be used. Actions permit use of the key, e.g. to generate a signature or to create a blob containing the key.

The process of validating an ACL involves iterating through the permission groups and inspecting them. In most cases for each permission group the actions are iterated over.

3.7.1. Trump ops groups

Step ACLV1: If any of the certifier fields (certifier, certmech or certmechex) matches H(KNSO) (from module state certificate) then the permission group must be the 'trump ops' group, i.e. the set of extra actions permitted under ACS authorization to carry out recovery operations. Such groups are disregarded from all the checks below, but the key is marked as recoverable.

3.7.2. Main use actions

The verifier may wish to apply a policy to what the key can be used for, e.g. to accept signature but forbid encryption.

3.7.2.1. Act_OpPermissions

For Act_OpPermissions the possible bits are categorized as follows:

- harmless: DuplicateHandle, GetAppData, ReduceACL, GetACL
- signature: Sign, Verify, SignModuleCert, UseAsCertificate
- encryption: Encrypt, Decrypt
- forbidden: ExportAsPlain, SetAppData, ExpandACL, UseAsBlobKey, UseAsKM, UseAsLoaderKey

Step ACLV3: If the action is Act_OpPermissions, inspect the permitted operations according to the table above.

- If any are forbidden, exit with an error.
- If any are inconsistent with the verifier's requirements, exit with an error.

3.7.2.2. Act_DeriveKey and Act_DeriveKeyEx

A verifier may choose to accept derivekey actions consistent with their requirements. Any unwanted or unrecognized derivekey actions should result in exiting with an error, as per step ACLV4 below. DeriveMech_PublicFromPrivate is generally considered harmless.

3.7.3. Working blobs (Act_MakeBlob)

Act_MakeBlob actions determine how the key is protected. They can be interpreted according to the following rules:

Step WB1: If neither AllowKmOnly nor kthash_present are set in flags, exit with an error. The protection is incoherent or too loose.

Step WB2: If KM is not trusted (see step WBCV5), or kmhash is not present or kmhash does not match hkm , exit with an error. All application keys must be protected by, at least, KM.

Step WB3: If AllowNullKMToken is present, exit with an error. All application keys must be protected by, at least, KM.

Step WB5: If flags includes AllowKmOnly then the key is module-protected.

Step WB6: If kthash is present, check that ktparams is present. If not, exit with an error. The token parameters are necessary to correctly interpret the key protection type.

Step WB7: If kthash is present, check whether the ktparams flags includes AllowSoftSlots. If it does then the key is softcard-protected; otherwise it is cardset protected.

3.7.4. Recovery blobs (Act_MakeArchiveBlob)

Act_MakeArchiveBlob actions permit the creation of recovery blobs. They can be interpreted according to the following rules:

Step RB1: If KRE is not trusted (see step WBCV4), exit with an error. Either the world is not recoverable or the recovery key is unknown.

Step RB2: If kahash is not present, or does not match hkre, exit with an error. The recover blob would be encrypted with the wrong key.

Step RB3: If the recovery mechanism is wrong for the ciphersuite, exit with an error.

Step RB5: If the above pass, the key is marked recoverable, in the same way as in ACLV1 above.

3.7.5. Other Actions

Step ACLV4: If any other action is found, exit with an error.

3.7.6. Outcome of ACL validation

After ACL validation:

- The key's recoverability is known, either from step ACLV1 or RB5:
  - ACLV1 indicates a trump ops group. The ACS holders can carry out recovery operations if the key is loaded either via the working blob or the recovery blob.
  - RB5 indicates that a recovery blob exists. The ACS holders can (at a minimum) load and use the key without access to the cardset or softcard protecting it.
  - Only if neither of these hold is the key to not considered to be recoverable.
- The key's protection type is known, either from step WB5 or WB7.

Step ACLV5: Determine protection type based on WB5 and WB7. If there are multiple Act_MakeBlob actions then there might be an inconsistent set of protection types. This could be reported as an error, or the least secure can be picked (i.e. the first in the list no protection, module protection, softcard protection, cardset protection).

3.8. Key validation

A verifier may choose to apply some kind of policy to the key. The fine detail is not part of the core attestation design, since different verifiers will have different requirements, but some broad statements can be made:

Step KV1: Apply local policy to key generation parameters. If they are not suitable, exit with an error.

Step KV2: Apply local policy to public key material. If they are not suitable, exit with an error.

Step KV3: Apply local policy to key protection and recoverability properties. If they are not suitable, exit with an error.

3.9. CSR linkage

A key attestation may be embedded in a PKCS#10 Certificate Signing Request (or a similar object). In this case, as well as verifying the attestation and verifying the CSR through the normal CSR verification and validation processes, it's essential that the two objects be properly linked.

Step CSRL1: pubkeydata should be compared with the public key from the CSR. If they do not match, exit with an error.

Note: Steps ACLV2, WB4 and RB4 existed in earlier versions of the design and were not renumbered when removed.

4. Verification example

The code below will showcase how to verify an attestation bundle in Python. It is split up to mirror the sections in Verification for readability. This implementation always raises an exception at the first error, but a viable alternative is to report all issues and allow for all the steps to complete. The steps for key validation and CSR linkage have been omitted but can be added once all relevant information about the key has been processed below.

4.1. Parsing and marshalling/format functions

While this example uses nShield libraries for cryptography and marshalling/formatting data, the relevant fields can be extracted from the byte strings directly.

The example also uses a simple `parse_bytes` function to format the raw base64 strings into ByteBlocks.

```
def parse_bytes(value):
  return nfkm.ByteBlock(base64.urlsafe_b64decode(value), fromraw=1)
```

4.2. Unpacking

The bundle is unpacked from the JSON file.

```
import json

bundle = open("path/to/bundle").read()
bundle = json.loads(bundle)
```

4.3. Warrant verification (WV1)

As outlined in nShield Warrants, the warrant will be a list containing the name of the root key followed by two or more certificates. The first certificate must be verified under the root key, and each subsequent certificate is verified under the key contained in the payload of the previous. The root key should always be KWARN-1 for nShield warrants. Its public key data is provided in nShield root key and is refered to as KWARN_1.

The code below is simplified for readability, but a more complete implementation might also include:

- checks to ensure the key and mechanism types are as expected
- optional support for `FieldUpgradeModuleInformation`, which will also have the fields

KLF2pub, KLF2mech and ElectronicSerialNumber

- sanity checks for any assumptions about the structure of the warrant

```
import nfddds
import choosealg
import nfkm

warrant = parse_bytes(bundle["warrant"])
warrant = nfddds.decode(warrant)

if str(warrant[0]) != "KWARN-1":
  raise Exception("Warrant error: Incorrect root name")

trusted_pubkey = KWARN_1
warrant_esn = None
for cert in warrant[1:]:
  # Parse the certificate
  payload = cert[nfddds.Symbol("Payload")]
  sig = cert[nfddds.Symbol("Signature")]

  # Convert to nCore format
  rslen = len(sig) // 2
  r_part = sig[:rslen]
  s_part = sig[rslen:]
  sig = nfkm.CipherText(
    ["ECDSAhSHA512", int.from_bytes(r_part, "big"), int.from_bytes(s_part, "big")]
  )

  # Verify the signature
  plain_text = nfkm.PlainText(["Bytes", payload])
  choosealg.verify(trusted_pubkey, sig.mech, plain_text, sig) # raises error if verify failed

  # Parse the payload
  payload = nfddds.decode(payload)
  cert_type = payload[nfddds.Symbol("WarrantCertificateType")]

  # Retrieve signing key and mechanism for next certificate or KLF2 in the last certificate
  if cert_type == nfddds.Symbol("Delegation"):
    trusted_pubkey = payload[nfddds.Symbol("DelegateKey")]
    mech = payload[nfddds.Symbol("SigMech")]
  elif cert_type == nfddds.Symbol("ModuleInformation"):
    trusted_pubkey = payload[nfddds.Symbol("KLF2pub")]
    mech = payload[nfddds.Symbol("KLF2mech")]
    warrant_esn = payload[nfddds.Symbol("ElectronicSerialNumber")]
  else:
    raise Exception("Warrant error: unrecognized certificate type")

  # Convert the key to nCore format
  trusted_pubkey = nfkm.KeyData(
    ["ECDSAPublic", "NISTP521", 0, trusted_pubkey[3][0], trusted_pubkey[3][1]]
  )

  # If it was a module information certificate then stop; we have no further delegation authority
  if warrant_esn is not None:
    break

if warrant_esn is None:
  raise Exception("Warrant error: No module information certificate found")

klf2 = trusted_pubkey
```

4.4. Module state certificate verification (MSCV1-5)

Verifying the module state certificate now uses the KLF2 obtained during the previous step.

```
modstatemsg = parse_bytes(bundle["modstatemsg"])
modstatesig = parse_bytes(bundle["modstatesig"])
modstatesig, _ = nfkm.unmarshal(modstatesig, nfkm.CipherText)
plain_text = nfkm.PlainText(["Bytes", modstatemsg])
if modstatesig.mech not in KLF2_MECHS:
  raise Exception("Module state certificate error: unexpected KLF2 signature mechanism")

# MSCV1 - verify module state cert
choosealg.verify(klf2, modstatesig.mech, plain_text, modstatesig)

# MSCV2 - extract the attributes we need
modstatemsg, _ = nfkm.unmarshal(modstatemsg, nfkm.ModCertMsg)
esn = None
kml = None
hknso = None
hkms = []

for attrib in modstatemsg.data.state.attribs:
  if attrib.tag == "ESN":
    esn = attrib.value.esn
  elif attrib.tag == "KML":
    kml = attrib.value.kmlpub
  elif "KMLEx" in nfkm.ModuleAttribTag.words and attrib.tag == "KMLEx":
    kml = attrib.value.kmlpub
  elif attrib.tag == "KNSO":
    hknso = nfkm.KeyHashEx(["SHA1Hash", attrib.value.hknso])
  elif "KNSOEx" in nfkm.ModuleAttribTag.words and attrib.tag == "KNSOEx":
    hknso = attrib.value.hknso
  elif attrib.tag == "KMList":
    hkms = [nfkm.KeyHashEx(["SHA1Hash", km.hk]) for km in attrib.value.hkms]
  elif "ModKeyInfoEx" in nfkm.ModuleAttribTag.words and attrib.tag == "ModKeyInfoEx":
    hkms = [mki.hk for mki in attrib.value.kms]

# MSCV2 - ESN and KML must be present
if not esn:
  raise Exception("Module state certificate error: No module serial number found")
if not kml:
  raise Exception("Module state certificate error: no KML public key found")
# MSCV3 - ESN must match that from warrant
if esn != warrant_esn:
  raise Exception("Module state certificate error: ESN does not match warrant")

# MSCV4 - If KNSO is present, verify it
if "knsopub" in bundle:
  if not hknso:
    raise Exception("Module state certificate error: no HKNSO found")
  knsopub, _ = nfkm.unmarshal(parse_bytes(bundle["knsopub"]), nfkm.KeyData)
  hknso_bundle = choosealg.calchash(knsopub, nfkm.KeyHashMech("SHA1Hash"))
  if hknso_bundle != hknso:
      raise Exception("Module state certificate error: KNSO inconsistent with HKNSO")

# MSCV5 - If KM is present check it matches the module state cert
if "hkm" in bundle:
  hkm, _ = nfkm.unmarshal(parse_bytes(bundle["hkm"]), nfkm.KeyHashEx)
  if not hkm in hkms:
    raise Exception("Module state certificate error: KM not found")
```

4.5. World binding certificate verification (WBCV1-5)

As specified in Verification, the certificates must be constructed in a raw bytes format which given a known certname and ciphersuite, then verified against the signatures Cert*. At the end of this process, the keys in hashes can be trusted.

```
CERTIFICATES = {
  "CertKMaKMCbKNSO": ["Module keys", "hknso", "hkm", "hkmc"],
  "CertKMaKMCaKFIPSbKNSO": ["Module setup, FIPS3", "hknso", "hkm", "hkmc", "hkfips"],
  "CertKREaKRAbKNSO": ["Card Recovery", "hknso", "hkre", "hkra"],
}
CERTTAIL = {"DLf1024s160mDES3": None, "DLf1024s160mRijndael": "KM type Rijndael"}
KEYS = ["hkre", "hkra", "hkfips", "hkmc", "hkm"]

hashes = {"hknso": hknso}
for key in KEYS:
  if key in bundle:
    hashes[key], _ = nfkm.unmarshal(parse_bytes(bundle[key]), nfkm.KeyHashEx)

# WBCV1-3 - Check any security world certificates supplied
for certname in CERTIFICATES:
  if certname not in bundle:
    continue
  if "knsopub" not in bundle:
    raise Exception("World binding certificate error: no KNSO found in bundle")
  if "ciphersuite" not in bundle:
    raise Exception("World binding certificate error: no ciphersuite found in bundle")

  ciphersuite = bundle["ciphersuite"]
  template = CERTIFICATES[certname]
  cert = b""
  for element in template:
    if element == "Module keys" or element = "Module setup, FIPS3":
      certtail = CERTTAIL.get(ciphersuite, f"suite = {ciphersuite}")
      if certtail is None:
        value = element + "\0"
      else:
        if element == "Module keys":
          value = element + "; " + certtail + "\0"
        else:
          value = element + ": " + certtail + "\0"
      value = bytes(value, "ASCII")
    elif element[0] == "h":
      value = hashes[element].data.hash
    else:
      value = bytes(element + "\0", "ASCII")
    cert += value

  plain_text = nfkm.PlainText(["Bytes", nfkm.ByteBlock(cert, fromraw=1)])
  sig, _ = nfkm.unmarshal(certbytes, nfkm.CipherText)
  choosealg.verify(pubkey, sig.mech, plain_text, sig)

# WBCV4-5 - Remove keys without a suitable certification chain
uncertified = set()
if "CertKREaKRAbKNSO" not in bundle:
  uncertified.add("hkre")
  uncertified.add("hkra")
if "CertKMaKMCaKFIPSbKNSO" not in bundle:
  uncertified.add("hkfips")
if "CertKMaKMCaKFIPSbKNSO" not in bundle and "CertKMaKMCbKNSO" not in bundle:
  uncertified.add("hkm")
  uncertified.add("hkmc")
```

```
for key in uncertified:
  if key in hashes:
    del hashes[key]
```

4.6. Key generation certificate verification (KGCV1-2)

```
KML_MECHS = set(["ECDSAhSHA512", "DSAhSHA256"])

# KGCV1 - Verify the key generation certificate
kcmsg = parse_bytes(bundle["kcmsg"])
kcsig = parse_bytes(bundle["kcsig"])
kcsig, _ = nfkm.unmarshal(kcsig, nfkm.CipherText)
plain_text = nfkm.PlainText(["Bytes", kcmsg])
if kcsig.mech not in KML_MECHS:
  raise Exception(f"Key generation certificate error: unexpected KML signature mechanism")
choosealg.verify(kml, kcsig.mech, plain_text, kcsig)
kcmsg, _ = nfkm.unmarshal(kcmsg, nfkm.ModCertMsg)

# KGCV2 - Verify that the key generation certificate references the right key
pubkeydata = parse_bytes(bundle["pubkeydata"])
pubkeydata, _ = nfkm.unmarshal(pubkeydata, nfkm.KeyData)
pubkeyhash = choosealg.calchash(pubkeydata, nfkm.KeyHashMech("SHA1Hash")).data.hash
kcmsghash = kcmsg.data.hka
if pubkeyhash != kcmsghash:
  raise Exception("Key generation certificate error: key hash mismatch")
```

4.7. ACL validation (ACLV1,3-5)

The code below is split for readability.

```
acl = kcmsg.data.acl

protection = "unknown"
recovery = False
permissions = set()

for perm_group in acl:
  # ACLV1 - If any of the certifier fields matches hknso, mark the key as recoverable
  # and skip 'main use group' checks for this group
  if is_trump_ops_group(perm_group):
    recovery = True
    continue

  for action in perm_group.actions:
    if action.type == "OpPermissions":
      # ACLV3 - Require allowed permissions only
      permissions = oppermissions(action, permissions)
    elif action.type == "MakeBlob":
      protection = makeblob(action)
    elif action.type == "MakeArchiveBlob":
      makearchiveblob(action)
      recovery = True
    elif action.type in ['DeriveKey', 'DeriveKeyEx'] and action.details.mech == 'PublicFromPrivate':
      # DeriveMech_PublicFromPrivate is harmless
      pass
    else:
      # ACLV4 - disallow any actions other than those above
```

```
      raise Exception("ACL error: unacceptable permitted action")
```

4.7.1. Trump ops group

```
def is_trump_ops_group(perm_group):
  if "hknso" in hashes:
    if ((perm_group.flags & "certifier_present"
        and hashes["hknso"].mech == "SHA1Hash"
        and perm_group.certifier == hashes["hknso"].data.hash)
        or
        (perm_group.flags & "certmech_present"
        and hashes["hknso"].mech == "SHA1Hash"
        and perm_group.certmech.hash == hashes["hknso"].data.hash)
        or
        ('certmechex_present' in nfkm.PermissionGroup_flags.words
        and perm_group.flags & 'certmechex_present'
        and hashes["hknso"] == perm_group.certmechex.hash)):
      return True
  return False
```

4.7.2. Main use actions

4.7.2.1. Act_OpPermissions

The permissions are saved as direct categories (sign, verify, encrypt, decrypt, export and unwrap) rather than nCore permission types.

```
OPPERMISSIONS_MAP = {
  "UseAsCertificate": ["sign"],
  "ExportAsPlain": ["export"],
  "ExpandACL": ["export"],
  "Encrypt": ["encrypt"],
  "Decrypt": ["decrypt"],
  "Verify": ["verify"],
  "UseAsBlobKey": ["unwrap"],
  "UseAsKM": ["unwrap", "sign"],
  "Sign": ["sign"],
  "UseAsLoaderKey": ["decrypt"],
  "SignModuleCert": ["sign"],
}
# 'good' permissions can be limited further according to need
good_oppermissions = nfkm.Act_OpPermissions_Details_perms(
  ["DuplicateHandle", "GetAppData", "ReduceACL", "Sign", "GetACL",
   "Decrypt", "UseAsCertificate", "UseAsBlobKey", "SignModuleCert"]
)

def oppermissions(action, permissions):
  bad_oppermissions = action.details.perms & ~good_oppermissions
  if bad_oppermissions != 0:
    raise Exception("ACL error: unacceptable permissions: {bad_oppermissions}")

  # Record the permissions we've seen
  for perm in action.details.perms.words:
    if action.details.perms & perm:
      for permission in OPPERMISSIONS_MAP.get(perm, []):
        permissions.add(permission)
```

```
  return permissions
```

4.7.2.2. Act_MakeBlob (WB1-3,5-7)

This action determines the key's protection type. Module protected keys are expected to contain flags `AllowNonKM0|AllowKmOnly|kmhash_present`. Token protected keys are expected to contain flags `AllowNonKM0|kmhash_present|kthash_present`

```
def makeblob(action):
  protection = {
    "module": False,
    "softcard": False,
    "cardset": False
  }

  if (not (action.details.flags & "AllowKmOnly") and
      not (action.details.flags & "kthash_present")):
    # WB1 - key must be either module or token protected, not both or neither
    raise Exception("ACL error: makeblob action permits incoherent protection")
  elif not (action.details.flags & "kmhash_present"):
    # WB2 - kmhash must be present
    raise Exception("ACL error: makeblob action specifies no module key")
  elif hashes["hkm"].mech != "SHA1Hash" or action.details.kmhash != hashes["hkm"].data.hash:
    # WB2 - kmhash must match hkm
    raise Exception("ACL error: makeblob action permits inappropriate module key")
  elif action.details.flags & "AllowNullKmToken":
    # WB3 - AllowNullKMToken is not allowed
    raise Exception("ACL error: makeblob action permits null module key")

  # Get the protection type
  if action.details.flags & "AllowKmOnly":
    # WB5 - AllowKmOnly indicates a module-protected key
    protection["module"] = True
  elif action.details.flags & "kthash_present":
    # WB6 - If kthash is present, ktparams must be present
    if action.details.flags & "ktparams_present":
      # WB7 - AllowSoftSlots indicates a softcard-protected key
      if action.details.ktparams.flags & "AllowSoftSlots":
        protection["softcard"] = True
      else:
        protection["cardset"] = True
    else:
      raise Exception("ACL error: makeblob action permits any token type")

  # ACLV5 - Get protection type, using weakest if more than one is present
  if protection["module"]:
    prot = "module"
  elif protection["softcard"]:
    prot = "softcard"
  elif protection["cardset"]:
    prot = "cardset"

  return prot
```

4.7.2.3. Act_MakeArchiveBlob (RB1-3,5)

This allows for the creation of a recovery blob. The flag `kahash_present` is required and the

mechanism must be correct for the ciphersuite.

```
RECOVERY_MECH = {
  "DLf1024s160mDES3": "RSApPKCS1",
  "DLf1024s160mRijndael": "BlobCryptv2kRSAeRijndaelCBC0hSHA512mSHA512HMAC",
  "DLf3072s256mRijndael": "BlobCryptv2kRSAeRijndaelCBC0hSHA512mSHA512HMAC",
  "DLf3072s256mAEScSP800131Ar1": "BlobCryptv3kRSAOAEPeAESCBC0dCTRCMACmSHA512HMAC",
}

def makearchiveblob(action):
  if "hkre" not in hashes:
    # RB1 - hkre must be present and trusted
    raise Exception("ACL error: makearchiveblob but no hkre in bundle")
  elif not (action.details.flags & "kahash_present"):
    # RB2 - kahash must be present
    raise Exception("ACL error: makearchiveblob action specifies no blobbing key")
  elif hashes["hkre"].mech != "SHA1Hash" or action.details.kahash != hashes["hkre"].data.hash:
    # RB2 - kahash must match hkre
    raise Exception("ACL error: makearchiveblob action specifies unsuitable blobbing key")
  elif action.details.mech != RECOVERY_MECH[ciphersuite]:
    # RB3 - recovery mech must match ciphersuite
    raise Exception("ACL error: makearchiveblob action specifies unsuitable blobbing mechanism")
```

5. Example using nfkmattest

The following bundles were generated using `nfkmattest bundle` for a simple module-protected recoverable RSA key and a PKCS11 softcard-protected unrecoverable ECDSA key respectively.


```
{
  "pubkeydata": "AQAAAAQAAAABAAEAAAEAAEsjFSZ9vXsr-
Bibs6IYtQvtUu1gX9_higQFUOaL_nc1NVbFgetZP86Ora7NJ59ZOKwOe_P9whRDjGPLczbI28-W7ejDl5-
8itPxgXQIAzNi3JyBbJuW0t4bzMEkAktX4L115sD56p-p8NhF3Y32M7i3keA-
jmyf847dcWn5rxvBYmcef3Ftl75sSIoQSPxF3dU0A7u5QaUBBmHqQmLv5n7OJi1bNdkh1-
iP8hJ5xHrnfFbIav16s4XYyAzLwjbTlWvDDZu4RiUVQFwwlanrzhbJSFM9FAjArCo7Ioh87FhLCeSQITtxkvoDZ3LPr3uG3iqfsO1Rt3ZzChybeQT
pPq0=",
  "kcmsg":
"AgAAAAAAAACAAAABAAAAAAIAAAEAAAAAAAAAAAAAAABAAAAAQAAACu1AAAAAAAAAQAAAAEAAABEyjvlUqaq0yvjrRdSEN1UufjWNgEAAAABAAAA
AgAAAAcAAACYbc9zdHsnf6ttVH5cuRvQHMo99QMAAAAAAAAAAwAAAAEAAAB9IAAAAgAAACMAAAADAAAAAAAAAAAAAAACHgP1yOO6G7jsSRIbSIrWXq
qeiqAAAAAABAAAAAQAAABCUQoUPNMVkyUtPyFWBT020gk9JAQAAAAEAAAADAAAAAQAAAOEAAABRVdxAldukzldMXAZqiHNVRREyEDCdZFXAOmyl0R
qCERTy733lvnlV",
  "kcsig": "qgAAACAAAAAmbwm5HsjQzmMeQsNKqMUMZV7sJ-1xVqPWuMwUdv2xvyAAAAAjFGYBQnORYkCxHcExCEoE501cf-
gDFMto3F9SMNANkw==",
  "modstatemsg":
"BAAAAAAAAAAFAAAAAgAAAA8AAAA4OTM4LTEwNzUtODhCQgAAAwAAAEBtWoIpxbOidJC_uExZegepnB6nAwAAAIABAADjwRLJlKVAir-
HlVAUCWojKksMqGyWGhwhMoqYP8ldIy7bb3UVQBp6M-fxVpSFFrz3bfDgJQNh_13YCAY1-
r1JYvEner7cnGatDIjnMgNqQPN6alqM787pMz3_eIq0L0xI8rVy99F_foV6aFcJVCvxsjL9wIQOd4AhjIgTfPTiAEC4UTl5Eg9YkKnjZXizpTxhRe
SZVMjIM8Fu2sjcvzh1Q8POqYcEuU5sZhQbLVjUvRpou2HpgOTwhcXW-
X4gWpMlXsVkwV7F24j4Ax6eiyaSp1HCx4savMxcyA3cxwp7dTUJnFPVr8npfqp2H3ai3khSIefMc7d8gyajJnOLlJQMzf6O5HrlVeuixd6hBdwdB1
Ku9rirlixkgEd-73tMVJ1FQz85aCWuRqJl04YB1YwFvZgvRXhHvzqLfeJZAUerKlLgIaZwDq1twoXzvHq88QcJdbr0i4-
87VorPKkEjKtSSGHOVkkHhoBC8uNgYXnTBxqcqCqpZl4whuiEBmJQLcwgAAAAg8rgckmo3ArobecQooPxQ9AjYbCmAoKOUTRi7grTzPyAAQAA3Bvu
z-tQ1uh5LvuKMLTtGDTTplG7ks6ZkL8b-
F2UW37jfN3lap27oAZq1otU4FOP4EVvoMmNSdI4uzCPi7VgcI3AcIkdjZIwbpYf9XQwvFwMxYvdBPGhPtc_t8Lslgs97rMkES4ZciNI_NwjKp0fW4
kCiSBSUQUAUcp6vgqg2vVL9naqRHhXNRJuweaRtOO60z0mBkTgCnAvscdr2ymErrWDZArHosYXJZrXghjNmXvu-
rS8GvTvTc189gfRjMpL5aBYYAkl1XqWDGHhFcltxTSzCMgWalxMOeOQxaZLbzYDtl2_udXIo0bn_PTgaOkPYTypvzFwsQM4axpDzVYCE064YVoyjU
pgWBU4kMlC4JuH5ytJAM-uA67xu36Iqx3j_mjMozTR1rGJuH-
b314zgHRjvfr8AA0juiXn8tdxkFFRQlzYC9Ulw4g6fOSa06ecBIOA5Q0ylvdVjqmcX2--J-snC0wTxHV-
vLKWh8m7_DDjExTXKpHo5EqyQB24tHCogAEAAEQ7JI0sd9qudg-S1JDTNdoEl3MDsiBG-
RgomrJ0irL2PPtosXH5hUmOPY3VMZDXBKN_5f6AEOiz9DG98pRiAmORNUoFsp5vzuVMWjchgBeKvCG0uAm7SSif4lcoUB-WBNTqPEc_rexUtuCG-
YNhiysb2pCxH55Tu4Fj-PBREg5iXjGViusWxTm7mXhTHt8MZqEh4AFQma8aFT4BfCRWJGr1bb-JxzJ-
_ofzYn9JaTWIMPnG_Dp4MAbJZEyY3L19T032zDhNYixLlLfc1JbSJUvRhXGjP6akhKLkUrkNHkTwc_FF3Z73rjqqZpepmBYebgusP9ex77aYY6qZg
CVkgJqwnJyEOPEH0XJOOEl1jcft_T83m90SonSh_gP1EE-
w6m3l5Dbc7l09lpU_1Vs3CWE3IwArHSYFUB5Nrcm6KmN9GJYaoxCKDGetBgwnRPpjvWFAPAZa0nYcBHyLxaRF6O4PfT8ufV4QtqVKibJxWWoG2Bwy
92PPgeytW8qEwVFUEaoAAAANAAAAZI_xi8Yo8_d7sHmZXa9gb8bDxzMuAAAABgAAAAAAAABEAAAAhj4VZuTq0BfkEzWdVrE-owP29b-
6tuiMsEtErvqJtbKDfsVARAnHosqEo4pod90XGWnygseLmqxckfeUxopNiuwAAABEAAAAoaiQkrLgqV24LqicWwUVhicGnYAPREK8ZPBxjBtYAXb_
8mOLuO72S7TvzAxmnvfjKVIcEWAXQPAPoCawCOnORvQBAAC7AAAABQAAAIeA_XI47obuOxJEhtIitZeqp6KoT04AAAYAAAADAAAABNsAd2AIDhjP2
O5PriLiL-_3YT-JAAAASQAAAMK-mf4cd_G3XUji_S343_wMlpvLiQAAAEkAAACYbc9zdHsnf6ttVH5cuRvQHMo99YkAAABJAAAA",
  "modstatesig":
"uwAAAEQAAABIF582_Xr16pI3UgQnbJ6BVC_qkiHGmIqnidVnXo2eFxSl5iKKTRCaIbHzB9LADNKwNygbldB7Idbn6BUP4bpayQEAAEQAAACv8Z4k
RqTNrxXdiZTyyBDWRFVY9Tm0Qq2Pm6EHbSgZyCxjH_XpKVZHxlTVnNW75DE7PT_vMKgDgU5joCLtuzrUigAAAA==",
  "CertKMaKMCbKNSO": "qgAAACAAAADc8se4fR8nUa_sh_FEg2w77RsTV-
UhesIUzrz5tS22dCAAAABLeFUybWjTQAQtZo4p7eRSVuiGUlQD_wwMiuspHtprdg==",
  "CertKREaKRAbKNSO": "qgAAACAAAAAkCk4Ntz-DvicbSRSNitkewe6ztvMuKhsqs4nhkSgj0CAAAADl4sSRL7ajS-
E1vXi9lVyRhgszvYj_wISJx486Uae7Ow==",
  "hkre": "LAAAAFFV3ECV26TOV0xcBmqIc1VFETIQ",
  "hkra": "LAAAAAlOPU1SuiXSjIUbc2nSckda0AR5",
  "hkmc": "LAAAAMTxAuEBdgYihQ-qDMPpIJ4PLe-v",
  "hkm": "LAAAAJhtz3N0eyd_q21Ufly5G9Acyj31",
  "knsopub": "AwAAAIABAADjwRLJlKVAir-HlVAUCWojKksMqGyWGhwhMoqYP8ldIy7bb3UVQBp6M-fxVpSFFrz3bfDgJQNh_13YCAY1-
r1JYvEner7cnGatDIjnMgNqQPN6alqM787pMz3_eIq0L0xI8rVy99F_foV6aFcJVCvxsjL9wIQOd4AhjIgTfPTiAEC4UTl5Eg9YkKnjZXizpTxhRe
SZVMjIM8Fu2sjcvzh1Q8POqYcEuU5sZhQbLVjUvRpou2HpgOTwhcXW-
X4gWpMlXsVkwV7F24j4Ax6eiyaSp1HCx4savMxcyA3cxwp7dTUJnFPVr8npfqp2H3ai3khSIefMc7d8gyajJnOLlJQMzf6O5HrlVeuixd6hBdwdB1
Ku9rirlixkgEd-73tMVJ1FQz85aCWuRqJl04YB1YwFvZgvRXhHvzqLfeJZAUerKlLgIaZwDq1twoXzvHq88QcJdbr0i4-
87VorPKkEjKtSSGHOVkkHhoBC8uNgYXnTBxqcqCqpZl4whuiEBmJQLcwgAAAAg8rgckmo3ArobecQooPxQ9AjYbCmAoKOUTRi7grTzPyAAQAA3Bvu
z-tQ1uh5LvuKMLTtGDTTplG7ks6ZkL8b-
F2UW37jfN3lap27oAZq1otU4FOP4EVvoMmNSdI4uzCPi7VgcI3AcIkdjZIwbpYf9XQwvFwMxYvdBPGhPtc_t8Lslgs97rMkES4ZciNI_NwjKp0fW4
```

```
kCiSBSUQUAUcp6vgqg2vVL9naqRHhXNRJuweaRtOO60z0mBkTgCnAvscdr2ymErrWDZArHosYXJZrXghjNmXvu-
rS8GvTvTc189gfRjMpL5aBYYAkl1XqWDGHhFcltxTSzCMgWalxMOeOQxaZLbzYDtl2_udXIo0bn_PTgaOkPYTypvzFwsQM4axpDzVYCE064YVoyjU
pgWBU4kMlC4JuH5ytJAM-uA67xu36Iqx3j_mjMozTR1rGJuH-
b314zgHRjvfr8AA0juiXn8tdxkFFRQlzYC9Ulw4g6fOSa06ecBIOA5Q0ylvdVjqmcX2--J-snC0wTxHV-
vLKWh8m7_DDjExTXKpHo5EqyQB24tHCogAEAAJJO8O0taDvrOGdm28V2Q8WAli51mjCo6enCTRWts7Mw8_90wDfYuNw1iklct_NSl_gwuea2TWwFQ
8dUkFSZVBL7qGZXmMVejN7CslLbSi1KjjmksYvzidFFRe0tW1bujWLm5c4WCsS2hYk7SAvQgnyZ_EKN6u9ikzKxx3msnieajy_GNegrWd8Iu_VvN5
puMGKxVWHiEBNswJslS3YPTxR7OJ1rbd24f3QVSvlLm1uM9uXFsp-J7dSFf43rlyLxxvCwe-
CBZvS7yjUYktgia17jQq2OyTU_sbJuo9cxED1UJxKf0yYWZ6XI1H4JyQGxFrt8ptLK4tL1avn262WYNG7G6dh7cFpu0gOCHCod9fUGV4vQWpVVF7c
6SAMjAlIW73ijB2-1uwhPuFsbr0g2LYVClwT_jWzwy3-o4tXLRTUQm3JEKlKFMKe45CxI_k89y-
94mWXhhk8dybqnStab2Fr1gzOzk7DAq0Zy_ZfKuAIuDrkyqMIgWGkW9Tggj8mJEw==",
  "ciphersuite": "DLf3072s256mAEScSP800131Ar1",
  "warrant": "kzdLV0FSTi0xsjdQYXlsb2Fkxe-
zO0RlbGVnYXRlS2V5lDVFQ0RTQTZQdWJsaWM4TklTVFA1MjGS9MVCAURQaU1HaGS1ttXrV6vBnN4lgjICn1mYi133pTJtH9eANE-
e2OKvNKq5h_GBY7WhIFxo0lY7JgKrAWM-kLtRyx4F9MVCAdRsDT0XvewlhJMNx3AR46c0CYAYaBpYhr3q-
pUolLXgjy2OYl8sO68QiACPH9IKTzoX8LkFQAoQADdto8Ekr8fRN1NpZ01lY2iSNUVDRFNBkjVFTVNBMTZTSEE1MTLEFldhcnJhbnRDZXJ0aWZpY2
F0ZVR5cGU6RGVsZWdhdGlvbjlTaWduYXR1cmXFhAAhzfbe4fvrBZt6CcIv_KUNa9JqxIsa-
3yzepAiFlWJ67HzV5yAvtrMvAkwUh6nvWVmyLLIkpRFM-rjmuFfRhSyigDnsAk-BD_fOHdhWdoavVxWAu55nanTlRAi-
NQonn2PCn1V1YvfAWSa0Mog9kd9vptaeGm8_k5mX46p8FNqmaAW17I3UGF5bG9hZNUBb7bEFkVsZWN0cm9uaWNTZXJpYWxOdW1iZXIuODkzOC0xMD
c1LTg4QkLEFFBoeXNpY2FsU2VyaWFsTnVtYmVyKTM2LUozMTAyODlBcHByb3ZhbHORlDdGSVBTMTQwAgPEEU11bHRpQ2hpcEVtYmVkZGVkxBZXYXJ
yYW50Q2VydGlmaWNhdGVUeXBlxB1GaWVsZFVwZ3JhZGVNb2R1bGVJbmZvcm1hdGlvbjdLTEYycHVilDVFQ0RTQTZQdWJsaWM4TklTVFA1MjGS9MVB
7IpNisaU95FcrJqLx4LyaRkX3XdoiqOEyqLHCURAxX6DsrWJ-q5ES7CM6La6v_X2A6M-
sVadNRPkF9Dq5GYVPob0xUIB9EbO6QiwJqAP8EAXYBEcUinj955mDMzvtEv27riLY_L_dgFYG4xx8GS8QkQPgJ0GJ4YVBVucqC64XangspKQqKE4S
0xGMm1lY2iSNUVDRFNBkjVFTVNBMTZTSEE1MTI5U2lnbmF0dXJlxYQB7AIdL1WTvyfGKCfLYwXgmU1xoq8yTjUDLa4ix8cMWTWjfLST-
rKXSs0XpgDmmn0KtWGs9n3qTiNyPoxOiC2GvR0BHJCIdw0TgTtvDpx3e47NKhYxOr-
CzXUy10MhFnehsNCxB0PX7Xx6BWp_Sgzrqx8HCLwuZhp_N7rcyjK-ZaUUgC4=",
  "root": "KWARN-1"
}
```



```
{
  "pubkeydata": "LgAAAAQAAAAAAAAAIAAAAN3QzxvTTkl3yEbZnNOcEk5XJg202sSXCDBITO3v0GLfIAAAACDMg-
bQtYAqpiT5Aoo9y2FR5lpUaYfD6bqJVOp3Ao6d",
  "kcmsg":
"AgAAAAAAAAAvAAAABAAAAAIAAAAAAAAAAAAAAAAEAAAABAAAAK7AAAAAAAAABAAAAAQAAAPNtt37lSiycKsctrhVUTNxMsr63AQAAAAEAAAACAAAA
HgAAAJhtz3N0eyd_q21Ufly5G9Acyj310YYwMYdXvvcfvD_o9BQliSmAJbsHAAAAAQAAAAEAAAAAAAAAAUju-CnMQXoZ1fYPHV4Xj3XZpnaA=",
  "kcsig": "qgAAACAAAAD6isCPSExgF4woKUP2lyJ7F7nhjBOHWZOUfIuzWkutsCAAAADO0B84j8KWl2oS9X9GusDQkNHHjxMni-
ChHwHTmpV4Qw==",
  "modstatemsg":
"BAAAAAAAAAAFAAAAAgAAAA8AAAA4OTM4LTEwNzUtODhCQgAAAwAAAEBtWoIpxbOidJC_uExZegepnB6nAwAAAIABAADjwRLJlKVAir-
HlVAUCWojKksMqGyWGhwhMoqYP8ldIy7bb3UVQBp6M-fxVpSFFrz3bfDgJQNh_13YCAY1-
r1JYvEner7cnGatDIjnMgNqQPN6alqM787pMz3_eIq0L0xI8rVy99F_foV6aFcJVCvxsjL9wIQOd4AhjIgTfPTiAEC4UTl5Eg9YkKnjZXizpTxhRe
SZVMjIM8Fu2sjcvzh1Q8POqYcEuU5sZhQbLVjUvRpou2HpgOTwhcXW-
X4gWpMlXsVkwV7F24j4Ax6eiyaSp1HCx4savMxcyA3cxwp7dTUJnFPVr8npfqp2H3ai3khSIefMc7d8gyajJnOLlJQMzf6O5HrlVeuixd6hBdwdB1
Ku9rirlixkgEd-73tMVJ1FQz85aCWuRqJl04YB1YwFvZgvRXhHvzqLFeJZAUerKlLgIaZwDq1twoXzvHq88QcJdbr0i4-
87VorPKkEjKtSSGHOVkkHhoBC8uNgYXnTBxqcqCqpZl4whuiEBmJQLcwgAAAAg8rgckmo3ArobecQooPxQ9AjYbCmAoKOUTRi7grTzPyAAQAA3Bvu
z-tQ1uh5LvuKMLTtGDTTplG7ks6ZkL8b-
F2UW37jfN3lap27oAZq1otU4FOP4EVvoMmNSdI4uzCPi7VgcI3AcIkdjZIwbpYf9XQwvFwMxYvdBPGhPtc_t8Lslgs97rMkES4ZciNI_NwjKp0fW4
kCiSBSUQUAUcp6vgqg2vVL9naqRHhXNRJuweaRtOO60z0mBkTgCnAvscdr2ymErrWDZArHosYXJZrXghjNmXvu-
rS8GvTvTc189gfRjMpL5aBYYAkl1XqWDGHhFcltxTSzCMgWalxMOeOQxaZLbzYDtl2_udXIo0bn_PTgaOkPYTypvzFwsQM4axpDzVYCE064YVoyjU
pgWBU4kMlC4JuH5ytJAM-uA67xu36Iqx3j_mjMozTR1rGJuH-
b314zgHRjvfr8AA0juiXn8tdxkFFRQlzYC9Ulw4g6fOSa06ecBIOA5Q0ylvdVjqmcX2--J-snC0wTxHV-
vLKWh8m7_DDjExTXKpHo5EqyQB24tHCogAEAAEQ7JI0sd9qudg-S1JDTNdoEl3MDsiBG-
RgomrJ0irL2PPtosXH5hUmOPY3VMZDXBKN_5f6AEOiz9DG98pRiAmORNUoFsp5vzuVMWjchgBeKvCG0uAm7SSif4lcoUB-WBNTqPEc_rexUtuCG-
YNhiysb2pCxH55Tu4Fj-PBREg5iXjGViusWxTm7mXhTHt8MZqEh4AFQma8aFT4BfCRWJGr1bb-JxzJ-
_ofzYn9JaTWIMPnG_Dp4MAbJZEyY3L19T032zDhNYixLlLfc1JbSJUvRhXGjP6akhKLkUrkNHkTwc_FF3Z73rjqqZpepmBYebgusP9ex77aYY6qZg
CVkgJqwnJyEOPEH0XJOOEl1jcft_T83m90SonSh_gP1EE-
w6m3l5Dbc7l09lpU_1Vs3CWE3IwArHSYFUB5Nrcm6KmN9GJYaoxCKDGetBgwnRPpjvWFAPAZa0nYcBHyLxaRF6O4PfT8ufV4QtqVKibJxWWoG2Bwy
92PPgeytW8qEwVFUEaoAAAANAAAAZI_xi8Yo8_d7sHmZXa9gb8bDxzMuAAAABgAAAAAAAABEAAAAhj4VZuTq0BfkEzWdVrE-owP29b-
6tuiMsEtErvqJtbKDfsVARAnHosqEo4pod90XGWnygseLmqxckfeUxopNiuwAAABEAAAAoaiQkrLgqV24LqicWwUVhicGnYAPREK8ZPBxjBtYAXb_
8mOLuO72S7TvzAxmnvfjKVIcEWAXQPAPoCawCOnORvQBAAC7AAAABQAAAIeA_XI47obuOxJEhtIitZeqp6KoT04AAAYAAAADAAAABNsAd2AIDhjP2
O5PriLiL-_3YT-JAAAASQAAAMK-mf4cd_G3XUji_S343_wMlpvLiQAAAEkAAACYbc9zdHsnf6ttVH5cuRvQHMo99YkAAABJAAAA",
  "modstatesig":
"uwAAAEQAAABIF582_Xr16pI3UgQnbJ6BVC_qkiHGmIqnidVnXo2eFxSl5iKKTRCaIbHzB9LADNKwNygbldB7Idbn6BUP4bpayQEAAEQAAACv8Z4k
RqTNrxXdiZTyyBDWRFVY9Tm0Qq2Pm6EHbSgZyCxjH_XpKVZHxlTVnNW75DE7PT_vMKgDgU5joCLtuzrUigAAAA==",
  "CertKMaKMCbKNSO": "qgAAACAAAADc8se4fR8nUa_sh_FEg2w77RsTV-
UhesIUzrz5tS22dCAAAABLeFUybWjTQAQtZo4p7eRSVuiGUlQD_wwMiuspHtprdg==",
```

```
  "CertKREaKRAbKNSO": "qgAAACAAAAAkCk4Ntz-DvicbSRSNitkewe6ztvMuKhsqs4nhkSgj0CAAAADl4sSRL7ajS-
E1vXi9lVyRhgszvYj_wISJx486Uae7Ow==",
  "hkre": "LAAAAFFV3ECV26TOV0xcBmqIc1VFETIQ",
  "hkra": "LAAAAAlOPU1SuiXSjIUbc2nSckda0AR5",
  "hkmc": "LAAAAMTxAuEBdgYihQ-qDMPpIJ4PLe-v",
  "hkm": "LAAAAJhtz3N0eyd_q21Ufly5G9Acyj31",
  "knsopub": "AwAAAIABAADjwRLJlKVAir-HlVAUCWojKksMqGyWGhwhMoqYP8ldIy7bb3UVQBp6M-fxVpSFFrz3bfDgJQNh_13YCAY1-
r1JYvEner7cnGatDIjnMgNqQPN6alqM787pMz3_eIq0L0xI8rVy99F_foV6aFcJVCvxsjL9wIQOd4AhjIgTfPTiAEC4UTl5Eg9YkKnjZXizpTxhRe
SZVMjIM8Fu2sjcvzh1Q8POqYcEuU5sZhQbLVjUvRpou2HpgOTwhcXW-
X4gWpMlXsVkwV7F24j4Ax6eiyaSp1HCx4savMxcyA3cxwp7dTUJnFPVr8npfqp2H3ai3khSIefMc7d8gyajJnOLlJQMzf6O5HrlVeuixd6hBdwdB1
Ku9rirlixkgEd-73tMVJ1FQz85aCWuRqJl04YB1YwFvZgvRXhHvzqLFeJZAUerKlLgIaZwDq1twoXzvHq88QcJdbr0i4-
87VorPKkEjKtSSGHOVkkHhoBC8uNgYXnTBxqcqCqpZl4whuiEBmJQLcwgAAAAg8rgckmo3ArobecQooPxQ9AjYbCmAoKOUTRi7grTzPyAAQAA3Bvu
z-tQ1uh5LvuKMLTtGDTTplG7ks6ZkL8b-
F2UW37jfN3lap27oAZq1otU4FOP4EVvoMmNSdI4uzCPi7VgcI3AcIkdjZIwbpYf9XQwvFwMxYvdBPGhPtc_t8Lslgs97rMkES4ZciNI_NwjKp0fW4
kCiSBSUQUAUcp6vgqg2vVL9naqRHhXNRJuweaRtOO60z0mBkTgCnAvscdr2ymErrWDZArHosYXJZrXghjNmXvu-
rS8GvTvTc189gfRjMpL5aBYYAkl1XqWDGHhFcltxTSzCMgWalxMOeOQxaZLbzYDtl2_udXIo0bn_PTgaOkPYTypvzFwsQM4axpDzVYCE064YVoyjU
pgWBU4kMlC4JuH5ytJAM-uA67xu36Iqx3j_mjMozTR1rGJuH-
b314zgHRjvfr8AA0juiXn8tdxkFFRQlzYC9Ulw4g6fOSa06ecBIOA5Q0ylvdVjqmcX2--J-snC0wTxHV-
vLKWh8m7_DDjExTXKpHo5EqyQB24tHCogAEAAJJO8O0taDvrOGdm28V2Q8WAli51mjCo6enCTRWts7Mw8_90wDfYuNw1iklct_NSl_gwuea2TWwFQ
8dUkFSZVBL7qGZXmMVejN7CslLbSi1KjjmksYvzidFFRe0tW1bujWLm5c4WCsS2hYk7SAvQgnyZ_EKN6u9ikzKxx3msnieajy_GNegrWd8Iu_VvN5
puMGKxVWHiEBNswJslS3YPTxR7OJ1rbd24f3QVSvlLm1uM9uXFsp-J7dSFf43rlyLxxvCwe-
CBZvS7yjUYktgia17jQq2OyTU_sbJuo9cxED1UJxKf0yYWZ6XI1H4JyQGxFrt8ptLK4tL1avn262WYNG7G6dh7cFpu0gOCHCod9fUGV4vQWpVVF7c
6SAMjAlIW73ijB2-1uwhPuFsbr0g2LYVClwT_jWzwy3-o4tXLRTUQm3JEKlKFMKe45CxI_k89y-
94mWXhhk8dybqnStab2Fr1gzOzk7DAq0Zy_ZfKuAIuDrkyqMIgWGkW9Tggj8mJEw==",
  "ciphersuite": "DLf3072s256mAEScSP800131Ar1",
  "warrant": "kzdLV0FSTi0xsjdQYXlsb2Fkxe-
zO0RlbGVnYXRlS2V5lDVFQ0RTQTZQdWJsaWM4TklTVFA1MjGS9MVCAURQaU1HaGS1ttXrV6vBnN4lgjICn1mYi133pTJtH9eANE-
e2OKvNKq5h_GBY7WhIFxo0lY7JgKrAWM-kLtRyx4F9MVCAdRsDT0XvewlhJMNx3AR46c0CYAYaBpYhr3q-
pUolLXgjy2OYl8sO68QiACPH9IKTzoX8LkFQAoQADdto8Ekr8fRN1NpZ01lY2iSNUVDRFNBkjVFTVNBMTZTSEE1MTLEFldhcnJhbnRDZXJ0aWZpY2
F0ZVR5cGU6RGVsZWdhdGlvbjlTaWduYXR1cmXFhAAhzfbe4fvrBZt6CcIv_KUNa9JqxIsa-
3yzepAiFlWJ67HzV5yAvtrMvAkwUh6nvWVmyLLIkpRFM-rjmuFfRhSyigDnsAk-BD_fOHdhWdoavVxWAu55nanTlRAi-
NQonn2PCn1V1YvfAWSa0Mog9kd9vptaeGm8_k5mX46p8FNqmaAW17I3UGF5bG9hZNUBb7bEFkVsZWN0cm9uaWNTZXJpYWxOdW1iZXIuODkzOC0xMD
c1LTg4QkLEFFBoeXNpY2FsU2VyaWFsTnVtYmVyKTM2LUozMTAyODlBcHByb3ZhbHORlDdGSVBTMTQwAgPEEU11bHRpQ2hpcEVtYmVkZGVkxBZXYXJ
yYW50Q2VydGlmaWNhdGVUeXBlxB1GaWVsZFVwZ3JhZGVNb2R1bGVJbmZvcm1hdGlvbjdLTEYycHVilDVFQ0RTQTZQdWJsaWM4TklTVFA1MjGS9MVB
7IpNisaU95FcrJqLx4LyaRkX3XdoiqOEyqLHCURAxX6DsrWJ-q5ES7CM6La6v_X2A6M-
sVadNRPkF9Dq5GYVPob0xUIB9EbO6QiwJqAP8EAXYBEcUinj955mDMzvtEv27riLY_L_dgFYG4xx8GS8QkQPgJ0GJ4YVBVucqC64XangspKQqKE4S
0xGMm1lY2iSNUVDRFNBkjVFTVNBMTZTSEE1MTI5U2lnbmF0dXJlxYQB7AIdL1WTvyfGKCfLYwXgmU1xoq8yTjUDLa4ix8cMWTWjfLST-
rKXSs0XpgDmmn0KtWGs9n3qTiNyPoxOiC2GvR0BHJCIdw0TgTtvDpx3e47NKhYxOr-
CzXUy10MhFnehsNCxB0PX7Xx6BWp_Sgzrqx8HCLwuZhp_N7rcyjK-ZaUUgC4=",
  "root": "KWARN-1"
}
```


They can be verified using `nfkmattest verify` and the results are summarized below.


```
{
  "path": "key_simple_test1.att",
  "protection": "module",
  "recovery": true,
  "type": "RSAPublic",
  "permissions": [
    "unwrap",
    "sign",
    "decrypt"
  ],
  "esn": "8938-1075-88BB",
  "hknso": "8780fd72 38ee86ee 3b124486 d222b597 aaa7a2a8",
  "k": {
    "type": "RSAPublic",
    "data": {
      "e": "AQAB",
      "n":
"rT7pBHmbHApzdrdR7bCfKt6Ge6/PcmcD+pJxOyGQ5AlLWOx8iCI7KqzACBQ9U0jJFs7rqZUwXEAVJUa4mw3Da5XTNsLLDMjYhbN6/WrIVnznesR5
EvKP6Nch2TVbLSbOfubvYkLqYQYBpUG5uwM01d1F/EgQikhsvpdtcX8eZ2LBG6/5aXHdjvOfbI4+4JG3uDP2jd1F2PCpn+r5wOZ1veBXSwIkwcwb3
tKWm2yBnNxiMwMIdIHx04q8n5fD6O2Wz9vINnPLY4xDFML983sOrDhZnyfNrq2Ozj9Z64HFVjU1d/6L5lAFBIrh319g7VLtC7UYorObGPgre719Jh
UjSw=="
```

```
    }
  }
}
```


```
{
  "path": "key_pkcs11_test2.att",
  "protection": "softcard",
  "recovery": false,
  "type": "ECDSAPublic",
  "permissions": [
    "sign"
  ],
  "esn": "8938-1075-88BB",
  "hknso": "8780fd72 38ee86ee 3b124486 d222b597 aaa7a2a8",
  "k": {
    "type": "ECDSAPublic",
    "data": {
      "curve": {
        "name": "NISTP256"
      },
      "Q": {
        "flags": [],
        "x": "32LQ7+1MSDAIl8TatA0mV04SnNOc2UbId0lO0xvP0N0=",
        "y": "nY4Cd+pUibrpw4dpVFrmUWHLPYoC+SSmKoC10OaDzCA="
      }
    }
  }
}
```