Named Data Networking Security

Developer's Guide

Zhiyi Zhang
Alexander Afanasyev
Yingdi Yu

Revision history

- **Revision 1 (Month ??, 201?)**: Initial release

This book describes how NDN libraries and tools achieve NDN security and how to use these libraries and tools.

NAMED DATA NETWORKING (NDN) PROJECT

HTTP://NAMED-DATA.NET

Month Year

Contents

1. NDN Security Overview

The Named Data Networking (NDN) architecture builds packet level security into the network layer. NDN security refers to the security related consideration and designs in the NDN. NDN security including security tools and libraries provides application developers with reliable mechanisms to build up the security for new protocol features, algorithms and applications for NDN. To help developers get better understand and improve the NDN security, this document explains all the security designs in NDN and also the future plan to make NDN security stronger.

NDN security including security tools and libraries provides application developers with reliable mechanisms to build up the security for new protocol features, algorithms and applications for NDN. The NDN security mainly involves two parts. The first part is **Signing and authentication**, which includes the packet format design and identity management. This part is realized by ndn-cxx library. The other part are security protocols which are designed taking use of NDN architecture.

1.1 Signing and authentication

In TCP/IP, it is endpoints' responsibility for security. In contrast, NDN secures the packets directly from the network layer by requiring the data producer to sign very single data packet. To sign packets on producer side and authenticate the signature on consumer side properly, in NDN security, there is a key management system which involves three important parts.

- Key
- Certificate
- Identity

Identity can be considered as a namespace which maps a entity in NDN. Identity may have one or more **keys** under identity's namespace. For each key, there may be one or more **certificates** associated. More library using details will be illustrated in section 2.2.

1.2 Security Protocols

Because the security is built into the packet level directly in network layer in NDN, many security designs and systems can benefit from the change. In NDN, there are a few protocols now or in plan:

- Name-based Access Control Protocol.
- NDN Certificate Management Protocol.
- Key Bundle Protocol.
- DeLorean Protocol.

In this document, we will first introduce the existing security libraries which includes ndn-cxx security library, name-based access control library, and certificate management library. After that, the existing security tools will be introduced. Projects that are in progress will be covered afterwards. Moreover, the future plan of NDN security is showed in the very last section.

2. ndn-cxx Security Library

In NDN, all the data packets are supposed to have a signature. The signature will ensure the integrity and enable determination of data provenance, allowing a consumer's trust in data to be decoupled from how or where it is obtained.

Regarding interest packet, cryptographically signed signature is not necessary for most cases. However, sometimes interest packet could also bring sender's authority. E.g., a command interest. In such cases, an interest packet also need to be signed.

2.1 Signature

2.1.1 Data packet signature

NDN Signature is defined as two consecutive TLV blocks: `SignatureInfo` and `SignatureValue`. The following general considerations about SignatureInfo and SignatureValue blocks that apply for all signature types:

- `SignatureInfo` is **included** in signature calculation and fully describes the signature, signature algorithm, and any other relevant information to obtain parent certificate(s), such as KeyLocator.
- `SignatureValue` is **excluded** from signature calculation and represent actual bits of the signature and any other supporting signature material.

The reason for separating the signature into two separate TLV blocks is to allow efficient signing of a contiguous memory block (e.g., for Data packet this block starts from Name TLV and ends with SignatureInfo TLV).

```
Signature ::= SignatureInfo
              SignatureValue

SignatureInfo ::= SIGNATURE-INFO-TYPE TLV-LENGTH
                    SignatureType
                    KeyLocator?
```

```
                    ... (SignatureType-specific TLVs)

SignatureValue ::= SIGNATURE-VALUE-TYPE TLV-LENGTH
                    ... (SignatureType-specific TLVs and BYTE+)
```

SignatureType

```
SignatureType ::= SIGNATURE-TYPE-TYPE TLV-LENGTH
                    nonNegativeInteger
```

This specification defines the following SignatureType values:

Table 2.1: SignatureType Value

Value	Reference	Description
0	2.1.3	Integrity protection using SHA-256 digest
1	2.1.3	Integrity and provenance protection using RSA signature over a SHA-256 digest
3	2.1.3	Integrity and provenance protection using an ECDSA signature over a SHA-256 digest
2, 5-200		reserved for future assignments
>200		unassigned

KeyLocator

A `KeyLocator` specifies either `Name` that points to another Data packet containing certificate or public key or `KeyDigest` to identify the public key within a specific trust model (the trust model definition is outside the scope of the current specification). Note that although `KeyLocator` is defined as an optional field in `SignatureInfo` block, some signature types may require presence of it and some require `KeyLocator` absence.

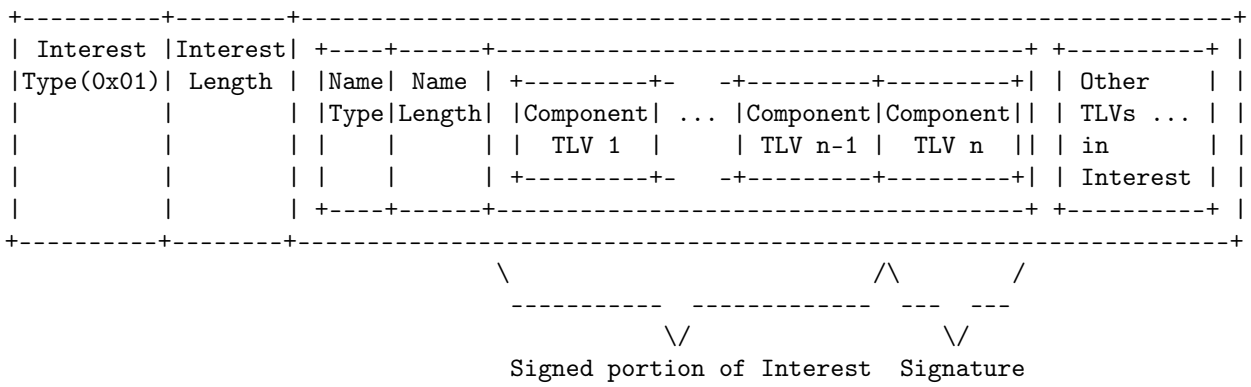
```
KeyLocator ::= KEY-LOCATOR-TYPE TLV-LENGTH (Name | KeyDigest)

KeyDigest ::= KEY-DIGEST-TYPE TLV-LENGTH BYTE+
```

The specific definition of the usage of `Name` and `KeyDigest` options in `KeyLocator` field is outside the scope of this specification. Generally, `Name` names the Data packet with the corresponding certificate. However, it is up to the specific trust model to define whether this name is a full name of the Data packet or a prefix that can match multiple Data packets. For example, the hierarchical trust model testbed-key-management uses the latter approach, requiring clients to fetch the latest version of the Data packet pointed by the KeyLocator (the latest version of the public key certificate) in order to ensure that the public key was not yet revoked.

2.1.2 Interest packet signature

Signed Interest is a mechanism to issue an authenticated interest.

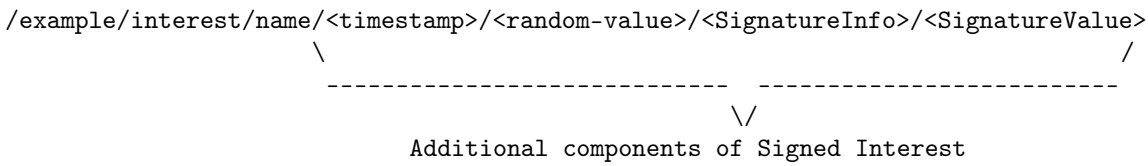
The signature of a signed Interest packet is embedded into the last component of the Interest name. The signature covers a continuous block starting from the first name component TLV to the penultimate name component TLV:



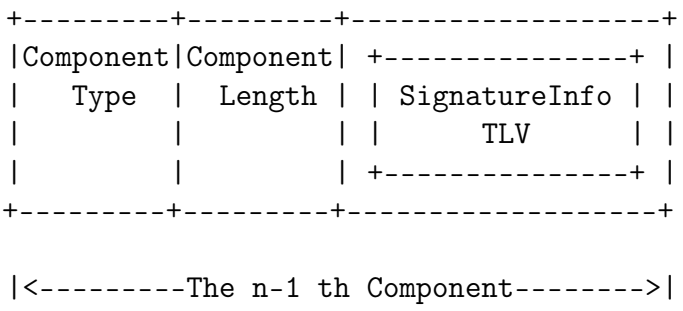
More specifically, the SignedInterest is defined to have four additional components:

- `<timestamp>`
- `<nonce>`
- `<SignatureInfo>`
- `<SignatureValue>`

For example, for "`/example/interest/name`" name, CommandInterest will be defined as:

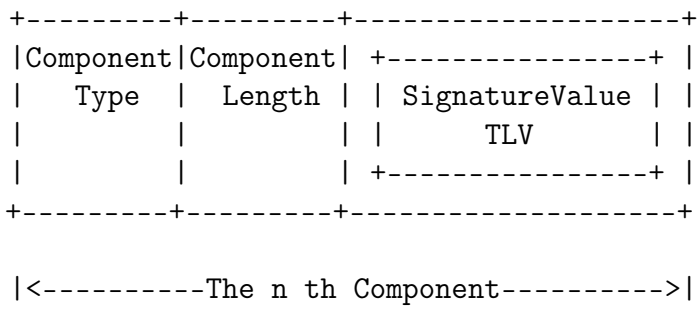


- Timestamp component (n-3 th)
  The value of the n-3 th component is the interest's timestamp (in terms of millisecond offset from UTC 1970-01-01 00:00:00) encoded as nonNegativeInteger. The timestamp may be used to protect against replay attack.
- Nonce component (n-2 th)
  The value of the n-2 th component is random value (encoded as nonNegativeInteger) that adds additional assurances that the interest will be unique.
- SignatureInfo component (n-1 th)
  The value of the n-1 th component is actually a SignatureInfo TLV which is the same as Data packet SignatureInfo.



- SignatureValue component (n th)

  The value of the n th component is actually a SignatureValue TLV which is the same as Data packet SignatureValue.



2.1.3 Supported Signature Type

SignatureSha256WithRsa

`SignatureSha256WithRsa` is the basic signature algorithm that MUST be supported by any NDN-compliant software. As suggested by the name, it defines an RSA public key signature that is calculated over SHA256 hash of the Name, MetaInfo, Content, and SignatureInfo TLVs.

```
SignatureInfo ::= SIGNATURE-INFO-TYPE TLV-LENGTH
                    SIGNATURE-TYPE-TYPE TLV-LENGTH(=1) 1
                    KeyLocator

SignatureValue ::= SIGNATURE-VALUE-TYPE TLV-LENGTH
                     BYTE+(=RSA over SHA256{Name, MetaInfo, Content, SignatureInfo})
```

Note: SignatureValue size varies (typically 128 or 256 bytes) depending on the private key length used during the signing process.

This type of signature ensures strict provenance of a Data packet, provided that the signature verifies and signature issuer is authorized to sign the Data packet. The signature issuer is identified using KeyLocator block in SignatureInfo block of `SignatureSha256WithRsa`. KeyDigest option in `KeyLocator` is defined as SHA256 digest over the DER encoding of the SubjectPublicKeyInfo for an RSA key as defined by <RFC 3279>.

Note: It is application's responsibility to define rules (trust model) of when a specific issuer (KeyLocator) is authorized to sign a specific Data packet. While trust model is outside the scope of the current specification, generally, trust model needs to specify authorization rules between KeyName and Data packet Name, as well as clearly define trust anchor(s). For example, an application can elect to use hierarchical trust model :cite:'testbed-key-management' to ensure Data integrity and provenance.

SignatureSha256WithEcdsa

`SignatureSha256WithEcdsa` defines an ECDSA public key signature that is calculated over the SHA256 hash of the Name, MetaInfo, Content, and SignatureInfoTLVs. The signature algorithm is defined in <Section 2.1 in RFC5753>.

```
SignatureInfo ::= SIGNATURE-INFO-TYPE TLV-LENGTH
                    SIGNATURE-TYPE-TYPE TLV-LENGTH(=1) 3
                    KeyLocator
```

```
SignatureValue ::= SIGNATURE-VALUE-TYPE TLV-LENGTH
                     BYTE+(=ECDSA over SHA256{Name, MetaInfo, Content, SignatureInfo})
```

Note: The SignatureValue size depends on the private key length used during the signing process (about 63 bytes for a 224 bit key).

This type of signature ensures strict provenance of a Data packet, provided that the signature verifies and the signature issuer is authorized to sign the Data packet. The signature issuer is identified using the KeyLocator block in the SignatureInfo block of the `SignatureSha256WithEcdsa`. KeyDigest option in `KeyLocator` is defined as SHA256 digest over the DER encoding of the SubjectPublicKeyInfo for an EC key as defined by <RFC 5480>.

The value of `SignatureValue` of `SignatureSha256WithEcdsa` is a DER encoded DSA signature as defined in <Section 2.2.3 in RFC3279>.

```
Ecdsa-Sig-Value  ::=  SEQUENCE  {
     r     INTEGER,
     s     INTEGER  }
```

DigestSha256

`DigestSha256` provides no provenance of a Data packet or any kind of guarantee that packet is from the original source. This signature type is intended only for debug purposes and limited circumstances when it is necessary to protect only against unexpected modification during the transmission.

`DigestSha256` is defined as a SHA256 hash of the Name, MetaInfo, Content, and SignatureInfo TLVs:

```
SignatureInfo ::= SIGNATURE-INFO-TYPE TLV-LENGTH(=3)
                    SIGNATURE-TYPE-TYPE TLV-LENGTH(=1) 0

SignatureValue ::= SIGNATURE-VALUE-TYPE TLV-LENGTH(=32)
                     BYTE+(=SHA256{Name, MetaInfo, Content, SignatureInfo})
```

Note that `SignatureInfo` does not require `KeyLocator` field, since there digest calculation and verification does not require any additional information. If `KeyLocator` is present in `SignatureInfo`, it must be ignored.

2.2 Identity, Key and Certificate

All keys, certificates and their corresponding identities are managed by `KeyChain`. There is an hierarchical structure of keys, certificates and identities; In real world, a user may have multiple identities. Each identity contains one or more keys, one of which is set as the default key of the identity. Similarly, each key contains one or more certificates, one of which is set as the default certificate of the key.

The private part which includes symmetric keys, and private keys of the asymmetric key pairs, is stored in a `Trusted Platform Module (TPM) <SecTpm>` in ndn-cxx security library. The public part which includes public keys of the asymmetric key pairs, identities, and certificates are managed in the `Public-key Information Base (PIB) <SecPublicInfo>` in ndn-cxx. The most important information managed by PIB is **certificates** of public keys.

2.2.1 Identity

An real world **identity** can be expressed by a namespace.
(e.g., "/ndn/edu/ucla/cs/zhiyi", or "/ndn/edu/ucla/BoelterHall/4805").

Create new identities

To create an identity with a default Key and Certificate, one can directly call the KeyChain::createIdentity function. The default key will be a ECDSA key with a random key id.

```
// create a new identity with the name /example
KeyChain keyChain;
Identity identity = keyChain.createIdentity(Name("/example"));
```

To create an identity with other kinds of key, one can use the KeyParams to configure the key type.

```
// create a new identity with the name /example
// the default key would be RSA key
KeyChain keyChain;
Identity identity = keyChain.createIdentity(Name("/example"), RsaKeyParams);

// create a new identity with the name /example
// the default key would be AES key
KeyChain keyChain;
Identity identity = keyChain.createIdentity(Name("/example"), AesKeyParams);
```

The developer can also change the KeyId and the length of the key bits by changing the KeyParams. More detail about KeyParams is in 2.2.2.

Delete identities

To delete an identity and all it's keys and certificates, one can call KeyChain::deleteIdentity function.

```
// delete an existing identity with the name /example
// first create a new identity
KeyChain keyChain;
Identity identity = keyChain.createIdentity(Name("/example"), AesKeyParams);

// delete the identity
keyChain.deleteIdentity(identity);
```

Set default identity

To set one specific identity as the default identity, one cal call KeyChain::setDefaultIdentity function.

```
// set an existing identity to be default
// first create a new identity
KeyChain keyChain;
Identity identity = keyChain.createIdentity(Name("/example"), AesKeyParams);

// delete the identity
keyChain.setDefaultIdentity(identity);
```

2.2.2 Key

When talking about a Key in the context of ndn-cxx, a key refers to a symmetric key or the public key of an asymmetric key pair. Regarding the asymmetric key, algorithm RSA and ECDSA are supported. Regarding the symmetric key, algorithm AES is supported.

Keys belonging to an identity are named under the identity's namespace, with a unique **KeyId**

```
/<identity_name>/KEY/[KeyId]
```

The KeyId is used to identify the key.

Create new keys

When creating a key, there are three types of KeyId could be set. Developer could decide which KeyId to be used by setting the KeyParams. KeyParams is one parameter of the key creating interface.

Also, if KeyParams parameter is not specified, the default value would be used. As mentioned in last section, the new key will be a ECDSA key with a random key id.

- user specified KeyId
  The example code is like:

```
// create a new RSA key for identity01
KeyChain keyChain;
RsaKeyParams params1(name::Component::fromNumber(123));
Key key = keyChain.createKey(identity01, params1);
```

- hash value of the key bits
  The example code is like:

```
// create a new RSA key for identity02
KeyChain keyChain;
RsaKeyParams params2(1024, KeyIdType::SHA256);
Key key = keyChain.createKey(identity02, params2);
```

- randomly generated KeyId
  This kind of KeyId is the default choice of KeyParams. The example code is like:

```
// create a new RSA key for identity03
KeyChain keyChain;
RsaKeyParams params3;
Key key = keyChain.createKey(identity02, params3);
```

Delete keys

To delete an existing key from an identity, one can call `KeyChain::deleteKey` function.

```
// delete a key for identity
// first create a new key
KeyChain keyChain;
Key key = keyChain.createKey(identity);

// delete the key
keyChain.deleteKey(identity, key);
```

Set default key

To set one specific key as the default key for the identity, one cal call `KeyChain::setDefaultKey` function.

```
// set an existing key to be default
// first create a new key
KeyChain keyChain;
Key key = keyChain.createKey(identity);

// set the default key
keyChain.setDefaultKey(identity, key);
```

2.2.3 Certificate

Certificate Format

A certificate binds a public key to its key name or the corresponding identity. The signer (or issuer) of a certificate vouches for the binding through its own signature. With different signers vouching for the binding, a public key may have more than one certificates.

Since signature verification is a common operation in NDN applications, it is important to define a common certificate format to standardize the public key authentication procedure. As every NDN data packet is signed, a data packet that carries a public key as content is conceptually a certificate. However, the specification of a data packet is not sufficient to be the specification of a common certificate format, as it requires additional components. For example, a certificate may follow a specific naming convention and may need to include validity period, revocation information, etc. This specification defines naming and structure of the NDN certificates and is complementary to NDN packet specification.

```
Overview of NDN certificate format
   +-------------------------+
   |           Name          |
   +-------------------------+
   |         MetaInfo        |
   |+-----------------------+|
   || ContentType:  KEY(2)  ||
   |+-----------------------+|
   |+-----------------------+|
   || FreshnessPeriod: >~ 1h ||
   |+-----------------------+|
   +-------------------------+
   |          Content        |
   |+-----------------------+|
   ||       Public Key      ||
   |+-----------------------+|
   +-------------------------+
   |       SignatureInfo     |
   |+-----------------------+|
   || SignatureType:  ...    ||
   || KeyLocator:     ...    ||
   || ValidityPeriod: ...    ||
   || ...                    ||
   |+-----------------------+|
   +-------------------------+
   |       SignatureValue    |
   +-------------------------+
```

```
CertificateV2 ::= DATA-TLV TLV-LENGTH
                    Name      (= /<NameSpace>/KEY/[KeyId]/[IssuerId]/[Version])
                    MetaInfo  (.ContentType = KEY,
                               .FreshnessPeriod >~ 1h))
                    Content   (= X509PublicKeyContent)
                    SignatureInfo (= CertificateV2SignatureInfo)
                    SignatureValue

X509PublicKeyContent ::= CONTENT-TLV TLV-LENGTH
                           BYTE+ (= public key bits in PKCS#8 format)

CertificateV2SignatureInfo ::= SIGNATURE-INFO-TYPE TLV-LENGTH
                                 SignatureType
                                 KeyLocator
                                 ValidityPeriod
                                 ... optional critical or non-critical extension blocks ...
```

- Name
  The name of a certificate consists of five parts as shown below:

  `/<SubjectName>/KEY/[KeyId]/[IssuerId]/[Version]`

  A certificate name starts with the subject to which a public key is bound. The following parts include the keyword `KEY` component, KeyId, IssuerId, and version components.
  `Issuer Id` is an opaque name component to identify issuer of the certificate. The value is controlled by the certificate issuer and, similar to KeyId, can be an 8-byte random number, SHA-256 digest of the issuer's public key, or a simple numerical identifier.
  For example,

  ```
    /edu/ucla/cs/yingdi/KEY/%03%CD...%F1/%9F%D3...%B7/%FD%d2...%8E
    \_________________/    \___________/ \___________/\___________/
  Certificate Namespace      Key Id       Issuer Id     Version
        (Identity)
  ```

- MetaInfo
  The `ContentType` of certificate is set to `KEY` (2).
  The `FreshnessPeriod` of certificate must be explicitly specified. The recommended value is 1 hour (3,600,000 milliseconds).
- Content
  By default, the content of a certificate is the public key encoded in <X509PublicKey>.
- SignatureInfo
  The SignatureInfo block of a certificate is required to include the `ValidityPeriod` field. `ValidityPeriod` includes two sub TLV fields: `NotBefore` and `NotAfter`, which carry two UTC time stamps in ISO 8601 compact format (`yyyymmddTHHMMSS`, e.g., "20020131T235959"). `NotBefore` indicates when the certificate takes effect while `NotAfter` indicates when the certificate expires.
  Note: Using ISO style string is the convention of specifying the validity period of certificate, which has been adopted by many certificate systems, such as X.509, PGP, and DNSSEC.

  ```
  ValidityPeriod ::= VALIDITY-PERIOD-TYPE TLV-LENGTH
                       NotBefore
                       NotAfter
  ```

```
NotBefore ::= NOT-BEFORE-TYPE TLV-LENGTH
                BYTE{15}

NotAfter ::= NOT-AFTER-TYPE TLV-LENGTH
               BYTE{15}
```

For each TLV, the TLV-TYPE codes are assigned as below:

TLV-TYPE	Assigned code (decimal)	Assigned code (hexadecimal)
ValidityPeriod	253	0xFD
NotBefore	254	0xFE
NotAfter	255	0xFF

- Extensions
  A certificate may optionally carry some extensions in SignatureInfo. An extension could be either critical or non-critical depends on the TLV-TYPE code convention. A critical extension implies that if a validator cannot recognize or parse the extension, the validator must reject the certificate. A non-critical extension implies that if a validator cannot recognize or cannot parse the extension, the validator may ignore the extension.
  The TLV-TYPE code range [256, 512) is reserved for extensions. The last bit of a TLV-TYPE code indicates whether the extension is critical or not: `1` for critical while `0` for non-critical. If an extension could be either critical or non-critical, the extension should be allocated with two TLV-TYPE codes which only differ at the last bit.
  We list currently defined extensions:

TLV-TYPE	Assigned code (decimal)	Assigned code (hexadecimal)
AdditionalDescription (non-critical)	258	0x0102

- AdditionalDescription
  `AdditionalDescription` is a non-critical extension that provides additional information about the certificate. The information is expressed as a set of key-value pairs. Both key and value are UTF-8 strings, e.g., `("Organization", "UCLA")`. The issuer of a certificate can specify arbitrary key-value pair to provide additional description about the certificate.

```
AdditionalDescription ::= ADDITIONAL-DESCRIPTION-TYPE TLV-LENGTH
                            DescriptionEntry+

DescriptionEntry ::= DESCRIPTION-ENTRY-TYPE TLV-LENGTH
```

```
                        DescriptionKey
                        DescriptionValue

DescriptionKey ::= DESCRIPTION-KEY-TYPE TLV-LENGTH
                     BYTE+

DescriptionValue ::= DESCRIPTION-VALUE-TYPE TLV-LENGTH
                       BYTE+
```

TLV-TYPE	Assigned code (decimal)	Assigned code (hexadecimal)
DescriptionEntry	512	0x0200
DescriptionKey	513	0x0201
DescriptionValue	514	0x0202

Create new certificates

To create a certificate, the developer needs to: (1) first get a certificate (2) install the certificate to `KeyChain`.

For the first step, one can get a certificate by

- ndn certificate management
  One can apply for a new certificate signed by the CA by using ndncert client module.
- the certificate file or standard input
  Reading a certificate from file:

```
// read a certificate from file
// header
#include <ndn-cxx/util/io.hpp>
...
// read from file inputFile
Certificate certificate = *(io::load<Certificate>(inputFile));
```

  Reading a certificate from standard input:

```
// read a certificate from standard input
// header
#include <ndn-cxx/util/io.hpp>
...
// read from input stream inputStream
Certificate certificate = *(io::load<Certificate>(inputStream));
```

- generate a new certificate locally

  To generate a certificate locally, one need to create a `Data` and make it a certificate.

```
// create a certificate for Key someKey
```

```
Certificate cert;
// the certificate name must satisfy the naming convention
cert.setName(someKey.getName().append("signer-info").appendVersion());

// the content type must be tlv::ContentType_Key
cert.setContentType(tlv::ContentType_Key);

// here we set the freshness period to be one day
cert.setFreshnessPeriod(time::hours(24));

// someKeyBlock is the Block for someKey.
// the type of Block must be tlv::Content
cert.setContent(someKeyBlock);

// set the validity period to be ten days from now
SignatureInfo signatureInfo;
signatureInfo.setValidityPeriod(
  security::ValidityPeriod(
    time::system_clock::now(),
    time::system_clock::now() + time::days(10)));

// sign the certificate with the signer's identity/key/certificate
// here we use the someKey as the signer
// in this way, the certificate is a self-signed certificate
m_keyChain.sign(cert, signingByKey(someKey));
```

For the second step, one can directly call `KeyChain::addCertificate` function.

```
// add the Certificate cert for the existing Key key
KeyChain keyChain;
keyChain.addCertificate(key, cert)
```

Delete certificates

To delete a certificate, the developer could use `KeyChain::deleteCertificate` function.

```
// delete the Certificate cert from the existing Key key
KeyChain keyChain;
keyChain.deleteCertificate(key, cert)
```

Set default certificates

To set a certificate to be default certificate for one key, the developer could use `KeyChain::setDefaultCertificate` function.

```
// set the Certificate cert to be default for the existing Key key
KeyChain keyChain;
keyChain.setDefaultCertificate(key, cert)
```

2.3 Signature Signing

Although the security library does not have the intelligence to automatically determine the signing key for each data packet, it still provides a mechanism, called **Default Signing Settings**, to facilitate signing process. To achieve the automatic signing and validating, the Trust Scheme is designed, which can refer to <TR-0030>.

The basic signing process in the security library would be like this: create `KeyChain` instance and supply the data packet and signing by `KeyChain::sign` method. One can sign a packet with an identity, a key, or a certificate. Also, a not strong signature generated by direct hash function is provided in ndn-cxx.

- Signing with identity
  User can use identity to sign a data packet. When signing, Identity object or identity name could be used as parameter. The default key will be used to generate signature. The default certificate of the default key will be used to generate KeyLocator.

```
// signing with Identity identity01 whose name is identityName01
KeyChain keyChain;
keyChain.sign(dataPacketA, signingByIdentity(identityName01));
keyChain.sign(dataPacketB, signingByIdentity(identity01));
```

- Signing with key
  Developer can use a specific key to sign a data packet. When signing, Key object or key name could be used as parameter. The default certificate of the default key will be used to generate KeyLocator.

```
// signing with Key key02 whose name is keyName02
KeyChain keyChain;
keyChain.sign(dataPacketA, signingByKey(keyName02));
keyChain.sign(dataPacketB, signingByKey(key02));
```

- Signing with certificate
  Developer can use a specific certificate to sign a data packet. When signing, Certificate object or certificate name could be used as parameter.

```
// signing with Key cert03 whose name is certName03
KeyChain keyChain;
keyChain.sign(dataPacketA, signingByCertificate(certName03));
keyChain.sign(dataPacketB, signingByCertificate(cert03));
```

- Hash signature
  Developer can directly sign the data packet with SHA256. There is no encryption in the signing process, which means this kind of signature is not strong.

```
// signing with SHA256
KeyChain keyChain;
keyChain.sign(dataPacketA, signingWithSha256());
```

The `KeyChain` instance will

- construct `SignatureInfo` using the signing certificate name;
- look up the corresponding private key in `TPM <SecTpm>`;
- sign the data packet if the private key exists.

2.4 Validation

Both Interest and Data validation is done through the Validator, which is a virtual base class. In some cases, the trust model is clear and application already have the knowledge of the required key or certificate. In this way, the verification could be done simply in one line of code. In some other scenarios, the application may need complex and rich policy. E.g., some application may require the name of the signer must has some relationship with name of the data. For the latter, the developer needs to create customized Validator instance by identifying the proper ValidationPolicy and CertificateFetcher.

2.4.1 Simple Verification

As mentioned before, the developer could use single line of code to achieve signature verification.

Verify Signature by Public Key

Developer can use a specific NDN Public Key to verify a Data packet or an Interest packet. When verifying, Public Key object or the Public Key Bits could be used as the parameter.

```
bool result;
// verifying signed interestA with Key key01
result = verifySignature(interestA, key01);

// verifying DataA with Key key01
result = verifySignature(DataA, key01);

// verifying signed interestB with Key bits
// uint8_t* keybits and size_t keyLen
result = verifySignature(interestB, keybits, keyLen);

// verifying DataB with Key bits
// uint8_t* keybits and size_t keyLen
result = verifySignature(DataB, keybits, keyLen);
```

Verify Signature by Certificate

Developer can use a specific NDN Certificate to verify a Data packet or an Interest packet. When verifying, Public Key object could be used as the parameter.

```
bool result;
// verifying signed interestA with Certificate cert01
result = verifySignature(interestA, cert01);

// verifying DataA with Certificate cert02
result = verifySignature(DataA, cert02);
```

Verify Digest Signature

When the packet uses digest as the signature, the validation would be done by comparing newly calculated hash value with the value in the signature.

```
bool result;
// verifying Sha256-hash-signed interestA
```

```
result = verifyDigest(interestA, tlv::DigestSha256);

// verifying Sha256-hash-signed DataA
result = verifyDigest(DataA, tlv::DigestSha256);
```

2.4.2 Verification with Validator and Policies

TBD

3. NDN Certificate Management Protocol

For the NDNCERT protocol detail, please refer to NDN-TR-50.

3.1 CA Configuration for NDNCERT

Here is a CA configuration sample.

```
{
  "ca-list":
  [
    {
        "ca-prefix": "/example1",
        "probe": "false",
        "issuing-freshness": "720",
        "validity-period": "360",
        "ca-anchor": "/example1/KEY/%9A%E0%i/self/%FD%00%00",
        "supported-challenges":
        [
            { "type": "PIN" }
        ]
    },
    {
        "ca-prefix": "/example2",
        "probe": "true",
        "issuing-freshness": "720",
        "validity-period": "180",
        "ca-anchor": "/example2/KEY/%1F%A0%0/NDNCERT/%CL%5G%60",
        "supported-challenges":
        [
```

```
            { "type": "email" }
        ]
    }
  ]
}
```

One CA server could support multiple running CAs for different namespaces. Every CA configuration file should at least contain one CA item. The developer or operator needs to know the following fields for each CA item.

- **ca-prefix**
  CA prefix is the name prefix to which the CA deamon would listen. Notice that all the NDNCERT requests are supposed to have this prefix followed by a fixed component "`/CA`".
- **probe**
  This field decides whether the CA would support probe function or not. Probe is for CA to assign identity for requesters. If the probe is `false`, requester would decide their identity by themselves and may cause identity conflicts. If the probe is `true`, requester would first send out a probe request to get an identity and then start applying new certificate.
- **issuing-freshness**
  This is to set the freshness time for the newly issued certificate data packet. Notice the metrics of this field is **second**. In the sample configuration, the value of issuing-freshness is 720. It means when CA publishes a new Certificate Data, the freshness time is 720 seconds.
  For this field, the recommended value is 720 seconds.
- **validity-period**
  This is to decide the validation period of newly issued certificate. Once the certificate is out of this period, it's no longer valid. The metrics of this field is **day**. In the sample configuration, the value of validity-period is 360, which means the certificate would be valid for one year.
  There is no recommended value for this field. CA developer or operator need to figure out the value based on their own needs.
- **ca-anchor**
  This value indicate the anchor certificate for this namespace. All the certificates issued by this CA would be signed by the certificate configured in this field. The value should be the full name of the certificate and the certificate must have been installed to local keychain already.
- **supported-challenges**
  This field contains a list of challenges supported by this CA. If the developer or operator want to add new challenge type here, they first need to have the challenge installed in the library and second add the unique challenge identifier to the list.

3.2 Client Configuration for NDNCERT

Here is a Client configuration sample.

```
{
  "ca-list":
  [
    {
        "ca-prefix": "/example1/CA",
        "ca-info": "This is example CA 1",
```

```
        "probe": "Send probe info first",
        "certificate": "Bv0sCANuZG4IBXNpdG..."
    },
    {
        "ca-prefix": "/example2/CA",
        "ca-info": "This is example CA 2",
        "certificate": "Bv0Cw9jvOPD4n0mBLA..."
    }
  ]
}
```

No matter in which kind of scenario of trust, some pre-installed certificates help to build up the first several trust CAs. The client configuration is how requester pre-installs the CA certificate. Information in each CA item should be provided by the CAs.

- **ca-prefix**
  CA prefix indicate which name prefix should be used by requester.
- **ca-info**
  A brief introduction to the CA.
- **probe**
  This field would indicate whether the CA support Probe or not and also indicate what kind of information should be provided when requesting identity.
- **certificate**
  The base64 format certificate. Client should install the certificate as the trust anchor so that later data from CA could be verified.

3.3 Sever: Build up CA for network and for local mechine

It's really simple to become a CA. To become a CA, make sure that

- There is at least one valid certificate, not matter the certificate is self-signed or issued from other CA.
- There is a configuration file for the CA.

3.3.1 Create CaModule Instance

All the functionality is implemented in the class `CaModule`. To create a CaModule, then

```
Face face;
KeyChain keyChain;
CaModule ca(face, keyChain);
```

It's OK if you want to have customized config file path.

```
Face face;
KeyChain keyChain;
std::string confPath = "/some/path";
CaModule ca(face, keyChain, confPath);
```

3.3.2 Set the ProbeHandler

It's an optional to have CA support **Probe**. Probe is for CA to assign the identity for the requester based on the probe information provided by requester. For example, you can define the rules to convert from a user email address to identity name.

Notice you can open the Probe switch in CA's configuration.

In the sample code, we simply use the requester's input probe string to become the identity name.

```
ca.setProbeHandler([&] (const std::string& probeInfo) {
      return probeInfo;
    });
```

3.3.3 Set the RequestUpdateCallback

If the developer wants CA to notify main application (e.g. Ndn Control Center) whenever the requester's request is updated, one can set the RequestUpdateCallback to achieve it.

```
ca.setRequestUpdateCallback([&] (const CertificateRequest& request) {
      std::cout << "request got updated!" << std::endl;
    });
```

3.3.4 Start the CA

When the face starts processing, CA would work as a running Deamon.

```
face.processEvents();
```

3.4 Client: Get certificate from CA for network node and for application

It's really simple to become a NDNCERT client. To become a client, make sure that

- There is a configuration file for the client containing at least one CA item.

3.4.1 Create the Client Module Instance

To create a Client instance, the code is like:

```
Face face;
security::v2::KeyChain keyChain;
ClientModule client(face, keyChain);
client.getClientConf().load(configFilePath);
```

3.4.2 Design Callback functions

The most important part of client is to use the callback functions. Here are the definitions of callback functions that would be used in application.

```
using RequestCallback = function<void (const shared_ptr<RequestState>&)>;
using ErrorCallback = function<void (const std::string&)>;
```

Here are the main functions for the certificate request

```
void
sendProbe(const ClientCaItem& ca, const std::string& probeInfo,
          const RequestCallback& requestCallback,
          const ErrorCallback& errorCallback);

void
sendNew(const ClientCaItem& ca, const Name& identityName,
        const RequestCallback& requestCallback,
        const ErrorCallback& errorCallback);

void
sendSelect(const shared_ptr<RequestState>& state,
           const std::string& challengeType,
           const JsonSection& selectParams,
           const RequestCallback& requestCallback,
           const ErrorCallback& errorCallback);

void
sendValidate(const shared_ptr<RequestState>& state,
             const JsonSection& validateParams,
             const RequestCallback& requestCallback,
             const ErrorCallback& errorCallback);

void
requestStatus(const shared_ptr<RequestState>& state,
              const RequestCallback& requestCallback,
              const ErrorCallback& errorCallback);

void
requestDownload(const shared_ptr<RequestState>& state,
                const RequestCallback& requestCallback,
                const ErrorCallback& errorCallback);
```

To start a new certificate application, the client either sends out `probe request` using `sendProbe(...)` or send `new interest` using `sendNew(...)`. These two functions would use `RequestCallback` and `ErrorCallback` as the parameters. In the `RequestCallback` function of `sendNew(...)`, developer needs to call the `sendSelect(...)`.

In the callback function of `sendSelect(...)`, developer needs to call the `sendValidate(...)`.

In the callback function of `sendValidate(...)`, developer needs to (i) call the `sendValidate(...)` (ii) call the `requestStatus(...)` (iii) call the `requestDownload(...)`.

3.4.3 Send Requests

The application developer knows which CA to send request. If the CA requires the Probe step, the code is like:

```
CaItem targetCaItem;
std::string probeInfo;
```

```
ClientModule::RequestCallback onSuccessCallback;
ClientModule::ErrorCallback onErrorCallback;

// figure out the parameter value

client.sendProbe(targetCaItem, probeInfo,
                 onSuccessCallback, onErrorCallback);
```

If the CA requires no the Probe step, the code is like:

```
CaItem targetCaItem;
Name identityName;
ClientModule::RequestCallback onSuccessCallback;
ClientModule::ErrorCallback onErrorCallback;

// figure out the parameter value

client.sendProbe(targetCaItem, identityName,
                 onSuccessCallback, onErrorCallback);
```

4. Name-based Access Control

Name-based Access Control (NAC) is an encryption-based access control scheme which provides distributed data sharing. For the NAC technical report, please refer to TR-0034.

In NAC library, totally there are five kinds of key being used.

- Content Key (C-KEY) which is a symmetric key used to encrypt content directly.
- Encryption Key (E-KEY) / Decryption Key (D-KEY) which is an asymmetric key pair used to encrypt/decrypt C-KEY.
- Consumer public key / private key which is an asymmetric key pair used to encrypt/decrypt D-KEY.

In NAC library, there are three parties with different functionality.

- Data owner. Data owner generates E-KEY/D-KEY pairs and controls consumers' access rights.
- Data producer. Data producer creates content and produce encrypted data.
- Data consumer. Data consumer fetches data and decrypt data.

4.1 Naming Conventions

4.1.1 Key naming conventions

Notice that the consumer public/private key pair is following the NDN key naming convention; the key pair will never be encrypted and encapsulated to data packet.

C-KEY

`/<data_prefix>/C-KEY/<time-slot>`

An example may be like: "`/alice/health/read/activity/C-KEY/20170101170000`". The name of the C-KEY indicates this C-KEY should be used to encrypt content which is produced in 01/01/2017 5:00 PM to 6:00 PM under prefix "`/alice/health/read/activity`".

D-KEY

`/<data_prefix>/D-KEY/[start-ts]/[end-ts]`

An example may be like: "/alice/health/read/activity/D-KEY/20170101160000/20170101180000". The name of the D-KEY indicates this D-KEY should be used to decrypt C-KEYs whose time slot is in the range of 01/01/2017 4:00 PM to 7:00 PM under prefix "/alice/health/read/activity".

E-KEY

/<data_prefix>/E-KEY/[start-ts]/[end-ts]

An example may be like: "/alice/health/read/activity/E-KEY/20170101160000/20170101180000". The name of the E-KEY indicates this E-KEY should be used to encrypt C-KEYs whose time slot is in the range of 01/01/2017 4:00 PM to 7:00 PM under prefix "/alice/health/read/activity".

4.1.2 Data packet naming conventions

Data packet containing encrypted content

/<data_name> or /<data_name>/FOR/<C-KEY name>

When the content is covered by multiple C-KEYs, to clarify which C-KEY should be fetched, the data packet should have the latter naming convention.

Data packet containing encrypted C-KEY

/<C-KEY name>/FOR/<D-KEY name>

An example may be like

/alice/SAMPLE/activity/C-KEY/20150101T090000/
FOR/alice/READ/activity/E-KEY/20150101T080000/20150101T100000}.

The name of the data packet indicates the C-KEY is encrypted by the E-KEY whose name is: "/alice /READ/activity/E-KEY/20150101T080000/20150101T100000".

Data packet containing encrypted D-KEY

/<D-KEY name>/FOR/<consumer name>

An example may be like

/alice/health/READ/activity/E-KEY
/20150101T080000/20150101T100000/alice/health/member-alice

The name of the data packet indicates the encrypted D-KEY should be decrypted by the private key of the consumer /member-alice.

Data packet containing encrypted E-KEY

/<E-KEY name>

4.2 Producer, Consumer and Data Owner

In NAC library, there are three parties with different functionality.

4.2.1 Data Owner

The main functionality of data owner.

- Create E-KEY/D-KEY pairs
- Grant consumers the access to D-KEYs by encrypting D-KEY with consumer's private key.
- Publish the E-KEYs
- Consumer management: adding/removing consumers from the access system
- Maintain the schedule for each consumer

1. Creating data owner

Assume the data namespace that the owner controls is "`/prefix/dataType`"; the database path for data owner is `dbDir`; the E-KEY/D-KEY length is `2048`; the freshness period of data packet carrying the keys is `1 hour`. Then we have:

```
GroupManager manager(Name("prefix"), Name("dataType"), dbDir, 2048, 1);
```

The function would create the data owner instance.

2. Getting E-KEY/D-KEY

Assume the data owner want to get the E-KEY/D-KEY for the specific time slot 16:30 04/13/2017. Then we have:

```
std::list<Data> result;
result = manager.getGroupKey(TimeStamp(from_iso_string("20170413T163000")));
```

The function will generate a list of Data packets. The first data packet is the E-KEY. The other data packets are encrypted D-KEY for each consumer who has access right at that time slot. Each D-KEY data packet is encrypted by corresponding consumer's public key.

3. Adding/Removing schedules

In NAC, the access unit is based on time; that is, only when one consumer have the access at one specific time slot, the consumer can get the D-KEY. Therefore on data owner's side, there is a concept of **schedule**. The schedule is like a timetable defining which user can have access at which time.

Assume the data owner want such a schedule: From 04/10/2017 Mon to 04/14/2017 Fri, the authorized time period is from 09:00 to 13:00 on Mon, Wed, and Fri, but no access from 11:00 to 13:00 on 04/12/2017 Wed. Then we can create a schedule:

```
RepetitiveInterval intervalA(from_iso_string("20170410T000000"),
                             from_iso_string("20170414T000000"),
                             9, 13, 2, RepetitiveInterval::RepeatUnit::DAY);
RepetitiveInterval intervalB(from_iso_string("20170412T000000"),
                             from_iso_string("20170412T000000"),
                             11, 13);
Schedule schedule;
schedule.addWhiteInterval(intervalA); // white interval grants access rights
schedule.addBlackInterval(intervalB); // black interval enforces no access
```

To add the schedule we created, we first need to name it. Here we name the schedule `schedule 1` and then:

```
manager.addSchedule("schedule1", schedule);
manager.deleteSchedule("schedule1");
```

4. Adding/Removing authorized consumers

All the authorized consumer is bound with a schedule. The consumer's access is based on the schedule. Adding/removing consumer is to add/remove consumer's certificate. Assume we have Alice whose certificate is "`/group/member/alice/KEY/123/NDNCERT/123`" (Here we call it `certAlice`) and Bob whose certificate is "`/group/member/bob/KEY/123/NDNCERT/123`" (Here we call it `certBob`).

```
manager.addMember("schedule1", certAlice);
manager.addMember("schedule1", certBob);
manager.removeMember(certAlice.getIdentity());
```

4.2.2 Data producer

The main functionality of data producer.

- Create C-KEY for each minimum access unit
- Fetch corresponding E-KEY from data owner
- For each C-KEY, encrypt C-KEY with E-KEY and publish the encrypted C-KEY
- Produce content
- For each piece of content, encrypt content with corresponding C-KEY and publish the encrypted content

Example:

1. Creating data producer

Assume the data producer has the namespace `/prefix` and would produce data with name "`/prefix/dataType/[TimeStamp]`";

the database for data producer is `dbDir`; the face through which the producer wants to publish data is `face`. Then we have:

```
Producer producer(prefix, dataType, face, dbDir);
```

2. Producing C-KEY

To create a C-KEY for a specific time slot which we assume it's 16:00 04/13/2017, we need the code:

```
typedef function<void(const std::vector<Data>&)> ProducerEKeyCallback;
Name contentKeyName;
contentKeyName = producer.createContentKey(from_iso_string("20170413T160001"),
                                           ProducerEKeyCallback());
```

Notice that the result, the C-KEY data packets encrypted by the corresponding E-KEYs, would be passed back through callback function.

3. Producing content.

To produce an data packet encrypted using the content key corresponding time slot, where we assume it's 16:00 04/13/2017, we need the code:

```
uint8_t DATA_CONTENT[] = {0xcb, 0xe5, 0x6a, 0x80, 0x41, 0x24, 0x58, 0x23};
Data result;
producer.produce(result, from_iso_string("20170413T160001"),
                 DATA_CONTENT, sizeof(DATA_CONTENT));
```

4.2.3 Data Consumer

The main functionality of data consumer.

- Fetch encrypted content from data producer
- Fetch encrypted C-KEY from data producer
- Fetch encrypted D-KEY from data owner
- Decrypt encrypted D-KEY with consumer's private key
- Decrypt encrypted C-KEY with D-KEY
- Decrypt encrypted content with C-KEY

1. Creating data consumer

Assume the data consumer's identity is "`/group/member/bob`" has access to data with namespace "`/prefix`"; the database for data consumer is `dbDir`; the face through which the consumer wants to send out interests is `face`. Then we have:

```
Consumer consumer(face, Name("prefix"), Name("/group/member/bob"), dbDir);
```

2. Consuming data

When data consumer wants to fetch data packet with name "`/prefix/data`", the code would be like:

```
typedef function<void (const Data&, const Buffer&)> ConsumptionCallBack;
consumer.consume(Name("/prefix/data"), ConsumptionCallback(), ErrorCallback);
```

The function would automatically fetch corresponding content data packet, C-KEY data packet, and D-KEY data packet. The decrypted content would be passed back through the `ConsumptionCallback`.

5. NDN Security Tools

5.1 ndn-cxx Security Command Line Tools

5.1.1 List Identities/Keys/Certificates

`ndnsec-list` or `ndnsec-ls-identity` is a tool to display entities stored in **Public Information Base (PIB)**, such as identities, keys, and certificates.

Usage

```
$ ndnsec-list [-h] [-KkCc]
```

Description

`ndnsec-list` lists names of all the entities according to the granularity specified in options (The default granularity is identity). The default entities will be marked with * in front of their names. For example:

```
$ ndnsec list
* /ndn/edu/ucla/cs/yingdi
  /ndn/test/cathy
  /ndn/test/bob
  /ndn/test/alice
```

Options

`-K, -k` Display key names for each identity. The key name with * in front is the default key name of the corresponding identity.

`-C, -c` Display certificate names for each key. The certificate name with * in front is the default certificate name of the corresponding key.

Examples

Display all the key names in PIB.

```
$ ndnsec-list -k
* /ndn/edu/ucla/cs/yingdi
  +->* /ndn/edu/ucla/cs/yingdi/KEY/1397247318867
  +->  /ndn/edu/ucla/cs/yingdi/KEY/1393811874052

  /ndn/test/cathy
  +->* /ndn/test/cathy/KEY/1394129695418

  /ndn/test/bob
  +->* /ndn/test/bob/KEY/1394129695308

  /ndn/test/alice
  +->* /ndn/test/alice/KEY/1394129695025
```

Display all the certificate names in PIB.

```
$ ndnsec-list -c
* /ndn/edu/ucla/cs/yingdi
  +->* /ndn/edu/ucla/cs/yingdi/KEY/1397247318867
       +->* /ndn/edu/ucla/cs/yingdi/KEY/KEY/1397247318867/NDNCERT/%00%00%01ERn%1B%BE
  +->  /ndn/edu/ucla/cs/yingdi/KEY/1393811874052
       +->* /ndn/edu/ucla/cs/yingdi/KEY/KEY/1393811874052/NDNCERT/%FD%01D%85%A9a%DD

  /ndn/test/cathy
  +->* /ndn/test/cathy/KEY/1394129695418
       +->* /ndn/test/KEY/cathy/KEY/1394129695418/NDNCERT/%FD%01D%98%9A%F3J

  /ndn/test/bob
  +->* /ndn/test/bob/KEY/1394129695308
       +->* /ndn/test/KEY/bob/KEY/1394129695308/NDNCERT/%FD%01D%98%9A%F2%AE

  /ndn/test/alice
  +->* /ndn/test/alice/KEY/1394129695025
       +->* /ndn/test/KEY/alice/KEY/1394129695025/NDNCERT/%FD%01D%98%9A%F2%3F
```

5.1.2 Delete Identities/Keys/Certificates

`ndnsec-delete` is a tool to delete security data from both **Public Info Base** and **Trusted Platform Module**.

Usage

```
ndnsec-delete [-h] [-kc] name
```

Description

By default, `ndnsec-delete` interpret `name` as an identity name. If an identity is deleted, all the keys and certificates belonging to the identity will be deleted as well. If a key is deleted, all the certificate belonging to the key will be deleted as well.

Options

`-k` Interpret `name` as a key name and delete the key and its related data.

`-c` Interpret `name` as a certificate name and delete the certificate.

Exit Status

Normally, the exit status is 0 if the requested entity is deleted successfully. If the entity to be deleted does not exist, the exit status is 1. For other errors, the exit status is 2.

Examples

Delete all data related to an identity:

```
$ ndnsec-delete /ndn/test/david
```

5.1.3 Get Default Identities/Keys/Certificates

`ndnsec-get-default` is a tool to display the default setting of a particular entity.

Usage

```
$ ndnsec-get-default [-h] [-kc] [-i identity|-K key] [-q]
```

Description

Given a particular entity, `ndnsec-get-default` can display its default setting as specified in options. If `identity` is specified, the given entity becomes the identity. If `key` is specified, the given identity becomes the key. If no entity is specified, the command will take the system default identity as the given entity.

Options

`-k` Display the given entity's default key name.

`-c` Display the given entity's default certificate name.

`-i identity` Display default setting of the `identity`

`-K key` Display default setting of the `key`.

`-q` Disable trailing new line character.

Examples

Display an identity's default key name.

```
$ ndnsec-get-default -k -i /ndn/test/alice
/ndn/test/alice/KEY/1394129695025
```

Display an identity's default certificate name.

```
$ ndnsec-get-default -c -i /ndn/test/alice
/ndn/test/KEY/alice/KEY/1394129695025/NDNCERT/%FD%01D%98%9A%F2%3F
```

Display a key's default certificate name.

```
$ ndnsec-get-default -c -K /ndn/test/alice/KEY/1394129695025
/ndn/test/KEY/alice/KEY/1394129695025/NDNCERT/%FD%01D%98%9A%F2%3F
```

5.1.4 Set Default Identities/Keys/Certificates

`ndnsec-set-default` is a tool to change the default security settings.

Usage

```
$ ndnsec-set-default [-h] [-k|c] name
```

Description

By default, `ndnsec-set-default` takes `name` as an identity name and sets the identity as the system default identity.

Options

`-k` Set default key. `name` should be a key name, `ndnsec-set-default` can infer the corresponding identity and set the key as the identity's default key.

`-c` Set default certificate. `name` should be a certificate name, `ndnsec-set-default` can infer the corresponding key name and set the certificate as the key's default certificate.

Examples

Set a key's default certificate:

```
$ ndnsec-set-default -c /ndn/test/KEY/alice/KEY/1394129695025/NDNCERT/%FD%01D%98%9A%F2%3F
```

Set an identity's default key:

```
$ ndnsec-set-default -k /ndn/test/alice/KEY/1394129695025
```

Set system default identity:

```
$ ndnsec-set-default /ndn/test/alice
```

5.1.5 Generate a New Key

`ndnsec-key-gen` is tool to generate a pair of key.

Usage

```
$ ndnsec-key-gen [-h] [-n] [-d] [-t keyType] identity
```

Description

`ndnsec-key-gen` creates a key pair for the specified `identity` and sets the key as the identity's default key. `ndnsec-key-gen` will also create a signing request for the generated key. The signing request will be written to standard output in base64 encoding. By default, it will also set the identity as the system default identity.

Options

`-n` Do not set the identity as the system default identity. Note that if it is the first identity/key/certificate, then it will be set as default regardless of `-n` flag.

`-t keyType` Specify the key type. `r` (default) for RSA key. `e` for ECDSA key.

Examples

```
$ ndnsec-key-gen /ndn/test/zhiyi
Bv0CwQcyCANuZG4IBHRlc3QIBXpoaXlpCANLRVkICLc7RCqUMJsBCARzZWxmCAn9
AAABXFeBQmEUCRgBAhkEADbugBX9ASYwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAw
ggEKAoIBAQDI+5XnQxL2iIORZ+ijbOF3na/ayDNhmVvy0QY2fpEIG6/u5xaKD0kt
r+m1P7Ibm1HTF/qdXDrJe++zif0WnGKfaNXH8ZGRej2TU387rUhE0b+RfCij1gaV
R5WHxB7us++3KsBiRi3s9yrGpRDtWXMjfTl09I6PSLKU57WPmATPSt9RF/24xWVN
/IiOqzB2aHDtrOO/DMaSe7gtOMANL8bsmu8GiVl9wdLm+UBQe1zDWbA1D19opkyO
Px7fg8ZNKqsrHIL7WQ/JWR6hoheOup3G5Fl4hByNM4Qkt3JwU3Pwx3+37z550+e+
HR/MQ1RMDm/4FS52xGqN9UcuAkPq/Q2ZAgMBAAEWUhsBARwjByEIA25kbggEdGVz
dAgFemhpeWkIA0tFWQgItztEKpQwmwH9AP0m/QD+DzE5NzAwMTAxVDAwMDAwMP0A
/w8yMDM3MDUyNVQwMzU4MjcX/QEAPyF53NI4lsMBB+maUBsT3DUG5ttgLzgTHIM5
x4dBJ3gaVqCm34/CqH/XoZDRMYr378fJ8zTjKhv+exgb/LRAkJYBtXmrRZpwafaP
GHsBQP89baxz48suUeUi6mGOaycCV/VmG+3DDjbqm5Gx+biv5j7jAgH7UsyVE7uU
eJKvCl117CR/JOG8G4CRc2e8AUg69GAngAAhLSLhdsoiGwg3gSST5K+jUtBDiCk1
pv+XEiAB/bk6LB/eGzCBtyCZdA44v5Sb6Y/qa1xhcSQKWKIKjHKQt/QU2X4UtL0Y
OCXnDOjv4zKvRce0YO3aUFuZ6w1ruyfarL1ZHngWpZJyv0CvGg==
```

5.1.6 Generate a Certificate Request for a Key

`ndnsec-sign-req` is a tool to generate a signing request for a particular key.

Usage

```
$ ndnsec-sign-req [-h] [-k] name
```

Description

The signing request of a key is actually a self-signed certificate. Given key's information, `ndnsec-sign-req` looks up the key in PIB. If such a key exists, a self-signed certificate of the key, or its signing request, will be outputed to **stdout** with base64 encoding.

By default, `name` is interpreted as an identity name. `ndnsec-sign-req` will generate a signing request for the identity's default key.

Options

`-k` Interpret `name` as a key name.

Examples

Create a signing request for an identity's default key.

```
$ ndnsec-sign-req /ndn/test/zhiyi
Bv0CyQc6CANuZG4IBHRlc3QIBXpoaXlpCANLRVkICLc7RCqUMJsBCAxjZXJ0LXJl
cXVlc3QICf0AAAFcV4jHGBQJGAECGQQANu6AFf0BJjCCASIwDQYJKoZIhvcNAQEB
BQADggEPADCCAQoCggEBAMj7ledDEvaIg5Fn6KNs4Xedr9rIM2GZW/LRBjZ+kQgb
r+7nFooPSS2v6bU/shubUdMX+p1cOsl777OJ/RacYp9o1cfxkZF6PZNTfzutSETR
v5F8KKPWBpVHlYfEHu6z77cqwGJGLez3KsalEO1ZcyN9OXT0jo9IspTntY+YBM9K
31EX/bjFZU38iI6rMHZocO2s478MxpJ7uC04wA0vxuya7waJWX3B0ub5QFB7XMNZ
sDUPX2imTI4/Ht+Dxk0qqyscgvtZD8lZHqGiF466ncbkWXiEHI0zhCS3cnBTc/DH
f7fvPnnT574dH8xDVEwOb/gVLnbEao31Ry4CQ+r9DZkCAwEAARZSGwEBHCMHIQgD
bmRuCAR0ZXN0CAV6aGl5aQgDS0VZCAi3O0QqlDCbAf0A/Sb9AP4PMjAxNzA1MzBU
MDQwNjQx/QD/DzIwMTcwNjA5VDA0MDY0MBf9AQB7grtVx1PbkUjRumbDREpCpUCl
```

```
iFXtznijlgucAgTgJEBJdE2caFwW1P2pgJmhkvIHCFSqhX3GvIDfpGgoh88rik83
IAX+gqKjdgsCbrecUAEEHG9HOvOZpfreNBI/a08095n0twHLj2gH3zC+hUJzt/tB
mfHswAxoi/e0Lfm2FMJzlaC0oNRzDRcnWMYGvvuZ7RNddVhh5rh8QVLnsOxiotNo
SLNy7QB/+PwHN1/fHXC18ZgIBHcXAVRMy6cgpjJTI5Jn31OEpWtx8v1HJGEefljk
xHPbRSTylNnqv9apVNTfA6/BlftZWGaipgo6nNLlqKkMyZpM695+ZBrRdLx7
```

Create a signing request for a particular key.

```
$ ndnsec-sign-req -k /ndn/test/zhiyi/KEY/%B7%3BD%2A%940%9B%01
Bv0CyQc6CANuZG4IBHRlc3QIBXpoaXlpCANLRVkICLc7RCqUMJsBCAxjZXJ0LXJl
cXVlc3QICf0AAAFcV4luAxQJGAECGQQANu6AFf0BJjCCASIwDQYJKoZIhvcNAQEB
BQADggEPADCCAQoCggEBAMj7ledDEvaIg5Fn6KNs4Xedr9rIM2GZW/LRBjZ+kQgb
r+7nFooPSS2v6bU/shubUdMX+p1cOsl777OJ/RacYp9o1cfxkZF6PZNTfzutSETR
v5F8KKPWBpVHlYfEHu6z77cqwGJGLez3KsalEO1ZcyN9OXT0jo9IspTntY+YBM9K
31EX/bjFZU38iI6rMHZocO2s478MxpJ7uC04wA0vxuya7waJWX3B0ub5QFB7XMNZ
sDUPX2imTI4/Ht+Dxk0qqyscgvtZD8lZHqGiF466ncbkWXiEHI0zhCS3cnBTc/DH
f7fvPnnT574dH8xDVEwOb/gVLnbEao31Ry4CQ+r9DZkCAwEAARZSGwEBHCMHIQgD
bmRuCAR0ZXN0CAV6aGl5aQgDS0VZCAi3O0QqlDCbAf0A/Sb9AP4PMjAxNzA1MzBU
MDQwNzI0/QD/DzIwMTcwNjA5VDA0MDcyMxf9AQAQE10+dbgLg7FkYEe8T1wRQqJN
9CJYSPNr1HLg3BqJ+evJjCtHGFk4e87r9HUhMx2JPsL8ldl8q/QzzqSgLuh1L+nm
Ts2d0Aod15mVOPJpZHsZnuTPemZS0lol0kYkSTi//xFoyrnFCriTiwW7+LesulnJ
ASoRVZJfyaj/G+O0g5DmrHBwDDUku9x7N6HQMiplK9iFpuWiM7Q5HIjsX/UCkGYD
em/K1dk8sb8pWVnE6K+o7IBJElqbyriDWPhH08AHnNSsyBr0jRjoCw4fUCBAoiRC
cWbzI+5gN0eTgmY6msRKofR0HkPwxwWTdLu9KCbXH6nsO/C14GdgvosuliUb
```

5.1.7 Generate a New Certificate

`ndnsec-cert-gen` is a tool to issue an identity certificate.

Usage

```
$ ndnsec-cert-gen [-h] [-S timestamp] [-E timestamp] [-I info] [-s sign-id] [-i issuer-id]
```

Description

`ndnsec-cert-gen` takes signing request as input and issues an identity certificate for the key in the signing request. The signing request can be created during `ndnsec-keygen` and can be re-generated with `ndnsec-sign-req`.

By default, the default key/certificate will be used to sign the issued certificate.

`request` could be a path to a file that contains the signing request. If `request` is `-`, then signing request will be read from standard input.

The generated certificate will be written to standard output in base64 encoding.

Options

`-S timestamp` Timestamp when the certificate becomes valid. The default value is now.

`-E timestamp` Timestamp when the certificate expires. The default value is one year from now.

`-I info` Other information to be included in the issued certificate. For example,

```
-I "affiliation Some Organization" -I "homepage http://home.page/"
```

`-s sign-id` Signing identity. The default key/certificate of `sign-id` will be used to sign the requested certificate. If this option is not specified, the system default identity will be used.

`-s issuer-id` Issuer's ID to be included as part of the issued certificate name. If not specified, "NA" value will be used

Examples

```
$ ndnsec-cert-gen -S 20140401000000 -E 20150331235959 -N "David"
-I "2.5.4.10 'Some Organization'" -s /ndn/test sign_request.cert
Bv0C9wc9CANuZG4IBHRlc3QIA0tFWQgFZGF2aWQIEWtzay0xMzk2OTEzMDU4MTk2
CAdJRC1DRVJUCAgAAAFFPp2g3hQDGAECFf0BdjCCAXIwIhgPMjAxNDA0MDEwMDAw
MDBaGA8yMDE1MDMzMTIzNTk1OVowKDAMBgNVBCkTBURhdmlkMBgGA1UEChMRU29t
ZSBPcmdhbml6YXRpb24wggEgMA0GCSqGSIb3DQEBAQUAA4IBDQAwggEIAoIBAQC0
urnS2nKcnXnMTESH2XqO+H8c6bCE6mmv+FMQ9hSfZVOHbX4kkiDmkcAAf8NCvwGr
kEat0NQIhKHFLFtofC5rXLheAo/UxgFA/9bNwiEjMH/c8EN2YTSMzdCDrK6TwE7B
623cLTsa3Bb11+BpzC1oLb3Egedgp+vIf+AFIgNQhvfwzsgsgOBB4iJBwcYegU7w
JsO0pjY69WQU2DGjABFef6C2Qh8x0TvtnynRLbWlh928+4ilVUvLuWcV3AbPIKLe
eZu13+v01JN6kFzNZDPMFtOFPvJ943IdYu7Q9k93PzhSk0+wFp3cHH21PfWeghWe
3zLIER8RTWPIQhWSbxRVAgERFjMbAQEcLgcsCANuZG4IA0tFWQgEdGVzdAgRa3Nr
LTEzOTQxMjk2OTQ3ODgIB0lELUNFUlQX/QEABUGcl7U+F8cwMHKckerv+1H2Nvsd
OfeqX0+4RzWU+wRx2emMGMZZdHSx8M/i45hb0P5hbNEF99L35/SrSTSzhTZdOriD
t/LQOcKBoNXY+iw3EUFM0gvRGU0kaEVBKAHtbYhtoHc48QLEyrsVaMqmrjCmpeF/
JOcClhzJfFW3cZ/SlhcTEayF0ntogYLR2cMzIwQhhSj5L/Kl7I7uxNxZhK1DS98n
q8oGAxHufEAluPrRpDQfI+jeQ4h/YYKcXPW3Vn7VQAGOqIi6gTlUxrmEbyCDF70E
xj5t3wfSUmDa1N+hLRMdEAI+IjRRHDSx2Lhj/QcoPIZPWwKjBz9CBL92og==
```

5.1.8 Dump a Certificate

`ndnsec-cert-dump` is a tool to dump a certificate from **Public Info Base** or file and output it to standard output.

Usage

```
$ ndnsec-cert-dump [-h] [-p] [-ikf] name
```

Description

`ndnsec-cert-dump` can read a certificate from **Public Info Base (PIB)** or a file and output the certificate to standard output.

By default, `name` is interpreted as a certificate name.

Options

`-i` Interpret `name` as an identity name. If specified, the certificate to dump is the default certificate of the identity.

`-k` Interpret `name` as a key name. If specified, the certificate to dump is the default certificate of the key.

`-f` Interpret `name` as a path to a file containing the certificate. If `name` is `-`, certificate will be read from standard input.

`-p` Print out the certificate to a human-readable format.

Examples

Dump a certificate from PIB to standard output:

```
$ ndnsec-cert-dump /ndn/test/david/KEY/1396913058196/NDNCERT/%00%00%01E%3E%9D%A0%DE
```

Dump a certificate to a human-readable format:

```
$ ndnsec-cert-dump -p /ndn/test/david/KEY/1396913058196/NDNCERT/%00%00%01E%3E%9D%A0%DE
Certificate name:
  /ndn/test/david/KEY/1396913058196/NDNCERT/%00%00%01E%3E%9D%A0%DE
Validity:
  NotBefore: 20140401T000000
  NotAfter: 20150331T235959
Subject Description:
  2.5.4.41: David
  2.5.4.10: Some Organization
Public key bits:
MIIBIDANBgkqhkiG9w0BAQEFAAOCAQ0AMIIBCAKCAQEAtLq50tpynJ15zExEh9l6
jvh/HOmwhOppr/hTEPYUn2VTh21+JJIg5pHAAH/DQr8Bq5BGrdDUCIShxSxbaHwu
a1y4XgKP1MYBQP/WzcIhIzB/3PBDdmE0jM3Qg6yuk8BOwett3C07GtwW9dfgacwt
aC29xIHnYKfryH/gBSIDUIb38M7ILIDgQeIiQcHGHoFO8CbDtKY2OvVkFNgxowAR
Xn+gtkIfMdE77Z8p0S21pYfdvPuIpVVLy7lnFdwGzyCi3nmbtd/r9NSTepBczWQz
zBbThT7yfeNyHWLu0PZPdz84UpNPsBad3Bx9tT31noIVnt8yyBEfEU1jyEIVkm8U
VQIB
```

5.1.9 Install a New Certificate

`ndnsec-cert-install` is a tool to install a certificate into **Public Information Base (PIB)**.

Usage

```
$ ndnsec-cert-install [-h] [-IKN] cert-source
```

Description

`ndnsec-cert-install` can insert a certificate into PIB. By default, the installed certificate will be set as the default certificate of its corresponding identity and the identity is set as the system default identity.

`cert-source` could be a filesystem path or an HTTP URL of a file containing to certificate to install or . If `cert-file` is `-`, the certificate will be read from standard input.

Options

`-I` Set the certificate as the default certificate of its corresponding identity, but do not change the system default identity.

`-K` Set the certificate as the default certificate of its corresponding key, but do not change the corresponding identity's default key and the system default identity.

`-N` Install the certificate but do not change any default settings.

Examples

Install a certificate and set it as the system default certificate:

```
$ ndnsec-cert-install cert_file.cert
```

Install a certificate with HTTP URL and set it as the system default certificate:

```
$ ndnsec-install-cert "http://ndncert.domain.com/cert/get/my-certificate.ndncert"
```

Install a certificate but do not change any default settings:

```
$ ndnsec-cert-install -N cert_file.cert
```

5.1.10 Export Security Data of an identity

`ndnsec-export` is a tool to export an identity's security data

Usage

```
$ ndnsec-export [-h] [-o output] [-p] identity
```

Description

`ndnsec-export` can export public data of the `identity` including default key/certificate. `ndnsec-export` can also export sensitive data (such as private key), but the sensitive data will be encrypted. The exported identity can be imported again using `ndnsec-import`.

By default, the command will write exported data to standard output.

Options

`-o output` Output the exported data to a file pointed by `output`.

`-p` Export private key of the identity. A password will be asked for data encryption.

Examples

Export an identity's security data including private key and store the security data in a file:

```
$ ndnsec-export -o id.info -p /ndn/test/alice
```

Export an identity's security data without private key and write it to standard output:

```
$ ndnsec-export /ndn/test/alice
```

5.1.11 Import Security Data of an identity

`ndnsec-import` is a tool to import an identity's security data that is prepared by `ndnsec-export`.

Usage

```
$ ndnsec-import [-h] [-p] input
```

Description

`ndnsec-import` read data from `input`. It will ask for password if the input contains private key. If `input` is `-`, security data will be read from standard input.

Options

`-p` Indicates the imported data containing private key. A password will be asked for data encryption.

Examples

Import an identity's security data including private key:

```
$ ndnsec-import -p input_file
```

5.1.12 Unlock the TPM

`ndnsec-unlock-tpm` is a tool to (temporarily) unlock the **Trusted Platform Module (TPM)** that manages private keys.

Usage

```
$ ndnsec-unlock-tpm [-h]
```

Description

`ndnsec-unlock-tpm` will ask for password to unlock the TPM.

6. In Progress Security Projects

6.1 Trust Schema

TBD

6.2 Certificate Bundle

TBD

6.3 Intra-Node NDNCERT

TBD

7. Future Plan of NDN Security

TBD

Bibliography