NDN, Technical Report NDN-0009 http://named-data.net/techreports.html
Revision 2: February, 2013

Deploying Key Management on NDN Testbed

Chaoyi Bian, Peking University, bcy@pku.edu.cn
Zhenkai Zhu, UCLA, zhenkai@cs.ucla.edu
Alexander Afanasyev, UCLA, afanasev@cs.ucla.edu
Ersin Uzun, PARC, ersin.uzun@parc.com
Lixia Zhang, UCLA, lixia@cs.ucla.edu

I. Introduction

This document describes an overall design and implementation of a key management system on NDN testbed.

II. Overview

To make the testbed key deployment happen in a timely fashion, we start with a simple certification chain trust model, with the root key of the NDN testbed as a common well known trust anchor. There is a root key for the NDN testbed, which signs the keys for each site, which in turn sign keys of the users at each site. Users' keys can be used to further sign their devices and specific applications (Fig. 1). This simple design allows one to follow the trust chain to verify the legitimacy of a key, a capability much needed to meet the need for a number of usages including NDN routing and auto configuration. This document specifically focuses on getting keys published on the testbed ASAP, as a preliminary step towards understanding general trust models.

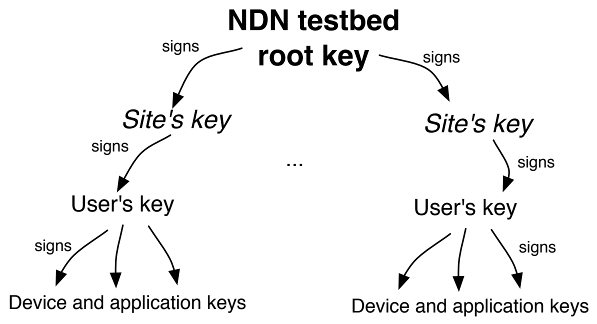


Fig. 1: Key trust model on NDN testbed

The keys of all the NDN sites and users should be made available to everybody on the testbed, regardless of current connectivity status of the particular site or user. At the initial deployment, in order to achieve this, each signed key needs to be published to a ccnx-repo and synchronized among all testbed hubs (Figure 2). To be able to use ccnx-sync protocol for key synchronization, all the key names need to follow the naming convention of having "`/ndn/keys`" prefix, which is routed to every testbed site (see Section III).

Note that presence of "`/ndn/keys`" prefix is only a shortcut to facilitate key deployment and synchronization using ccnx-sync protocol. We expect that this prefix will be phased out at some point in the future.

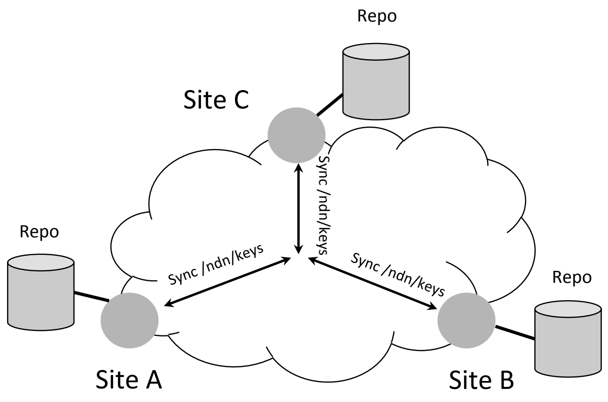


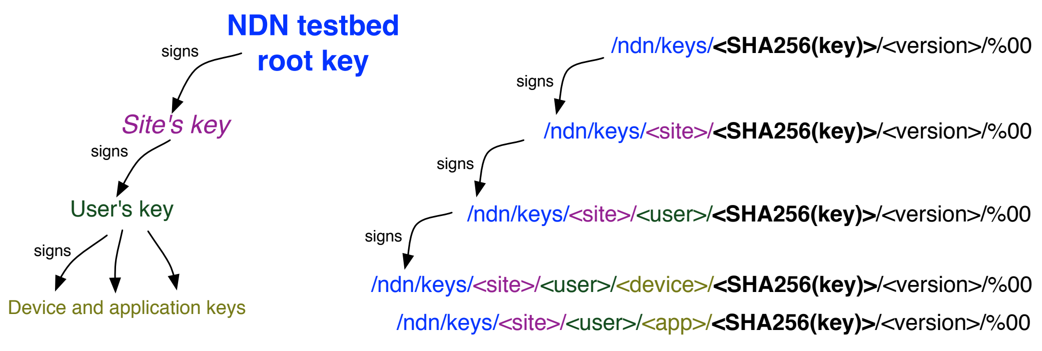
Fig. 2: Sites syncs keys

NDN testbed root key manager (root-key-admin@named-data.net) is responsible to sign and publish public keys of each NDN participating site. Users keys are signed and published by each individual site operator. At the discretion of the site's operator, device and applications keys signed by individual users may be also published to the NDN testbed repos.

Each individual user is responsible to maintain a local repo,[1] which store copies of key and signatures that form certification chain from user's key to the NDN testbed root key. This requirement is necessary to allow key verification without permanent connection to the testbed, e.g., site's A user should be able to verify site's B user's key relying only on Interest/Data exchange between these two users.

Any application, when creating a Data packet, should put full name of the key used to sign this packet in "`KeyLocator/KeyName`" field (http://www.ccnx.org/releases/latest/doc/technical/ContentObject.html). When a Data packet is received, one can follow the "`KeyLocator/KeyName`" field to fetch the public key object and verify the signature. The key object itself is also a Data packet which contains another "`KeyLocator/KeyName`", and thus one can always trace the

[1]CCNx version 0.7.1 introduces an ability to automatically launch repo using ccndstart command, provided "`CCNR_DEFAULT_PREFIX`" variable is configured. For example, users may set "`CCNR_DEFAULT_PREFIX=/ndn/keys`" in ccndrc file.



NDN testbed root key
signs
Site's key
signs
User's key
signs
Device and application keys
/ndn/keys/<SHA256(key)>/<version>/%00
signs
/ndn/keys/<site>/<SHA256(key)>/<version>/%00
signs
/ndn/keys/<site>/<user>/<SHA256(key)>/<version>/%00
signs
/ndn/keys/<site>/<user>/<device>/<SHA256(key)>/<version>/%00
/ndn/keys/<site>/<user>/<app>/<SHA256(key)>/<version>/%00

Fig. 3: Key naming

"`KeyLocator/KeyName`" chain until a trusted anchor is reached. NDN testbed root key is "public knowledge" and is stored in a self-signed Data packet, in which the root key is stored in "`Content`" section, as well as duplicated in "`KeyLocator/Key`" field. Due to limitation in the current key verification mechanism in ccnx libraries, an external key verification process (see Section V) is needed to ensure that certification chain is terminated at the trust anchor.

III. Naming

This section shows what the names of public keys will look like (Figure 3).

The first part of the name is a common prefix "`/ndn/keys`", which indicates that it is a name for a public key, and only this prefix needs to be routed to all NDN testbed sites. Using a common prefix also has the advantage of making repo synchronization easier (see Section VI).

The middle part of the name reflects the hierarchy in the testbed. The key will be verified in a chain in accordance with the name hierarchy, which is also explicitly indicated in the key content (described in Section IV).

Next part of the name is the hash value (SHA256) of the corresponding public key, which provides uniqueness guarantee for the key name on NDN testbed.

Note that SHA256 component of the key name provides only a guaranteed uniqueness of the key name. Similar guarantees can be provided by other means and future revisions of the current document may specify other acceptable forms of this component.

As the key name not only identifies key itself, but also key certification, the current design mandates creating a new public-private key pair (and as a result, new unique name of the key content object) whenever there is any change with key or key certification.

Note that the last part of the key name ("`<version>/%00`") is currently necessary due to naming restrictions imposed by ccnx-repo protocol. These two components are not an integral part of the name and will be eventually phased out.

Also note that since "`<version>/%00`" is not an integral part of the name, it should not be included in "`KeyLocator/KeyName`" fields of Data packets.

IV. NDN Key Object

This section describes how the public key object on NDN testbed looks. We compare the key object to the traditional X.509 certificate [1] in Figure 4.

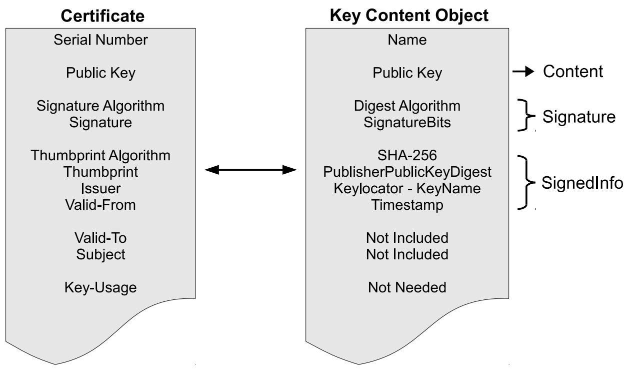


Fig. 4: The comparison of X.509 certificate and key object

- The "`Serial number`" in the X.509 certificate used for unique identification is replaced by the "`Name`" of the key content object.
- The "`Signature algorithm`" and "`Signature`" in the certificate are the same as the "`DigestAlgorithm`" and "`SignatureBits`" fields contained in the signature part of content object.
- The "`Thumbprint`" is the same as the "`PublisherPublicKeyDigest`" field in the "`SignedInfo`" part of the content object. The current CCNx implementation uses SHA-256 and it is expected that more choices will be provided in future distributions.
- The "`Issuer`" in the certificate is the entity that verified the information and issued this certificate. In the content object, "`KeyLocator/KeyName`" serves the same purpose of telling how to find the public key to verify this content object.
- The "`Valid-From`" information can be expressed with the time-stamp field in the "`SignedInfo`" part of content

object.

- Current format of content object in ccnx does not have equivalents for the "`Subject`" (the entity identified by this certificate, which is the real-world identity of the public key's owner), "`Valid-To`" (expressed in seconds since the start of Unix time), and "`Key-Usage`" information (restriction on how the key can be used, e.g., to only create digital signatures, only to sign certificates, etc.). These fields can be emulated using additional meta-info content object, including "`/info`" suffix to the public key's name:

  `/ndn/keys/<site>../info/SHA256(key)`

  Initially, we will be using an XML form of meta-info content, which specifies the real-world identity, affiliation, and validity of the key ("`Name`", "`Affiliation`", and "`Valid_to`" fields). More information can be added to the meta-info if needed.

```
<Meta>
  <Name>Alice </Name>
  <Affiliation >Wonderland</Affiliation >
  <Valid_to >1370981430</Valid_to>
</Meta>
```

V. Verification

This section describes the verification process in detail. As indicated previously, whenever a data packet comes, in addition to basic verification done by ccnx library, an additional verification needs to be invoked in order to ensure validity of the packet.

We believe that trust evaluation and final determination of the data packet validity is ultimate responsibility of the application that requested this data packet. This process should follow the following steps:

1) Check the "`KeyLocator`" field of the content object contained in the packet. If the field does not contain a "`KeyName`" indicating a signing key, or the key itself, the packet cannot be verified. Otherwise, there is a "`Key-Name`", a key itself or a certificate in the "`KeyLocator`" field.
   The only exceptions from this rule is when the content object corresponds to a self-certified trust anchor. Each application on the NDN testbed will have NDN testbed's root key as a preconfigured self-certified trust anchor. In addition to that, each individual site may elect to a provide local trust anchor for local users. For example, UCLA can self-sign "`/ndn/keys/ucla.edu/SHA256(key)`" and distribute this key out-of-band to all local users and applications.
2) Check if "`KeyName`" is a name of the trust anchor (root key) or one of the known and previously verified keys.[2] If a match is found, the application can verify packet's signature and make the final determination of the packet validity.
3) Otherwise, the application needs to express an Interest for "`KeyName`". If the fetching is unsuccessful, the packet cannot be verified. Otherwise, we get the Data packet of the key object, which needs to be verified recursively applying steps 1, 2, and 3.

Figure 5 illustrates the verification process for the signature of a content. Note that when an application maintains a cache of verified keys and the corresponding entries are still valid, there is no need to trace the "`KeyLocator`" field until we reach the root key.

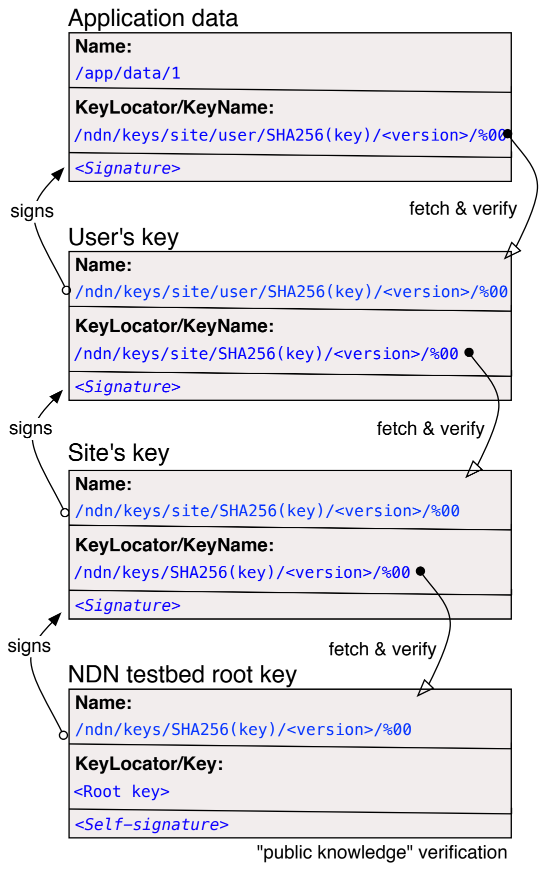


Fig. 5: Verification of the signature

A. Key and Namespace Association

The current document describes only basic key management and key verification on NDN testbed and does not impose any limits on how keys are used. In other words, verification

[2] While not strictly necessary, an application can maintain a trusted key database, storing previously fetched and verified keys. This would avoid necessity to express Interests to fetch keys and eliminate redundant key verification. At the same time, in most cases such Interests will be immediately satisfied from caches of the local ccnd, as the basic key verification mechanism in ccnx library will fetch the keys before passing the data packet to the application.

process described above will succeed if certification chain is rooted at the NDN testbed root's key, no matter how long is the chain or what are the chain links.

As the application is ultimately responsible for final decision, it is also application's responsibility to define trust model for keys (i.e., which key is authorized to sign which content). For example, the application may define and enforce trust model in which key "`/ndn/keys/<site>/<user>/SHA256(key)`" is authorized to sign only application data in "`/ndn/<site>/<user>/`" namespace and keys in "`/ndn/keys/<site>/<user>/`" namespace. A separate document(s) will describe alternative trust models that can be used for different applications.

VI. Implementation and Operations

The implementation utilizes the sync facility of CCNx repositories. It consists of two parts. First, a simple (command-line) application, utilizing CCNx sync an repo functions, to sign and publish keys. This is mainly used by site operators. Second, a key verification library (C/C++ APIs provided) for applications to verify the signatures of Data packets and manage the local key copies. Application developers can use this library to verify the signatures with ease. Detailed instructions for users and site operators can be found at http://irl.cs.ucla.edu/key-publishing.html.

The root key will be published under name "`/ndn/keys/SHA256(RootKey)`" (self-signed) and can be obtained using other out-of-band methods (currently it is bundled with key verification library). The corresponding private key will be maintained by one or more NDN testbed root operators (root-key-admin@named-data.net).

Root operators are responsible to sign and publish site keys as per request of the site operator (using out-of-band method to ensure the request indeed comes from the claimed site operator).

The site operators download the key signing and publication app, configure it with their respective site signing key and institute name. When requested by a user of their site, the operator signs and publishes the user's key (also needs to verify the identify of the user by out-of-band method).

The published keys will be automatically synced to the repositories on all the sites. By default, a sync slice for name prefix "`/ndn/keys`" would be automatically created to synchronize all keys in the NDN test bed when a site operator runs the app for the first time.

Furthermore, there will be a simple tool named "pem" to let users extract the public key in openssl "PEM" format [2] from the keystore for CCNx. Users can send the extracted public key file to the site operators.

The key signing and publication code is available at:

http://github.com/zhenkai/mkey.

The key verification library code is available at:

http://github.com/zhenkai/vkey.

VII. Use Case

The simple use case of this system, the verification of the signature of a packet, is straightforward. In this section we describe a slightly complex use case of the key management system, which also utilizes the meta-info of the keys, to securely collect encryption public keys from eligible participants of a private conference.

In ACT [3], a private conference organizer needs to know participants' (public) encryption keys so that he/she can securely distribute conference decryption key to the legitimate participants only. However, the conference organizer is required to obtain all keys of the participants beforehand through offline method, which is inconvenient and inefficient.

With the help of the key management system in the test bed, we could enable a conference organizer to learn participants' encryption keys incrementally when the participants join. In this way, an organizer no longer needs to do the cumbersome work of collecting keys manually, and participants can be added on the fly with ease.

A participant P sends out an Interest /conference-prefix/name-of-P's-encryption-key to join the conference. The organizer fetches P's encryption key content object at /name-of-P's-encryption-key. The content object is signed by P's signing key, and contains a keylocator that points to P's signing public key. Hence, the organizer can fetch P's public key, verify it by following the trust chain from root and then verify the signature of the encryption key content object. Besides, organizer can also learn P's real world identity by fetching the meta-info for P's public key, which is signed by the site key. By now, the organizer is assured that the signature for P's public encryption key is trustworthy and that the real-world identity of P is confirmed by some site operator.

A dialog can then pop up to display the meta-info for participant P, and the organizer (the person) can decide whether to let P in based on the meta-info. If the organizer decides to let P in, it will reply P's Interest with the conference decryption key encrypted by P's public encryption key. P can then learn about symmetric sessions and join the conversation in the conference.

References

[1] R. Housley, W. Ford, W. Polk, and D. Solo, "Internet X.509 public key infrastructure certificate and CRL profile," RFC2459, 1999, http://www.ietf.org/rfc/rfc2459.txt.

[2] "OpenSSL project," *http://www.openssl.org/*.

[3] Z. Zhu, P. Gasti, Y. Lu, J. Burke, V. Jacobson, and L. Zhang, "A new approach to securing audio conference tools," *AWFIT*, 2011.