The KeeeX Identity Scheme

Understanding KeeeX Identity Providers and Certificates

Authors: **the KeeeX Team**

1. Abstract

The standard trust model for signing entities uses certificate hierarchies, whereby a root certificate owned by a central trusted tier signs the keys of sub certificate tiers, which ultimately results in delivering signed public keys to final users. This document aims at providing a more flexible infrastructure that enables entities to become their own source of trusted signatories, by leveraging their existing use of TLS. Indeed TLS already builds upon certificates emitted by trusted tiers, recognized by operating systems. This paper presents a technical solution : two tools essential to help entities publish trusted lists of signing addresses entitled to represent them, without necessarily relying upon a centralized trusted tier (via a root certificate authority for instance) for that part of the scheme.

Table of Contents

2. Introduction

The standard trust model for signing entities uses certificate hierarchies, whereby a root certificate owned by a central trusted tier signs the keys of sub certificate tiers, which ultimately results in delivering signed public keys to final users.

This concept enforces a trust model where a centralizing actor (not necessarily themselves a root certificate producer) commits on a list of "trusted" root certificate authorities. For instance a web browser knows a list of acceptable root certificate authorities required to establish SSL/TLS connections.

The complexity of dealing with the necessary enrollment and KYC verifications in this model yields a usage restricted to company level : the same key is used to sign large numbers of documents produced by a possibly large number of participants. This is so also in distributed equipments like smart phones or cameras where every item possibly embarks the same signing key.

This document aims at providing a more flexible infrastructure that enables entities to become their own source of trusted signatories, by leveraging their existing use of TLS. Indeed TLS already builds upon certificates emitted by trusted tiers, recognized by operating systems.

The idea is simple : every business actor today already has deployed and safeguards their TLS certificates used to secure connections to their website. The care taken to keep their servers away from prying eyes and the their private keys secret is paramount and it parallels the care for preventing their website to be defaced, or used as a proxy for fake information. Modern actors have achieved considerable progress in securing this. If a website publishes under their own responsibility a trusted list of signing keys, this list can be trusted with the same strength as the site itself. The complexity of attacking this is multiple. It involves attacking the servers, the keys and possibly the keys of some users with known reputation. This is hard.

This paper presents a technical solution : two tools essential to help entities publish trusted lists of signing addresses entitled to represent them, without necessarily relying upon a centralized trusted tier (via a root certificate authority for instance).

Of course as a disclaimer KeeeX allows for signing data and documents using X509 certificates. The possibility therefore exists to use "standard" certificate authority management and authorities when signing files.

Files however usually embed several signatures, one of which may be representing the moral person, and others may represent individual signatories and delegators. An individual or a device may even use different identities over time. The required flexibility of managing such granular identities from creation to revocation calls for a different scheme :

- a tool to create a hierarchy of files that deliver certificates for a list of keys under the control of a self declared single authority
- a format for these certificates

This therefore covers the KeeeX model for deploying such self controlled identities and the rationale for the whole scheme plus a lightweight certificate format to associate a name with a keypair address. Part of this specification mimics the general PKI infrastructure based on X509 certificates, but is trimmed down for simplicity.

After the current introduction (part 1), part 2 introduces the KeeeX HTTP identity provider, and part 3 introduces the format. Part 4 concludes.

3. KeeeX HTTP identity provider

This tool is used to create a hierarchy of files to be statically served by a HTTP server, providing certificates for a list of keys under the control of a single authority. Certificate files conform to the format described in the next chapter.

3.1. Basic design

The program is a simple CLI tool that take as input a JSON containing all "identities" to handle, and output the file hierarchy to serve with an HTTP server.

The tool also maintains its internal state using an encrypted JSON file to keep track of generated files for existing entities when it is re-run.

3.1.1. Input

The input file has a structure similar to this:

```
{
  "authority": {
    "subject": { // subject as decribed in certificate.md  },
  },
  "identities": {
    "entityName1": {
      "subject": { // subject as described in certificate.md },
      "address": "entity key's address (if not provided will generate a
keypair)",
    },
    "entityName2": {
      "subject": { // subject as described in certificate.md },
      "address": "entity key's address (if not provided will generate a
keypair)",
    },
  },
  "parentAuthority": "https://parent.domain", //optional
  "subAuthority": [ // optional
    "https://other.domain"
  ]
}
```

where multiple files can possibly be merged to make handling this easier.

Entity names must not change between updates; it is advised to use simple identifier-like names.

To operate, the authority's keypair must be unlocked. It can come from:

- a file containing an encrypted JsKeys export, in which case the password is read from stdin
- a file containing a passphrase
- reading the passphrase from stdin

When first started, the program suggests to automatically generating a random key and asking for a password to save it.

3.1.2. State database

The state database stores :

- The list of issued certificates
- The list of revoked certificates (entities that got removed from the list earlier)
- The private key for entities with automatically generated keypairs

For convenience, database states are stored in a JSON:

```
{
  "issued": {
    "address1": {
      "subject": { // subject },
      "idx": "<certificate idx>",
      "privateKey": "private key in whatever format it is outputted from
jskeys",
      "status": "ok"
    },
    "address2": {
      "subject": { // subject },
      "idx": "<certificate idx>",
      "status": "revoked",
      "revokationDate": <timestamp in ms>
    }
  },
  "authority": {
    "subject": { // subject },
    "fromKeeeX": false
  }
}
```

This file is never written in plaintext (as it may contain private keys) but encrypted using the authority's keypair.

3.1.3. Output files

For convenience, each output files indicated below are relative to the "root" of a `.well-known` namespace called `keeex/identity`. This means that a file named `test.txt` would in fact be located at the following path relative to the configured output:

```
.well-known/keeex/identity/test.txt
```

3.2. Certificates

The certificate hierarchy uses sharding to store data relative to key addresses. Sharding is done by splitting two words of 4 characters from the beginning, then the address. Example for address "1234ABCD5678EFGH": `certificate/1234/ABCD/5678EFGH.json`.

Each certificate file has the authority's subject as issuer and signer. Identity's certificate's issuer is set using the authority's address as reference, not by embedding the authority's

certificate, as it can change.

The authority itself issues her own self-signed certificate. The strength of such a self signed certificate stems from the fact that it is served by a server trusted via TLS, which represents the key property of the presented architecture.

Optionally the tool may ask its parent authority if there is a certificate with the same address available. If so, this certificate is used instead of the self-signed one. This certificate is stored at both the address path and `certificate/authority.json`.

3.3. Status

When an identity is removed from the list, a revokation is issued. Otherwise the issued certificates remain valid.

The certificate's status is stored in `status/<sharded IDX>.json` (is uses the certificate's IDX). The sharding takes the first 3 words as separate subdirectories, and uses the remaining words as-is.

This file's content is as follows :

```
{
  "status": "ok",
  "address": "<certificate's key's address>",
  "subject": "<certificate's IDX>",
}
```

The status can be "revoked", in which case a "revokationDate" property is added (a numeric timestamp).

In addition, a file named `status/service.json` is also created with only `{"status": "ok"}`.

All these status file are also keeexed and signed by the authority when created.

3.4. Private keys

As a side service, the tool can generate keypairs for entries that do not have one. This is an insecure feature, provided only as a convenience; it is advised to generate keys elsewhere, either client-side or within controlled environments.

To avoid storing private keys unencrypted on storage, retrieving a generated keypair is done per entity and output to the standard output as the JSON export of the key.

The generated keys are bitcoin based.

3.4.1. KeeeX Signature/sub authority

Authorities can be linked as sub/parent. When generating certificates, the parent authority can be checked if it has an updated certificate. It can also check sub authorities; if their subject changed, a confirmation is asked to the user and if valid a new certificate for said authority is generated.

A notification system should be set in place to notify someone of new sub authority subject when this tool is run as a batch.

4. KeeeX Identity Certificates

This part of the document details the syntax of the data structures used to publish verifiable identities. The protocol uses Bitcoin key-pairs, where the public key can be verified after signature verification against a public "address". This allows for not publicly revealing public keys until a signature is actually met which guards against some security risks.

A certificate links a key address to a subject, and is issued by another entity.

The general structure is:

```
{
  "type": "certificate",
  "version": 1,
  "subject": {},
  "key": {
    "type": "bitcoin",
    "address": "…",
  },
  "issuer": {},
}
```

This JSON object is keeexed with a signature matching the issuer.

4.1. Components

4.1.1. Subject

A subject is a JSON object with at least a `name` property which is a string displayed instead of the associated key address. The subject can have many more properties, but the following ones are understood by KeeeX verifier and displayed in a special manner:

- url (string): The URL to a website representing the entity
- imageUrl (string): The URL to an image representing the entity
- imageB64 (string): the base64 representation of an image representing the entity. This should be limited to 2kB data, or roughly 2.8kB of b64 data.

A full example would be:

```
{
  "name": "KeeeX SAS",
  "url": "https://keeex.me",
  "imageUrl": "https://keeex.me/wp-content/uploads/Logo-13.11.2020_kx_xizak-
gozag.png"
}
```

4.1.2. Key

The key object represents the keypair associated with this certificate's subject. Keys are identified by two fields: their type and their "address". Here, address is a term based on the "address" used in some cryptographic schemes (in particular used in some blockchain signatures). Generally speaking the address of a key provides enough public information to validate a digital signature and can be seen as a less detailed view of a public key.

4.1.3. Issuer

The issuer of a certificate can be described in three ways. All issuer's definitions include an optional `revokeUrl` property. If present, it is handled in the way described below.

4.1.3.1. Self-signed certificate

In a self-signed certificate, the issuer is the subject. In that case, the `issuer` property should be:

```
{
  "type": "self",
  "revokeUrl": ""
}
```

4.1.3.2. Signed by another certificate

If the certificate's issuer has its own certificate, the `issuer` property should have this form:

```
{
  "type": "certificate",
  "certificate": "<issuer's certificate JSON as a string>",
  "certificateAddress": "<issuer's certificate's address>",
  "revokeUrl": ""
}
```

The issuer's certificate is provided as a string, not as a JSON object. This allows further verification, since a certificate is a keeexed file. It is also possible to instead set "certificateAddress" in which case a suitable certificate for this address will be looked up by the appropriate software.

4.1.3.3. Signed by a key directly

It is possible, although not recommended, to emit certificates without a certificate for the issuer. In that case, the `issuer` property should be:

```
{
  "type": "key",
  "key": {
    "type": "bitcoin",
    "address": "<issuer key's address>"
  }
  "revokeUrl": ""
}
```

4.2. Revokation mechanism

An issuer has the option to revoke a certificate. To do so, the certificate must contain the URL to a server that would provide a status information about revoked certificates.

Assuming the URL set in a certificate is named `$REVOKE_URL`, the URLs used to check a certificate status are the following, called in order:

- `$REVOKE_URL/status/service.json` to check the server status.
- `$REVOKE_URL/status/<idx sharded>.json` to check the certificate status.

(sharding take the first 3 words as separate subdirectories, and use the remaining words as-is)

In that case, the certificate's IDX is used to retrieve this certificate's status, which should return a suitable status object:

```
{
  "status": "ok",
  "address": "<certificate's key's address>",
  "subject": "<certificate's IDX>",
}
```

Both this JSON object and the one returned by the "status" URL are keeexed by the issuer of the certificate currently checked.

Possible status are:

- "ok": certificate is currently valid
- "revoked": certificate was revoked. In that case, an additional "revokationDate" property is present with the timestamp of the revokation date (in milliseconds since epoch).

In case the server isn't "ready", or the certificate status can't be retrieved, applications should display appropriate warning messages to the users.

5. Conclusion

This document helps understand the usage and structure of identity certificates that can be found on websites and that are used by the many occurrences of the KeeeX Verifier (as e.g. on https://s.keeex.me/verify) when verifying files.

Cheerful thanks from the KeeeX team for reading
