Secure Update Framework Client Key Management and Trust Delegation

Geremy Condra Justin Cappos

October 11, 2010

1 Introduction

1.1 Scope

This document specifies the required trust delegation and key management routines for TUF clients.

1.2 Relationship to Other Documents

Much of the behavior specified in this document is partially laid out in the core TUF document. The system's behavior with regard to freeze and replay attacks is covered in the document entitled "Software Update Security Framework: Client Library Replay and Freeze Attack Protection". The repository-side counterpart to this document contains a large amount of information on the response that subsystem will demonstrate in many of the same circumstances.

2 Overview

The Update Framework is a Python library designed to allow software developers to safely, securely, and easily update clients running their software. In particular, it focuses on the issues of timeliness of data, rapid recovery from a key compromise, and ensuring the authenticity and integrity of installed updates. This document describes the behavior the client must demonstrate in order to provide those properties.

3 Example

The examples used throughout both this and its companion repository-side document are designed to be easy to reproduce, both to verify TUFs behavior and to emulate it. The following subsections demonstrate how to set up a small but fully functional TUF system in which to do so.

3.1 Setting up the Repository

You'll need to run the following steps to set the stage:

```
#! /bin/sh

# create the relevant directories
mkdir tufdemo
cd tufdemo
mkdir demorepo
mkdir demoproject

# add a file to the project
echo "#! /usr/bin/env python" > demoproject/helloworld.py
echo "print 'hello, world!'" >> demoproject/helloworld.py

# run the quickstart script
quickstart.py -t 1 -k keystore -l demorepo -r demoproject
```

This will prompt you for a password for your keystore and an expiration date. Choosing your expiration date is something of a balancing act: on the one hand, you want to make sure that all your clients have had a chance to update before your keys and metadata expire, but on the other hand you want to choose a short time so that keys you revoke expire quickly. A range of one to six weeks is likely to be reasonable for most applications.

After running this and choosing an expiration date, you'll see that it has created an encrypted keystore and a repository for you to use, and that the repository's contents match those of the demo project we created.

3.2 Running the Server

To actually perform an update out of this, you'll need to run a web server through which the client can access the files. Fortunately, Python comes with an easy-to-use module to do this for you:

```
cd demorepo
python -m SimpleHTTPServer 8001
```

3.3 Setting up the Client

TUF isn't designed as a replacement for package managers so it doesn't provide a mechanism with which to perform the initial installation of our demo project's metadata. To do that, open up another terminal and run the following:

```
#! /bin/sh

```

```
mkdir democlient
cp -r demorepo/meta democlient/cur
cp -r democlient/cur democlient/prev
```

Once we've installed our metadata, getting the software is a simple matter of running the demonstration client, found with TUF's source at examples/example_client.py.

4 Basic Client Behavior

When trying to update, the goal of the client is to efficiently obtain the most up-to-date legitimate version of the package. Doing that means three things: first, that the client has to be able to get enough metadata from the repo to determine which files to update, second that it has to be able to verify that metadata, and third that it needs to be able to verify the files once it receives them.

To start with, TUF downloads the timestamp.txt file, which tells it the last time an update was made. To verify it, we pull the last known good public key for the timestamp role out of our copy of root.txt. Assuming that the repo has updated since the last time we did, TUF will continue by downloading the release.txt metadata file and, like timestamp.txt, verifying it against the release role key stored in our copy of root.txt. When combined with the timestamp metadata, the release file will allow us to determine if we're receiving the appropriate version of the other metadata files.

Since we now have enough verified data to ensure that we're getting the proper version of the rest of our metadata, we can go ahead and obtain and verify the root.txt metadata file. As we've already seen, this stores metadata about both other roles and their keys, and as we'll see later, this is also how we handle key revocation.

Assuming everything else has checked out, we can now download targets.txt, which allows us to determine which target files (aka, non-metadata files) we will need to update. Since we have all the hashes of all the target files and know that those hashes are authentic, up-to-date, and valid, we can fetch the matching files and complete the update.

If, at any point in the process, we cannot verify a file against its signature or if it hashes incorrectly, the update process will terminate with an error. This signature verification process is positive in that the existence of a threshold of signatures from the appropriate role is both necessary and sufficient for a signature to be valid.

5 Key Storage

As we've seen, the proper operation of a TUF client only depends on the ability to verify that any results it obtains when polling the server or mirrors have been signed by all of the necessary keys. Since this only requires the use of public keys, client key management reduces to the task of properly associating roles with their keys. In TUF, the mechanism for doing so is via its metadata files, and especially root.txt.

5.1 root.txt

This metadata file is responsible for storing all the trusted keys for TUF, along with the key metadata needed to do routine key management. It must be located at the base URL of the repository's metadata files and signed by the root role's key.

5.1.1 Format

The format of root.txt is as follows:

```
{ "_type" : "Root",
          "ts" : TIME,
          "expires" : EXPIRES,
          "keys" : {
              KEYID : KEY
              , ... },
          "roles" : {
              ROLE : {
                "keyids" : [ KEYID, ... ] ,
                "threshold" : THRESHOLD }
              , ... }
}
```

The format of each element in the above should be consistent with that described in the TUF specification sections 4.1, 4.2, and 4.3, which is to say that the TIME and EXPIRES values should be in "YYYY-MM-DD HH:MM:SS" format, KEYID is a 64 character hexadecimal string, and THRESHOLD is a (normally small) integer.

If the current date is past that specified in EXPIRES, the update process should end with an error. The same is true of all other metadata in TUF.

The KEY value should be of the following format:

```
{ "keytype" : KEYTYPE,
  "keyval" : KEYVAL }
```

where KEYTYPE is a string signifying the encryption primitive (e.g., 'rsa') and KEYVAL is a canonical JSON mapping specifying the appropriate key parameters for that primitive.

The ROLE value should be one of 'root', 'release', 'targets', 'timestamp', or 'mirrors'.

5.1.2 Verification and Validation

In addition to validation of the format as specified above, the client library is required to perform the following checks upon receiving an updated root.txt:

1. Check the 'ts' field against the current time and date to ensure that they do not replace this file with an older version. This is to ensure that an attacker can't replay metadata from before a compromised key was revoked.

2. Verify that each of the top level roles is correctly specified in the "roles" field, with the exception of the optional "mirrors" role.

3. Verify that each keyid matches its respective key and is unique.

In addition to the above, the client library must also take care not to trust a root.txt past its expiration time and to ensure that keys specified in it meet algorithm-specific standards of safety. **Clients must not be allowed trust or install an improperly signed root.txt.** Doing so would allow an attacker to forge arbitrary updates, effectively removing all of TUF's security properties.

6 Trust Delegation

Trust delegation (the process of a trusted user allowing another user to share some or all of their privileges) is built into TUF's trust model, particularly with respect to the Target role. The goal of this delegation mechanism is to make it possible for individual developers to be trusted on some portion of a project, but not the project as a whole. Note that since authorization is positive in TUF, if a valid signature for an update exists under any role with permission over it that update will be accepted as valid.

6.1 targets.txt

The targets.txt file, signed by the target role's key, is responsible for providing the mechanism for trust delegation. It must be located at the repository's metadata base URL and have the following format:

```
{ "_type" : "Targets",
  "ts" : TIME,
  "expires" : EXPIRES,
```

```
  "targets" : TARGETS,
  ("delegations" : DELEGATIONS)
}
```

The TIME and EXPIRES fields are formatted as for the root.txt format.

The TARGETS value should be a list of elements in the following format:

```
{ TARGETPATH : {
      "length" : LENGTH,
      "hashes" : HASHES,
      ("custom" : { ... }) }
  , ...
}
```

Where LENGTH is an integer and HASHES is a list of the cryptographic hashes of the path's destination. The 'custom' field's contents are application-specific and have no impact on the delegation behavior.

The DELEGATIONS part of the targets.txt file points to a list of items formatted like so:

```
{ "keys" : {
      KEYID : KEY,
      ... },
  "roles" : {
      ROLE : {
        "keyids" : [ KEYID, ... ] ,
        "threshold" : THRESHOLD,
        "paths" : [ PATHPATTERN, ... ] }
      , ... }
}
```

With the exception of PATHPATTERN, all the variables seen here are formatted identically to the variables of the same names in the previous listings. PATHPATTERN itself represents either a literal path in UNIX format or a path with ending with a wildcard represented by '/**'.

6.2 Example

We can use the tools built into TUF to see how the above translates into a full targets.txt. While in the demorepo/meta directory created by our first script, we can use signercli.py like this:

```
cd ../..
signercli.py delegate --keystore=keystore ROLE KEYID PATH
```

to modify our default targets.txt to add a delegated role named ROLE associated with KEYID that has permission to modify elements in PATH. Here's the (scrubbed) output:

```
{"signatures": [{
                "keyid": KEYID,
                "method": "sha256-pkcs1",
                "sig": SIGNATURE
                }],
         "signed": {
                        "_type": "Targets",
                        "expires": EXPIRES,
                        "targets": {
                                PATH: {
                                        "hashes": {
                                                "sha256": HASH},
                                                "length": LENGTH}
                        },
                        "ts": EXPIRES}
         "delegations": {
                        "keys": {
                                KEYID: {
                                        "keytype": "rsa",
                                         "keyval": {"e": E, "n": N}
                                        }
                                },
                  "roles": {ROLE: {"keyids": [KEYID],
                                   "paths": [PATH],
                                   "threshold": 1}
                  }
         }
}
```

6.3 Delegated Targets Metadata

Delegated targets metadata is stored in /targets/ROLE.txt, where ROLE is the name of the role to be delegated. This file must be signed by that role and be formatted identically to the top level targets.txt file. Hierarchically delegated trust is, appropriately, handled hierarchically- if DELEGATED_ROLE delegates trust to ANOTHER_ROLE, then the metadata file for ANOTHER_ROLE can be found at /targets/DELEGATED_ROLE/ANOTHER_ROLE.txt. We can create a simple example with the following command:

```
signercli.py maketargets \
--keystore=../keystore \
--parentdir=targets \
--keyid=KEYID \
```

```
--rolename=ROLE \
TARGETS
```

And here's the result, stored at targets/ROLE.txt:

```
{
 "signatures": [
  {
   "keyid": KEYID,
   "method": "sha256-pkcs1",
   "sig": SIGNATURE
  }
 ],
 "signed": {
  "_type": "Targets",
  "expires": EXPIRES,
  "targets": {
   PATH: {
    "hashes": {
     "sha256": HASH
    },
    "length": LENGTH
   }
  },
  "ts": TIMESTAMP
 }
}
```

7 Key Revocation

Key revocation in TUF falls into one of three cases:

1. Revocation of a delegated target key
2. Revocation of a non-root top level key
3. Revocation of a root key

Revocation of a delegated target key is simple- the key in question is simply removed from the metadata files that delegated to it. Similarly, revoking or replacing a non-root top level key is just a matter of replacing it in root.txt with the new value. For example, suppose that a role ALICE delegates trust to another role EVE. ALICE can then revoke EVE's trust entirely by deleting the targets/ALICE/EVE.txt file and removing the DELEGATIONS data structure from either targets/ALICE.txt (if ALICE is a child of a toplevel role) or from her parent role otherwise. The next time an update is generated, EVE's trust will have been completely revoked. Replacing EVE's key can then be done by

adding her like a new delegation.

Revocation of the root key is only slightly more complex. Merely replacing it would leave older clients unable to update, so the better way is to simply sign with both the new key and the old key until you are confident that all the relevant clients have updated. Once you've done that you can stop signing with the old key.

8 Future Work

In the future, the format for keys may be opened to support OpenSSL-style keys. Support for skewed clocks may also be added as noted in the core TUF spec, since many clients seem to be operating under substantial clock drift. Support for automatically integrating TUF with other projects using distutils is also a potential future direction.