DEPRECATED AS OF 2/21/19.
Please go to the Uptane Standards Document at
https://uptane.github.io/uptane-standard/uptane-standard.html

Uptane Implementation Specification (v2019.02.21)

Disclaimer

This is a living document, in the sense that breaking changes may arrive in the near future. Therefore, if you start an implementation based off the current state of the document, please be advised that you may need to reimplement some of your code as these changes occur. We are releasing this document now so that you may begin to familiarize yourself with the larger pieces of implementation. We hope to make the specification stable in the near future, after which point, any breaking changes should be far and few in between.

The views and conclusions contained herein are the authors' and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the US Department of Homeland Security (DHS) or the US government.

1. Introduction

This is the first of two documents designed to facilitate the design, implementation, and deployment of Uptane, a secure software update framework for automobiles. This document, the *Implementation Specification*, is intended to be read by the programmers of an OEM who wish to implement Uptane. Such an implementation will help ECUs on a vehicle to download, distribute, and verify software updates from an OEM repository in a compromise-resilient manner.

The keywords "MUST," "MUST NOT," "REQUIRED," "SHALL," "SHALL NOT," "SHOULD," "SHOULD NOT," "RECOMMENDED," "MAY," and "OPTIONAL" in this document are to be interpreted as described in RFC 2119. In order for an implementation to be considered **"Uptane-compliant"**, an implementation MUST follow all of these rules as specified in the document.

Whenever you see text in berry red, please be advised that this text describes a particular way to implement a feature, and is included to make the discussion more concrete.

DEPRECATED AS OF 2/21/19.
Please go to the Uptane Standards Document at https://uptane.github.io/uptane-standard/uptane-standard.html

However, an OEM is free to use other techniques to implement the same feature, as long as the behavior is the same.

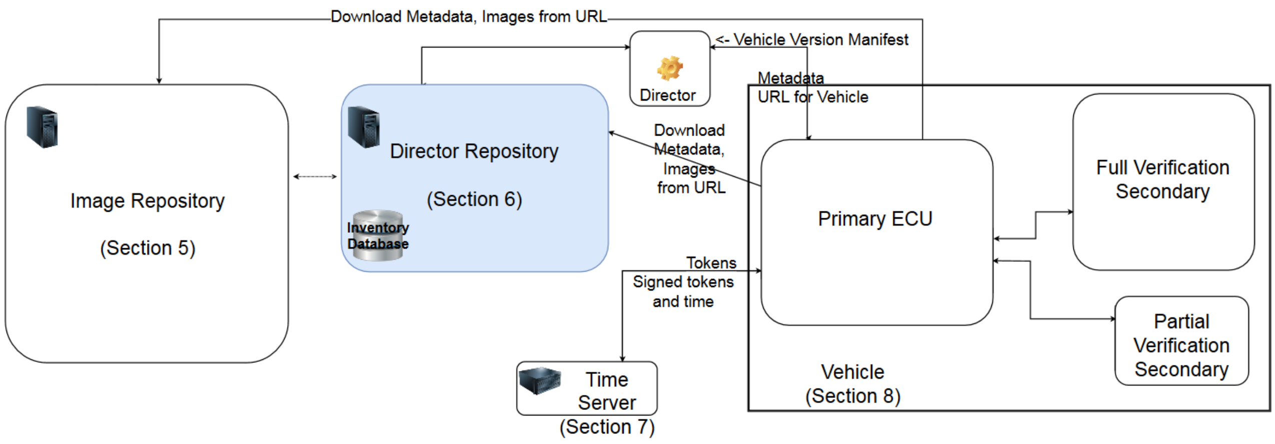

Figure 1a: A "big picture" overview of what needs to be implemented for an Uptane-compliant system.

1.1 Backend servers

A *repository* contains metadata and images (Section 2). An *image* is an archive that contains code and / or data necessary for an ECU to function. (Section D.1 of the Deployment Considerations document discusses how an OEM and its suppliers MAY use an image to arbitrarily update any code and / or data on an ECU.) *Metadata* describe higher-level information, such as the cryptographic hashes and file lengths of images or even other metadata, that can be used to verify that images are up-to-date and have not been altered since being uploaded to the repository (Section 3). Different *roles* produce different pieces of metadata on a repository (Section 2). The OEM MAY use *repository tools* to generate signed metadata, and add it to a repository (Section 4). An OEM SHALL include two different types of repositories.

1.1.1 Image repository

The OEM SHALL build and run an *image repository*, which MAY contain unencrypted images, and SHALL contain image metadata
signed and uploaded by the OEM and / or its suppliers (Section 5).

1.1.2 Director repository

The OEM SHALL also build and run a *director repository*, which contains metadata about which images should be installed by ECUs (Section 6).

Unlike the image repository, the director repository: (1) produces metadata on demand, (2) does instruct ECUs as to which images should be installed, and (3) MAY encrypt images per ECU.

1.1.3 Time server

Finally, the OEM SHALL build and run a *time server*, which is not a repository of metadata and images, but an online server that signs the latest time on demand for ECUs (Section 7).

1.2 Software on the vehicle

The OEM and / or its suppliers SHALL modify software on its ECUs to download and verify the time, metadata, and images before installing a new image (Section 8).

ECUs fall into one of two categories. *Primaries* inform the director repository about which images have been installed by ECUs on the vehicle. They also download and verify the latest time, metadata, and images. *Secondaries* download and verify the latest time, metadata and images from primaries. To do so, primaries and secondaries SHALL follow the steps in Section 8.1.

Before installing a new image, primaries and secondaries SHALL verify the latest downloaded time, metadata, and images. To do so, they SHALL follow the steps in Section 8.2.

Depending on whether an ECU is critical to the operational security of a vehicle, and whether it has enough speed and memory, an ECU SHOULD perform either *partial* or *full verification* of metadata. Partial verification means that the director repository is the only root of trust about the image to be installed on an ECU, whereas full verification means that both the image and director repositories must agree about the image to be installed. Primary and safety-critical secondary ECUs that are updated over-the-air MUST use full verification. Non-safety-critical secondary ECUs that are updated over-the-air SHOULD use at least partial verification; if resources permit, they SHOULD perform full verification. If they are incapable of performing even partial verification, then they SHOULD NOT be updated over-the-air.

2. Repositories and roles

In order to read and write metadata, it helps to understand that all metadata on a repository is produced by one of four basic *roles*: *root*, *targets*, *snapshot*, and *timestamp*. The relationships between these roles are illustrated in Figure 2a, whereas Table 2a summarizes their respective responsibilities. To better understand the design behind these roles, the reader should read the CCS'10 paper on The Update Framework (TUF), a precursor to Uptane.

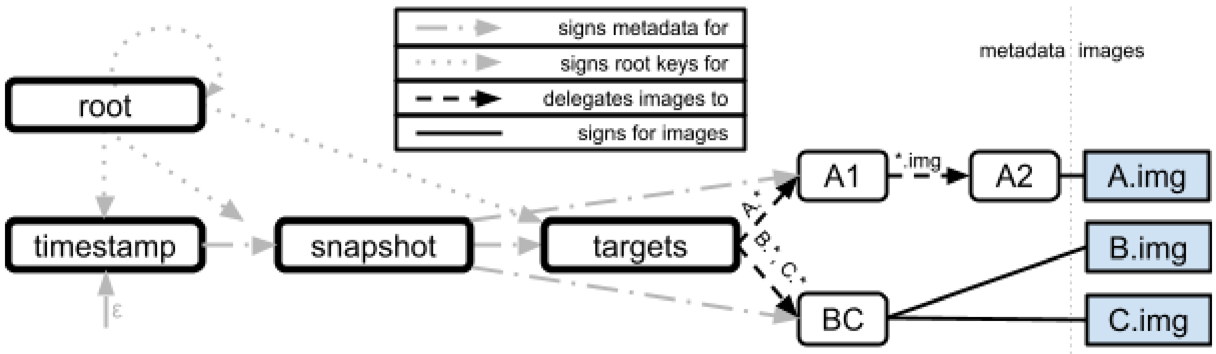

Figure 2a: The four basic (root, timestamp, snapshot, and targets) roles used in Uptane, along with the corresponding metadata and images.

Role	Purpose
root	Serves as the certificate authority for the repository. Distributes and revokes the public keys used to verify the root, timestamp, snapshot, and targets role metadata.
timestamp	Indicates whether there is any new metadata or image on the repository.
snapshot	Indicates which images have been released at the same time by the repository.
targets	Indicates metadata, such as the cryptographic hashes and file sizes of images. May delegate this responsibility to other, custom-made roles.

Table 2a: The four basic roles that Uptane uses to add signed metadata to a repository.

2.1 The root role

The *root* role serves as the certificate authority. It distributes and revokes the *public keys* used to verify metadata produced by each of the four basic roles (including itself).

2.2 The targets role

The *targets* role provides metadata, such as hashes and file sizes, about images.

2.2.1 Delegations

Instead of signing metadata itself, the targets role MAY *delegate* the responsibility of signing this metadata to other, custom-made roles. For example, in Figure 2a, the targets role has delegated all images that match the filename pattern “A.*” to the A1 role, and all images that match the filename patterns “B.*” and “C.*” to the BC role. In turn, the A1 role delegates a subset of its images (in this case, only the “A.img”) to the A2 role. A delegation binds the public keys used by a delegatee to a subset of the images these keys are trusted to sign. This means that the targets role distributes and revokes the public keys for the A1 and BC roles, whereas the A1 role does the same for the A2 role.

There are two types of delegations: prioritized and / or terminating delegations. Furthermore, a delegation may require multiple roles, as explained below.

2.2.1.1 Prioritized delegations

Sometimes an image may be delegated to more than one role. For example, the targets role could delegate all images to both A1 and BC. In this case, if both A1 and BC sign metadata about the same image, it may be unclear to a client which role should be trusted. In order to solve this problem, all delegations in Uptane are *prioritized*. Returning to the example, the targets role lists its delegations such that A1 is prioritized over BC. Thus, if A1 signs metadata about some image, its metadata is trusted over BC. For more information, the reader should read the NSDI'16 paper on prioritized and terminating delegations.

2.2.1.2 Terminating delegations

In other cases, it is desirable for a role X to delegate an image to role Y such that, if Y (or any of its own delegations) has not signed any metadata about the image, then the client SHALL NOT search the rest of the delegations in role X for this image. Using a *terminating delegation* from X to Y lets X endow this delegation with precisely this meaning. In Uptane, delegations SHOULD NOT be terminating by default, unless stated otherwise. For more information, the reader should read the NSDI'16 paper on prioritized and terminating delegations.

2.2.1.3 Multi-role delegations

There may be occasions where multiple roles may be required to sign the same metadata about the same image. For example, a supplier may require both its development and release engineering teams to sign off on all images. Using *multi-role delegations*, the supplier delegates all images to a combination of its development and release engineering roles, so that a client installs an image only if both roles have signed the same metadata about it. For more information, the reader should read TAP 3.

2.2.2 Delegated targets roles

Although the targets role is necessary, Uptane does not force any particular model of delegations on an OEM. The Deployment Considerations document discusses some useful models for delegations in deployment scenarios.

2.2.2.1 The supplier roles

An OEM MAY use the targets role to delegate the signing of images for an ECU to the supplier that develops and maintains those images. There MAY be as many of these roles as there are suppliers.

2.3 The snapshot role

The *snapshot* role indicates which images have been released by the repository at the same time. It does so indirectly, by signing metadata about all targets metadata files released by the repository at the same time.

2.4 The timestamp role

The *timestamp* role indicates whether there are any new metadata or images on the repository. It does so indirectly, by signing metadata about the snapshot metadata file.

2.5 The map file

The OEM SHALL use *the map file* to specify that all images must be signed by both the image and director repositories. As discussed in Section A.2 of the Deployment Considerations document, every primary and full verification secondary is supplied with a copy of this map file during manufacture. The contents of this file is specified in Section 3.8 of this document. For more information about the map file, the reader should read TAP 4.

3. Metadata abstract syntax

Metadata files on a repository SHOULD be written using the ASN.1 abstract syntax specified in this section. (The ASN.1 modules may be found in this GitHub repository.) These files MAY be encoded and decoded using any transfer syntax that an OEM desires (e.g., BER, CER, DER, JSON, OER, PER, XER).

3.1 Common data structures

Metadata files SHOULD share the data structures in Table 3.1a. These data structures specify how information, such as cryptographic hashes, digital signatures, and public keys, should be encoded.

```
CommonModule DEFINITIONS AUTOMATIC TAGS ::= BEGIN

  EXPORTS ALL;

  RoleType        ::= ENUMERATED {root, targets, snapshot, timestamp}

  -- String types.
  Filename        ::= VisibleString (SIZE(1..32))
  -- No known path separator allowed in a strict filename.
  StrictFilename  ::= VisibleString (SIZE(1..32))
                                    (PATTERN "[^/\\]+")
  BitString       ::= BIT STRING    (SIZE(1..1024))
  OctetString     ::= OCTET STRING  (SIZE(1..1024))
  HexString       ::= VisibleString (SIZE(1..1024))
                                    (PATTERN "[0-9a-f]+")
  -- Table 1 of RFC 4648.
  Base64String    ::= VisibleString (SIZE(1..1024))
                                    (PATTERN "[A-Za-z0-9\+/=]+")
  -- Adjust length of SEQUENCE OF to your needs.
  Paths            ::= SEQUENCE (SIZE(1..8)) OF Path
  Path             ::= VisibleString (SIZE(1..32))
                                  (PATTERN "[\w\*\\/]+")
  -- Adjust length of SEQUENCE OF to your needs.
  URLs             ::= SEQUENCE (SIZE(0..8)) OF URL
  URL              ::= VisibleString (SIZE(1..1024))
  -- A generic identifier for vehicles, primaries, secondaries.
  Identifier       ::= VisibleString (SIZE(1..32))

  Natural          ::= INTEGER (0..MAX)
  Positive         ::= INTEGER (1..MAX)
  Length           ::= Positive
  Threshold        ::= Positive
  Version          ::= Positive
  -- The date and time in UTC encoded as a UNIX timestamp.
  UTCDateTime      ::= Positive

  BinaryData       ::= CHOICE {
    bitString      BitString,
    octetString    OctetString,
    hexString      HexString,
    base64String   Base64String
  }

  -- Adjust length of SEQUENCE OF to your needs.
  Hashes           ::= SEQUENCE (SIZE(1..8)) OF Hash
  Hash             ::= SEQUENCE {
    function       HashFunction,
    digest         BinaryData
  }
  HashFunction ::= ENUMERATED {sha224, sha256, sha384, sha512, sha512-224,
                              sha512-256, ...}

  -- Adjust length of SEQUENCE OF to your needs.
  Keyids           ::= SEQUENCE (SIZE(1..8)) OF Keyid
  -- Usually, a hash of a public key.
  Keyid            ::= HexString

  -- Adjust length of SEQUENCE OF to your needs.
  Signatures       ::= SEQUENCE (SIZE(1..8)) OF Signature
  Signature        ::= SEQUENCE {
    keyid          Keyid,
    method         SignatureMethod,
    -- For efficient checking, sign the hash of the message instead of the
    -- message itself.
    hash           Hash,
    -- The signature itself.
    value          HexString
  }
  SignatureMethod ::= ENUMERATED {rsassa-pss, ed25519, ...}

  -- Adjust length of SEQUENCE OF to your needs.
  PublicKeys       ::= SEQUENCE (SIZE(1..8)) OF PublicKey
  PublicKey        ::= SEQUENCE {
    publicKeyid      Keyid,
    publicKeyType    PublicKeyType,
    publicKeyValue   BinaryData
  }
  PublicKeyType    ::= ENUMERATED {rsa, ed25519, ...}

END
```

Table 3.1a: An ASN.1 module that defines common data structures used by metadata files.

An OEM MAY use any hash function (`Hash.function`; e.g., SHA-2) and signature scheme (`Signature.method`; e.g., RSASSA-PSS, Ed25519). Uptane is not restricted to any particular hash function or signature scheme.

A hash digest (`Hash.digest`), signature (`Signature.sig`), or public key (`PublicKey.keyval`) SHOULD be encoded as either a bit, octet, hexadecimal, or Base64 string. For example, an RSA public key MAY be encoded using the PEM format, whereas an Ed25519 public key MAY be encoded as a hexadecimal string.

Every public key has a unique identifier (`PublicKey.keyid`). This identifier MAY be, for example, the SHA-256 hash of the public key.

An ECU SHOULD verify that a `Keyids`, `Hashes`, `Signatures`, and `PublicKeys` sequence contains unique `KeyId`, `Hash.function`, `Signature.keyid`, and `PublicKey.keyid` values, respectively. For example, it MAY reject a sequence containing duplicate values, or simply ignore such values.

3.2 The metadata format common to all roles

All metadata files SHOULD share the format in Table 3.2a.

```
MetadataModule DEFINITIONS AUTOMATIC TAGS ::= BEGIN

  EXPORTS Metadata;

  -- https://sourceforge.net/p/asn1c/discussion/357921/thread/aced2512/
  IMPORTS Length,
          Positive,
          RoleType,
          Signatures,
          UTCDateTime       FROM CommonModule
          RootMetadata      FROM RootModule
          TargetsMetadata   FROM TargetsModule
          SnapshotMetadata  FROM SnapshotModule
          TimestampMetadata FROM TimestampModule;

  Metadata      ::= SEQUENCE {
    signed                Signed,
    numberOfSignatures    Length,
    signatures            Signatures
  }
  Signed        ::= SEQUENCE {
    type        RoleType,
    expires     UTCDateTime,
    version     Positive,
    body        SignedBody
  }
  SignedBody    ::= CHOICE {
    rootMetadata       RootMetadata,
    targetsMetadata    TargetsMetadata,
    snapshotMetadata   SnapshotMetadata,
    timestampMetadata  TimestampMetadata
  }

END
```
Table 3.2a: An ASN1. module that defines the metadata format common to all roles.

Every metadata file contains three parts: a signed message (`Signed`), the number of signatures on the following message, and a sequence of signatures for the message (`Signatures`).

The signed message is a sequence of four attributes: (1) an enumerated type of the metadata (i.e., root, targets, snapshot, or timestamp), (2) an expiration date and time for the metadata (specified using the ISO 8601 format), (3) a version number, and (4) the role-specific metadata. This version number SHOULD be incremented every time the metadata file is updated. The attributes of role-specific metadata will be discussed in the rest of this section.

Note that signatures are computed over the hash of the signed message, instead of the signed message itself.

3.3 The root metadata

The root metadata, specified in Table 3.3a, distributes and revokes the public keys of the top-level root, targets, snapshot, and timestamp roles. These keys are revoked and replaced by changing the public keys specified in the root metadata. This metadata is signed using the root role's private keys.

```
RootModule DEFINITIONS AUTOMATIC TAGS ::= BEGIN

  EXPORTS RootMetadata;

  IMPORTS Keyids,
          Length,
          PublicKeys,
          RoleType,
          Threshold,
          URLs FROM CommonModule;

  RootMetadata ::= SEQUENCE {
    numberOfKeys  Length,
    keys          PublicKeys,
    numberOfRoles Length,
    roles         TopLevelRoles,
    -- https://tools.ietf.org/html/rfc6025#section-2.4.2
    ...
  }
  -- Adjust length of SEQUENCE OF to your needs.
  TopLevelRoles ::= SEQUENCE (SIZE(4)) OF TopLevelRole
  TopLevelRole  ::= SEQUENCE {
    role            RoleType,
    -- TAP 5: The URLs pointing to the metadata file for this role.
    numberOfURLs    Length OPTIONAL,
    urls            URLs OPTIONAL,
    numberOfKeyids  Length,
    keyids          Keyids,
    threshold       Threshold,
    ...
  }

END
```

Table 3.3a: An ASN.1 module that specifies the body of the root metadata.

The root metadata contains two important attributes. First, the `keys` attribute lists the public keys used by the root, targets, snapshot, and timestamp roles. Second, the `roles` attribute maps each of the four roles to: (1) the URL pointing to its metadata file, (2) its public keys, and (3) the threshold number of keys required to sign the metadata file. An empty sequence of URLs denotes that the metadata file SHALL NOT be updated. An ECU SHOULD verify that each of the four roles has been defined exactly once in the metadata.

The following link points to an example of the root metadata encoded using JSON.

WARNING: Presently, we are working to specify the format of the map file. This is being drafted as TAP 5. We will update the discussion in this section to reflect these changes as they are finalized.

3.4 The targets metadata

At a minimum, a targets metadata file contains metadata (i.e., filename, hashes, length) about unencrypted images on a repository. The file MAY also contain two optional pieces of information: (1) custom metadata about which images should be installed by which ECUs, and whether encrypted images are available, and / or (2) other delegated targets roles that have been entrusted to sign images. This file is signed using the private keys of either the top-level targets role or a delegated targets role. Table 3.4.a specifies all of the RECOMMENDED attributes of the targets metadata.

```
TargetsModule DEFINITIONS AUTOMATIC TAGS ::= BEGIN

  EXPORTS TargetsMetadata, Target;

  IMPORTS BinaryData,
          Filename,
          Hashes,
          Identifier,
          Keyids,
          Length,
          Natural,
          Paths,
          Positive,
          PublicKeys,
          StrictFilename,
          Threshold FROM CommonModule;

  TargetsMetadata ::= SEQUENCE {
    -- Allowed to have no targets at all.
    numberOfTargets Natural,
    -- Metadata about unencrypted images on a repository.
    targets         Targets,
    -- Delegations are optional.
    delegations     TargetsDelegations OPTIONAL,
    -- https://tools.ietf.org/html/rfc6025#section-2.4.2
    ...
  }

  -- Adjust length of SEQUENCE OF to your needs.
  Targets           ::= SEQUENCE (SIZE(1..128)) OF TargetAndCustom
  TargetAndCustom   ::= SEQUENCE {
    -- The filename, length, and hashes of unencrypted images on a repository.
    target  Target,
    -- This attribute is used to specify additional information, such as which
    -- images should be installed by which ECUs, and metadata about encrypted
    -- images.
    custom  Custom OPTIONAL
  }
  Target ::= SEQUENCE {
    filename        Filename,
    length          Length,
    numberOfHashes  Length,
    hashes          Hashes
  }
  Custom ::= SEQUENCE {
    -- NOTE: The following attributes are specified by both the image and
    -- director repositories.
    -- The release counter is used to prevent rollback attacks on images when
    -- only the director repository is compromised.
    -- Every ECU should check that the release counter of its latest image is
    -- greater than or equal to the release counter of its previous image.
    releaseCounter        Natural OPTIONAL,
    -- The hardware identifier is used to prevent the director repository,
    -- when it is compromised, from choosing images for an ECU that were not
    -- meant for it.
    -- Every ECU should check that the hardware ID of its latest image matches
    -- its hardware ID.
    -- An OEM MAY define other types of information to further restrict the
    -- choices that can be made by a compromised director repository.
    hardwareIdentifier    Identifier OPTIONAL,
    -- NOTE: The following attributes are specified only by the director
    -- repository.
    -- The ECU identifier specifies information, e.g., serial numbers, that the
    -- director uses to point ECUs as to which images they should install.
    -- Every ECU should check that the ECU ID of its latest image matches its
    -- own ECU ID.
    ecuIdentifier         Identifier OPTIONAL,
    -- This attribute MAY be used by the director to encrypt images per ECU.
    encryptedTarget       Target OPTIONAL,
    -- This attribute MAY be used if ECU keys are asymmetric, and a per-image
    -- symmetric encryption key is desired for faster decryption of images.
    -- In that case, the director would use the asymmetric ECU key to encrypt
    -- this symmetric key.
    encryptedSymmetricKey EncryptedSymmetricKey OPTIONAL,
    ...
  }
  EncryptedSymmetricKey ::= SEQUENCE {
    -- This is the symmetric key type.
    encryptedSymmetricKeyType   EncryptedSymmetricKeyType,
    -- This is the symmetric key encrypted using the asymmetric ECU key.
    encryptedSymmetricKeyValue  BinaryData
  }
  EncryptedSymmetricKeyType ::= ENUMERATED {aes128, aes192, aes256, ...}

  -- https://github.com/theupdateframework/taps/blob/master/tap3.md
  TargetsDelegations  ::= SEQUENCE {
    -- The public keys of all delegatees.
    numberOfKeys        Length,
    keys                PublicKeys,
    -- The role name, filename, public keys, and threshold of a delegatee.
    numberOfDelegations Length,
    -- A list of paths to roles, listed in order of priority.
    delegations         PrioritizedPathsToRoles
  }

  -- Adjust length of SEQUENCE OF to your needs.
  PrioritizedPathsToRoles ::= SEQUENCE (SIZE(1..8)) OF PathsToRoles
  PathsToRoles ::= SEQUENCE {
    -- A list of image/target paths entrusted to these roles.
    numberOfPaths   Length,
    paths           Paths,
    -- A list of roles required to sign the same metadata about the matching
    -- targets/images.
    numberOfRoles   Length,
    roles           MultiRoles,
    -- Whether or not this delegation is terminating.
    terminating     BOOLEAN DEFAULT FALSE
  }

  -- Adjust length of SEQUENCE OF to your needs.
  MultiRoles ::= SEQUENCE (SIZE(1..8)) OF MultiRole
  MultiRole ::= SEQUENCE {
    -- The rolename (e.g., "supplierA-dev").
    -- No known path separator allowed in a rolename.
    rolename        StrictFilename,
    -- The public keys used by this role.
    numberOfKeyids  Length,
    keyids          Keyids,
    -- The threshold number of these keys.
    threshold       Threshold
  }

END
```

Table 3.4a: An ASN.1 module that specifies the body of the targets metadata.

3.4.1 Metadata about images

At the very least, a targets metadata file MUST contain the `TargetsMetadata.targets` attribute, which specifies a sequence of unencrypted images. An empty sequence is used to indicate that no targets/images are available. For every unencrypted image, its filename, version number, length, and hashes are listed using the `Target` sequence. An ECU SHOULD verify that each unencrypted image has been defined exactly once in the metadata file.

The unencrypted image SHOULD also be associated with additional information using the `Custom` sequence. The following attributes SHOULD be specified by both the image and director repositories. The `Custom.releaseCounter` attribute is used to prevent rollback attacks when the director repository is compromised. The director repository cannot choose images for an ECU with a *release counter* that is lower than the release counter of the image it has currently installed. The `Custom.hardwareIdentifier` attribute is used to prevent a compromised director repository from causing ECUs to install images that were not intended for them. For example, this attribute MAY be the ECU part number. The OEM and its suppliers MAY define other attributes that can be used by ECUs to further restrict which types of images they are allowed to install.

The following attributes SHOULD be specified by the director repository. The `Custom.ecuIdentifier` attribute specifies the identifier (e.g., serial number) of the ECU that should install this image. An ECU SHOULD verify that each ECU identifier has been defined exactly once in the metadata file. If the director repository wishes to publish per-ECU encrypted images, then the `Custom.encryptedTarget` attribute MAY be used to specify metadata about the encrypted images. An ECU MUST then download the encrypted image, check its metadata, decrypt the image, and check its metadata again. Finally, if an ECU key is an *asymmetric* public key, the director repository MAY use a *symmetric* private key to reduce the time used to decrypt the image. To do so, the director repository MAY use the asymmetric ECU key to encrypt, e.g., a private AES symmetric key, and place the encrypted key in the `Custom.encryptedSymmetricKey` attribute.

3.4.2 Metadata about delegations

Besides directly signing metadata about images, the targets role MAY delegate this responsibility to delegated targets roles. To do so, the targets role uses the OPTIONAL `TargetsMetadata.delegations` attribute. If this attribute is not used, then it means that there are no delegations.

The `TargetsDelegations.keys` attribute lists all of the public keys used by the delegated targets roles in the current targets metadata file. An ECU SHOULD verify that each public key (identified by its `Keyid`) has been defined exactly once in the metadata file.

The `TargetsDelegations.delegations` attribute lists all of the delegations in the current targets metadata file. All delegations are prioritized: a sequence is used to list delegations in order of appearance, so that the earlier the appearance of a delegation, the higher its priority. Every delegation contains three important attributes.

The `PathsToRoles.paths` attribute describes a sequence of target/image paths that the delegated roles are trusted to provide. A desired target/image needs to match only one of these paths for the delegation to apply. A path MAY be either to a single file, or to a directory to indicate all files and / or subdirectories under that directory. A path to a directory is used to indicate all possible targets sharing that directory as a prefix; e.g. if the directory is "targets/A," then targets which match that directory include "targets/A/B.img" and "targets/A/B/C.img."

The `PathsToRoles.roles` attribute describes all of the roles that SHALL sign the same non-custom metadata (i.e., filename, length, and hashes of unencrypted images) about delegated targets/images. Every delegated targets role has (1) a name, (2) a set of public keys, and (3) a threshold of these keys required to verify its metadata file.

Note that a role name SHOULD follow the filename restrictions of the underlying file storage mechanism. For example, it may be “director” or "targets/director." As discussed in Section 3.7, the role name will determine part of the actual metadata filename of the delegated targets role. If it is “director” or “targets/director,” then its delegated targets metadata file MAY use the filename “director.ext” or “targets/director.ext,” respectively. However, the role name SHALL NOT use the path separator (e.g., “/” or “\”) if it is a character used to separate directories on the underlying file storage mechanism. In other words, all targets metadata files are implicitly assumed to reside in the same directory. It is safe to use this character in key-value databases or stores that do not have a notion of directories (e.g., Amazon S3).

Finally, the `PathsToRoles.terminating` attribute determines whether or not a backtracking search for a target/image should be terminated.

The metadata file for a delegated targets role SHALL have exactly the same format as for the top-level targets role. For example, the metadata file for a supplier role has precisely the same format as the the top-level targets role.

3.4.3 An example of targets metadata on the image repository

The following link points to an example of a targets metadata file on the image repository, encoded using JSON.

A targets metadata file on the image repository SHALL provide non-custom metadata about unencrypted images, and MAY use delegations.

3.4.4 An example of targets metadata on the director repository

The following link points to an example of a targets metadata file on the director repository, encoded using JSON.

The targets metadata file on the director repository SHOULD provide *custom metadata* (see the `TargetsModule.Custom` sequence) about images, but SHALL NOT use delegations.

3.5 The snapshot metadata

The snapshot metadata lists the version numbers of all targets metadata files on the repository. It is signed using the snapshot role keys, and follows the format specified in Table 3.5a.

```
SnapshotModule DEFINITIONS AUTOMATIC TAGS ::= BEGIN

  EXPORTS SnapshotMetadata;

  IMPORTS Length,
          Hashes,
          StrictFilename,
          Version FROM CommonModule;

  -- Adjust length of SEQUENCE OF to your needs.
  SnapshotMetadata ::= SEQUENCE {
    numberOfSnapshotMetadataFiles Length,
    snapshotMetadataFiles         SnapshotMetadataFiles
  }
  SnapshotMetadataFiles ::= SEQUENCE (SIZE(1..128)) OF SnapshotMetadataFile
  SnapshotMetadataFile ::= SEQUENCE {
    filename  StrictFilename,
    version   Version,
    -- https://tools.ietf.org/html/rfc6025#section-2.4.2
    ...
  }

END
```

Table 3.5a: An ASN.1 module that specifies the body of the snapshot metadata.

The `filename` attribute specifies a metadata file's relative path from the metadata root of a repository, and SHALL NOT contain a path separator.

For example, if the file is to be downloaded using the URL "http://example.com/file/metadata/targets/supplier.ext", then its filename is "supplier.ext".

An ECU SHOULD verify that each filename has been defined exactly once in the snapshot metadata file.

The following link points to an example of a snapshot metadata file encoded using JSON.

WARNING: Presently, we are working to update the format of root metadata, which may, in turn, affect the format of the snapshot metadata. This is being drafted as TAP 5. We will update the discussion in this section to reflect these changes as they are finalized.

3.6 The timestamp metadata

The timestamp metadata specifies metadata (e.g., filename and version number) about the snapshot metadata file. It is signed using the timestamp role keys, and follows the format specified in Table 3.6a.

```
TimestampModule DEFINITIONS AUTOMATIC TAGS ::= BEGIN
```

```
  EXPORTS TimestampMetadata;

  IMPORTS Filename,
          Hashes,
          Length,
          Version FROM CommonModule;

  TimestampMetadata ::= SEQUENCE {
    filename        Filename,
    version         Version,
    length          Length,
    numberOfHashes  Length,
    hashes          Hashes,
    -- https://tools.ietf.org/html/rfc6025#section-2.4.2
    ...
  }


END
```

Table 3.6a: An ASN1.module that specifies the body of the timestamp metadata.

The following link points to an example of a timestamp metadata file encoded using JSON.

3.7 Filename in metadata or on an ECU vs. filename on a repository

There is a difference between the file name in a metadata file or an ECU, and the file name on a repository. This difference exists in order to avoid race conditions, where metadata and images are read from and written to at the same time. For more details, the reader should read PEP 458.

Unless stated otherwise, all metadata files SHALL be written as such to a repository. If a metadata file A was specified as FILENAME.EXT in another metadata file B, then it SHALL be written as VERSION.FILENAME.EXT where VERSION is A's version number (Section 3.2).

For example, if the top-level targets metadata file is referenced as “targets.json” in the snapshot metadata file, it is read and written using the filename “1.targets.json” instead. In a similar example, if the snapshot metadata file is referenced as “snapshot.json” in the timestamp metadata file, it is read and written using the filename “1.snapshot.json” instead. To take a final example using delegations (Section 3.4.2), if the ROLENAME of a delegated targets metadata file is “director,” and it is referred to in the snapshot metadata file using the filename “director.json” and the version number 42, then it is read and written using the filename “42.director.json” instead.

There are two exceptions to this rule. First, if the version number of the timestamp metadata is not known in advance, it MAY also be read from and written to a repository using a filename that is not qualified with a version number (i.e., FILENAME.EXT). As we will see in Section 6, this is the case with the timestamp metadata file on the image repository, but not the director repository. Second, the root metadata SHALL also be read from and written to a repository using a filename that is not qualified with a version number (i.e., FILENAME.EXT). This is because, as we will see in Section 8, the root metadata may be read without knowing its version number in advance.

All target files are written as such to a repository. If a target's metadata file specifies a target file as FILENAME.EXT then it SHALL be written as HASH.FILENAME.EXT where HASH is one of the *n* hashes of the targets file (Section 3.4). This means that there SHALL be *n* different file names that all point to the same target file. Each filename is distinguished only by the value of the digest in its filename.

However, note that although a primary SHALL download a metadata or target file using the filename written to the repository, it SHALL write the file to its own storage using the original filename in the metadata. For example, if a metadata file is referred to as FILENAME.EXT in another metadata file, then a primary SHALL download it using either the filename FILENAME.EXT, VERSION.FILENAME.EXT, or HASH.FILENAME.EXT (depending on which of the aforementioned rules applies), but it SHALL always write it to its own storage as FILENAME.EXT. This implies that the previous set of metadata and target files downloaded from a repository SHALL be kept in a separate directory on an ECU from the latest set of files.

For example, the previous set of metadata and target files MAY be kept in the "previous" directory on an ECU, whereas the latest set of files MAY be kept in the "current" directory.

3.8 The map file

The map file specifies which images should be downloaded from which repositories. In most deployment scenarios for full verification ECUs, this will mean downloading images from both the image and director repositories. It is not signed, and follows the format specified in Table 3.8a.

```
MapFileModule DEFINITIONS AUTOMATIC TAGS ::= BEGIN

  IMPORTS Length,
          Paths,
          StrictFilename,
          URLs FROM CommonModule;

  -- https://github.com/theupdateframework/taps/blob/master/tap4.md
  MapFile ::= SEQUENCE {
    -- A list of repositories.
    numberOfRepositories  Length,
    repositories          Repositories,
    --A list of mapping of images to repositories.
    numberOfMappings      Length,
    mappings              Mappings
  }

  -- Adjust length of SEQUENCE OF to your needs.
  Repositories    ::= SEQUENCE (SIZE(2)) OF Repository
  Repository      ::= SEQUENCE {
    -- A shorthand name for the repository, which also specifies the name of the
    -- directory on the client which contains previous and latest metadata.
    name              RepositoryName,
    -- A list of servers where metadata and targets may be downloaded from.
    numberOfServers   Length,
    servers           URLs,
    -- https://tools.ietf.org/html/rfc6025#section-2.4.2
    ...
  }
  -- Adjust length of SEQUENCE OF to your needs.
  RepositoryNames ::= SEQUENCE (SIZE(2)) OF RepositoryName
  RepositoryName  ::= StrictFilename

  -- Adjust length of SEQUENCE OF to your needs.
  Mappings ::= SEQUENCE (SIZE(1)) OF Mapping
  Mapping  ::= SEQUENCE {
    -- The list of targets delegated to the following repositories.
    numberOfPaths          Length,
    paths                  Paths,
    -- The repositories which MUST all sign the preceeding targets.
    numberOfRepositories   Length,
    repositories           RepositoryNames,
    -- Whether or not this delegation is terminating.
    terminating            BOOLEAN DEFAULT FALSE,
    -- https://tools.ietf.org/html/rfc6025#section-2.4.2
    ...
  }

END
```

Table 3.8a: An ASN1.module that specifies the map file.

The `MapFile.repositories` attribute specifies a list of available repositories. For each repository, a short-hand name, and a list of servers where metadata and targets may be downloaded from, are specified. The short-hand name also specifies the metadata directory on an ECU containing the previous and current sets of metadata files.

The `MapFile.mappings` attribute specifies which images are mapped to which repositories. An OEM MAY map the same set of images to multiple repositories. Typically, an OEM would map all images to both the image and director repositories. See the deployment considerations document for other configurations, especially with regard to fleet management.

For more information about the map file, the reader should read TAP 4.

TAP 4 points to an example of a map file encoded using JSON.

WARNING: Presently, we are working to specify the format of the map file. This is being drafted as TAP 4. We will update the discussion in this section to reflect these changes as they are finalized.

4. Repository tools for writing metadata

As noted in Section 1.1, an OEM SHALL add metadata about images to its repositories. An OEM SHOULD build and use *repository tools*, or software that generate these signed metadata given certain inputs (e.g., keys, other metadata files, images).

The OEM MAY use our reference implementation of the repository tools (compatible with Python 2 and 3) to write these metadata files. Our repository tools encode metadata using JSON.

If the OEM prefers another type of encoding (e.g., BER, CBOR, CER, DER, OER, PER, XER), then it SHOULD fork our reference implementation, and make only the adjustments necessary to support the alternate encoding. Alternatively, it MAY fork other, compatible implementations, such as Notary or go-tuf.

5. Image repository

As noted in Section 1.1.1, an OEM SHALL implement an *image repository* in order for the OEM and / or its suppliers to upload metadata, as well as unencrypted images. It contains metadata about available images for all ECUs on all types of automobiles, but it does not contain instructions as to which images should be installed by which ECUs.

Furthermore, the image repository is largely controlled by human intervention. Metadata and unencrypted images on the image repository are expected to be updated relatively infrequently. For example, as discussed in the Deployment Considerations document, an OEM may release new metadata and unencrypted images every few weeks or months.

As its underlying file storage mechanism, the image repository MAY use a filesystem, or a key-value store / database.

For example, the repository MAY use Amazon S3, Google Cloud Storage, Microsoft Azure Storage, MongoDB, MySQL, PostgreSQL, or Redis to store files.

The exact mechanism does not matter, as long as repository administrators are able to write a file using a unique name, and later read the same file using the same name.

The OEM SHOULD expose a public interface for primaries to download metadata and unencrypted images from the image repository.

For example, the repository MAY make metadata and images available over FTP, FTPS, SFTP, HTTP, or HTTPS.

The OEM SHALL require authentication (e.g., passwords) to control write access to the file storage mechanism. For example, a supplier SHALL be able to write only metadata and images that have been delegated to it. Suppliers and their delegatees upload metadata files similar to the example in Section 3.4.3. The OEM MAY require authentication to control read access. The reader MAY wish to consult implementations of community repositories, such as Docker Hub, PyPI, RubyGems, or Quay, to see how this has been achieved in other settings. Community repositories allow anyone to download packages from software projects, but only project owners can upload packages to their respective projects.

See Section B.2.3 of the Deployment Considerations document for more details.

6. Director repository

As noted in Section 1.1.2, the OEM SHALL build and run a *director repository*, which contains instructions, in the form of metadata, about which images SHOULD be installed by ECUs.

Unlike the image repository, the director repository is largely controlled by automated, online processes. It: (1) produces metadata on demand, (2) instructs ECUs as to which images should be installed, and (3) MAY encrypt images per ECU (Section 6.1). Generally, the director repository produces different sets of metadata for different vehicles. Furthermore, in order to download metadata from the director repository, a primary SHALL need to submit a *vehicle version manifest*, or signed records of which images are currently installed by which ECUs.

As part of its function, it also consults a private *inventory database* about information on ECUs (Section 6.2).

See Section B.2.2 of the Deployment Considerations document for more details.

6.1 Directing installation of images on vehicles

The automated, online processes on the director repository SHALL handle requests by primaries as follows. There are five important steps.

First, the director repository decodes the *vehicle version manifest* sent by the primary within its request. The metadata format of this manifest is discussed in Section 8.1. From this manifest, it determines the unique *vehicle identifier*. Using the vehicle identifier, the director repository queries the *inventory database* for information about every ECU on the vehicle, such as whether it is a primary, whether it needs encrypted images, the *ECU key* used to encrypt its image and / or verify its signature, and so on. This database is discussed in the next subsection.

Second, it checks this manifest for accuracy. At the very least, it ensures that: (1) every ECU recorded in the inventory database is also represented in the manifest, (2) the signature of the manifest matches the ECU key of the primary which sent the manifest, and (3) the signature of a secondary's contribution to the manifest matches the ECU key of the secondary. It MAY perform additional checks, such as whether the primary has been endlessly replaying the same manifest. If these checks fail, then the repository SHOULD log the event to the inventory database, and drop the request. In this abnormal but rare event, the OEM MAY require human intervention in order for the vehicle to be able to install new updates (e.g., by requiring the vehicle owner to diagnose the problem at the nearest dealership).

Third, it determines which images should be installed next. It extracts information from the manifest, such as file names and hashes, about the images that are currently installed on ECUs. Using this information, it performs *dependency resolution*, or computation about which versions of the latest images may be safely installed together. As part of this process, information about which images *depend* or *conflict* with each other MUST be known. These dependencies and conflicts MAY be represented in the inventory database. The exact process of dependency resolution is out of the scope of this document. However, the OEM SHOULD consult well-known techniques, such as those used by the Debian package manager or our package dependency resolution project.

Fourth, the director repository MAY encrypt images for ECUs that need them. Each image is encrypted using the ECU key and key type queried earlier from the inventory database.

Fifth, the director repository generates and returns fresh metadata using information obtained from dependency resolution. It uses repository tools (Section 4) to generate fresh targets (Section 3.4), snapshot (Section 3.5), and timestamp metadata (Section 3.6). The targets metadata on the director repository is similar to the example in Section 3.4.4: there are no delegations, and every image is associated with custom metadata, such as ECU identifiers. Finally, it returns the location of this `timestamp` metadata file to the primary.

In order to write metadata and encrypted images, the director repository MAY use a filesystem, or a key-value store / database.

For example, the repository MAY use Amazon S3, Google Cloud Storage, Microsoft Azure Storage, MongoDB, MySQL, PostgreSQL, or Redis to store files.

The OEM SHOULD expose a public interface for primaries to download metadata and encrypted images from the director repository.

For example, the repository MAY make metadata and images available over FTP, FTPS, SFTP, HTTP, or HTTPS.

6.2 Inventory database

As noted in the previous subsection, the director repository uses a private *inventory database* to read and write information about ECUs and vehicles. The OEM MAY use any durable database to implement the inventory database.

For example, the OEM MAY use Amazon S3, Google Cloud Storage, Microsoft Azure Storage, MongoDB, MySQL, PostgreSQL, Redis.

In order for the director repository to read and write information from the inventory database, the OEM SHALL expose a connection to the inventory database using a standard database driver, such as ODBC or JDBC. The OEM SHALL require authentication (e.g., passwords) to control access to this interface.

The OEM MAY use the database model in Table 6.2a to read and write information about ECUs and vehicles. The OEM is free to use any other database model suitable to its own requirements.

```
BEGIN;
--
-- A table for vehicles
--
CREATE TABLE "vehicle" ("identifier" varchar(17) NOT NULL PRIMARY KEY);
--
-- A table for ECUs
--
CREATE TABLE "ecu" (
"identifier" varchar(64) NOT NULL PRIMARY KEY,
"public_key" varchar(4096) NOT NULL,
"cryptography_method" varchar(16) NOT NULL,
"primary" bool NOT NULL,
"vehicle_id" varchar(17) NOT NULL REFERENCES "vehicle" ("identifier"));
--
-- An index for the ECU table
--
CREATE INDEX "ecu_35ec04dc" ON "ecu" ("vehicle_id");
--
-- Commit the tables
--
COMMIT;
```

Table 6.2a: An example of a database model for the inventory database.

At the very least, the OEM SHALL record the following pieces of information. First, there SHOULD be a unique vehicle identifier (such as the VIN) for every vehicle. Second, there SHOULD be a unique ECU identifier, the ECU key, key type, whether it is a primary, and the vehicle identifier for every ECU. Other types of information are OPTIONAL. For example, the OEM MAY record vehicle version manifests submitted by primaries, which metadata were sent to which primaries, or dependencies and conflicts between images.

7. Time server

As noted in Section 1.1.3, an OEM SHALL implement and run a *time server* that informs its vehicles about the current time. The time server runs an infinite loop, responding to requests sent by primaries on a first-come, first-served fashion.

A primary SHALL send the time server a token from each of its secondaries. To do so, it MAY use the `Tokens` message (Table 7.2a). In response, the time server SHALL sign the tokens together with the current time.

To do so, it MAY use the `CurrentTime` message (Table 7.2a).

The time server SHALL expose a public interface so primaries may communicate with it.

The time server and primaries MAY communicate over FTP, FTPS, SFTP, HTTP, or HTTPS.

See Section B.2.1 of the Deployment Considerations document for more details.

```
TimeServerModule DEFINITIONS AUTOMATIC TAGS ::= BEGIN

  IMPORTS Length,
          Signatures,
          UTCDateTime FROM CommonModule;

  -- What a primary sends the time server: a token from each of its secondaries.
  SequenceOfTokens ::= SEQUENCE {
    numberOfTokens  Length,
    tokens          Tokens
  }
  -- Adjust length of SEQUENCE OF to your needs.
  Tokens   ::= SEQUENCE (SIZE(1..128)) OF Token
  Token    ::= INTEGER

  -- What the time server sends in response.
  CurrentTime    ::= SEQUENCE {
    signed              TokensAndTimestamp,
    numberOfSignatures  Length,
    signatures          Signatures
  }
  TokensAndTimestamp ::= SEQUENCE {
    numberOfTokens  Length,
    tokens          Tokens,
    timestamp       UTCDateTime,
    -- https://tools.ietf.org/html/rfc6025#section-2.4.2
    ...
  }

END
```

Table 7.2a: An ASN.1 modules that specifies messages that MAY be used for the time server.

8. Downloading, verifying, and installing updates on the vehicle

As noted in Section 1.2, the OEM and / or its suppliers SHALL modify software on its ECUs to download and verify the time, metadata, and images before installing a new image.

ECUs fall into one of two categories. *Primaries* inform the director repository about which images have been installed by ECUs on the vehicle. They also download and verify the latest time, metadata, and images. *Secondaries* download and verify the latest time, metadata and images from primaries. To do so, primaries and secondaries SHALL follow the steps in Section 8.1.

Before installing a new image, primaries and secondaries SHALL verify the latest downloaded time, metadata, and images. To do so, they SHALL follow the steps in Section 8.2.

Depending on whether an ECU is critical to the operational security of a vehicle, an ECU SHOULD perform either *partial* or *full verification* of metadata. Partial verification means that the director repository is the only root of trust about the image to be installed on an ECU, whereas full verification means that both the image and director repositories must agree about the image to be installed. Primary and safety-critical secondary ECUs that are updated over-the-air MUST use full verification. Non-safety-critical secondary ECUs that are updated over-the-air SHOULD use at least partial verification; if resources permit, they SHOULD perform full verification. If they are incapable of performing even partial verification, then they SHOULD NOT be updated over-the-air. To perform either full or partial verification, primaries and secondaries SHALL follow the steps in Section 8.3.

8.1 How a primary shall download and verify the time, metadata, and images, and distribute them to secondaries

A primary SHALL download, verify, and distribute the latest time, metadata, and images. To do so, it SHALL perform the following seven steps. All steps need not complete around the same time: for example, triggers MAY be used to decide when the next step SHOULD happen (e.g., the primary is idle). However, each step SHALL be performed in order.

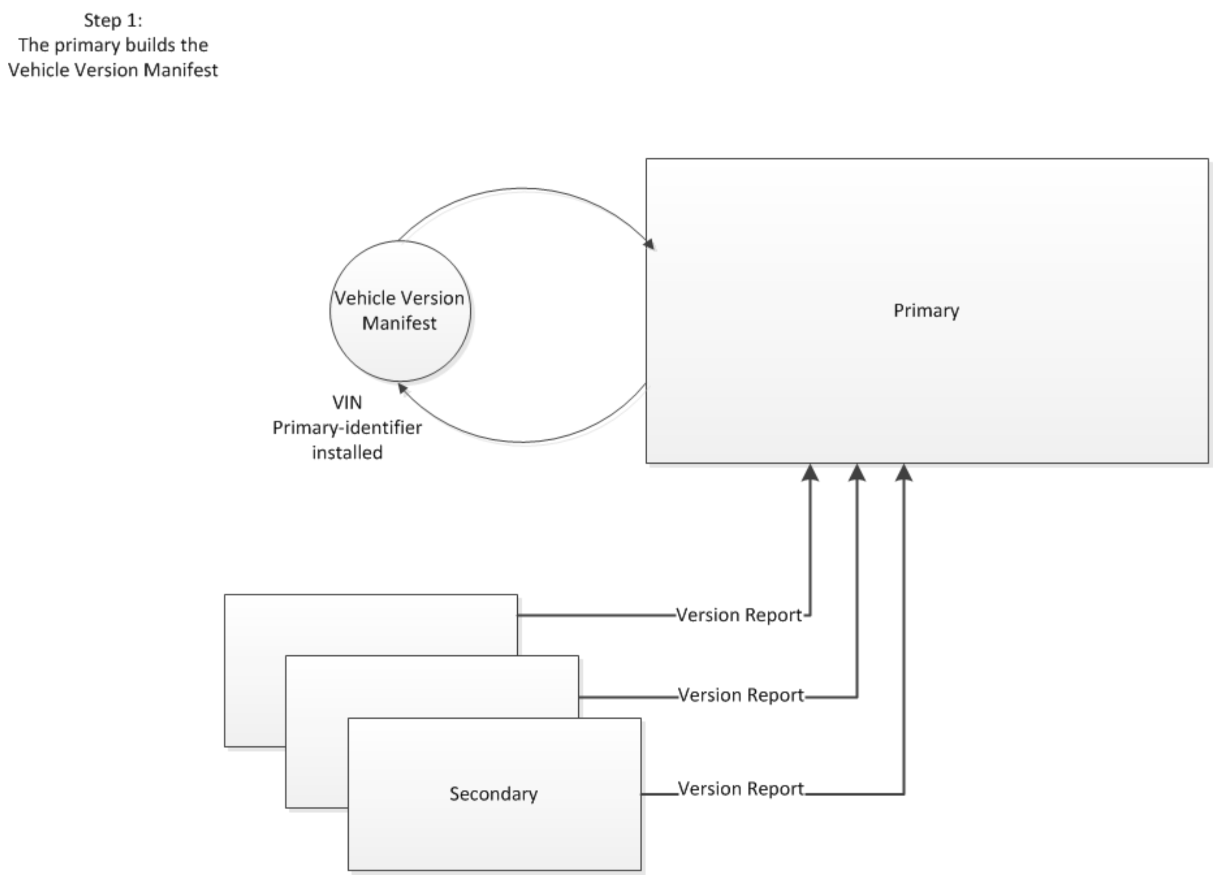

Figure 8.1a: A primary first compiles the vehicle version manifest.

First, the primary builds the *vehicle version manifest*, or a signed record of what it and each of its secondaries has currently installed (Figure 8.1a). In order for the primary to build the vehicle version manifest, prior to this step each of its secondaries SHOULD have sent the primary its version report. This report contains its ECU version manifest (Section 8.2). The primary SHOULD wait for some fixed amount of time to receive these ECU version manifests before compiling the latest vehicle version manifest, and proceeding to the next step. The primary SHALL send this vehicle version manifest to the director repository in the third step.

The primary MAY use the `ECUModule.VehicleVersionManifest` message (Table 8.1a) to encode its vehicle version manifest.

DEPRECATED AS OF 2/21/19.
Please go to the Uptane Standards Document at https://uptane.github.io/uptane-standard/uptane-standard.html

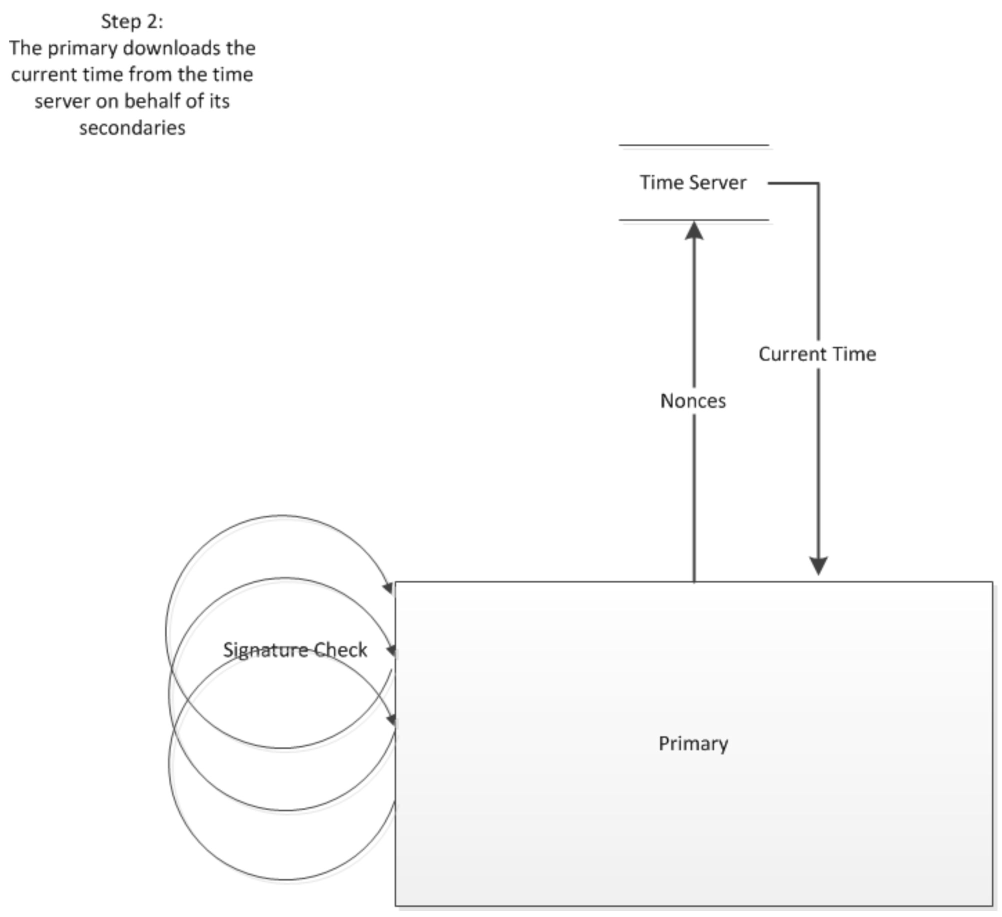
Step 2:
The primary downloads the current time from the time server on behalf of its secondaries
Time Server
Current Time
Nonces
Signature Check
Primary

Figure 8.1b: The primary downloads the current time from the time server on behalf of secondaries.

Second, the primary SHALL download the current time from the time server on behalf of its secondaries (Figure 8.1b). The primary gathers the tokens sent by each secondary in its version report (Section 8.2). The primary then sends these tokens to the time server to fetch the current time (Section 7). The primary checks the current time on behalf of every ECU: the primary verifies that the signatures are valid, and that the current time is greater than the previous downloaded time.

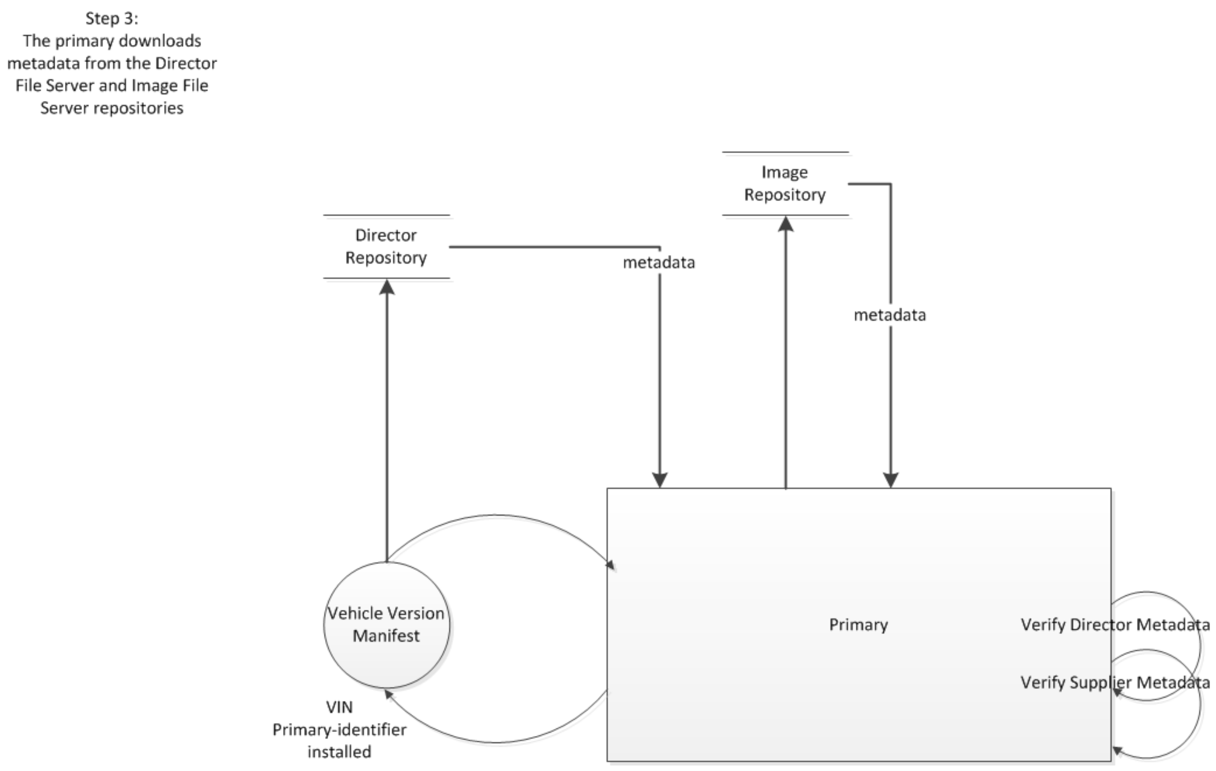

Figure 8.1c: The primary then downloads and verifies metadata from both the image and director repositories.

Third, the primary SHALL download metadata from repositories in the following order, so that full verification can be performed correctly (Figure 8.1c). Using the vehicle version manifest, the primary SHALL download metadata from the director repository. The primary SHALL then perform full verification of metadata from both the director and image repositories (Section 8.3.2). In order to do so, the primary SHALL download metadata as required from the image repository, without using the vehicle version manifest. It need not download all target metadata files from the image repository. Instead, it SHALL download only as many targets metadata files as are required to perform full verification. More details on how metadata and images are downloaded are in Section 8.3.2.

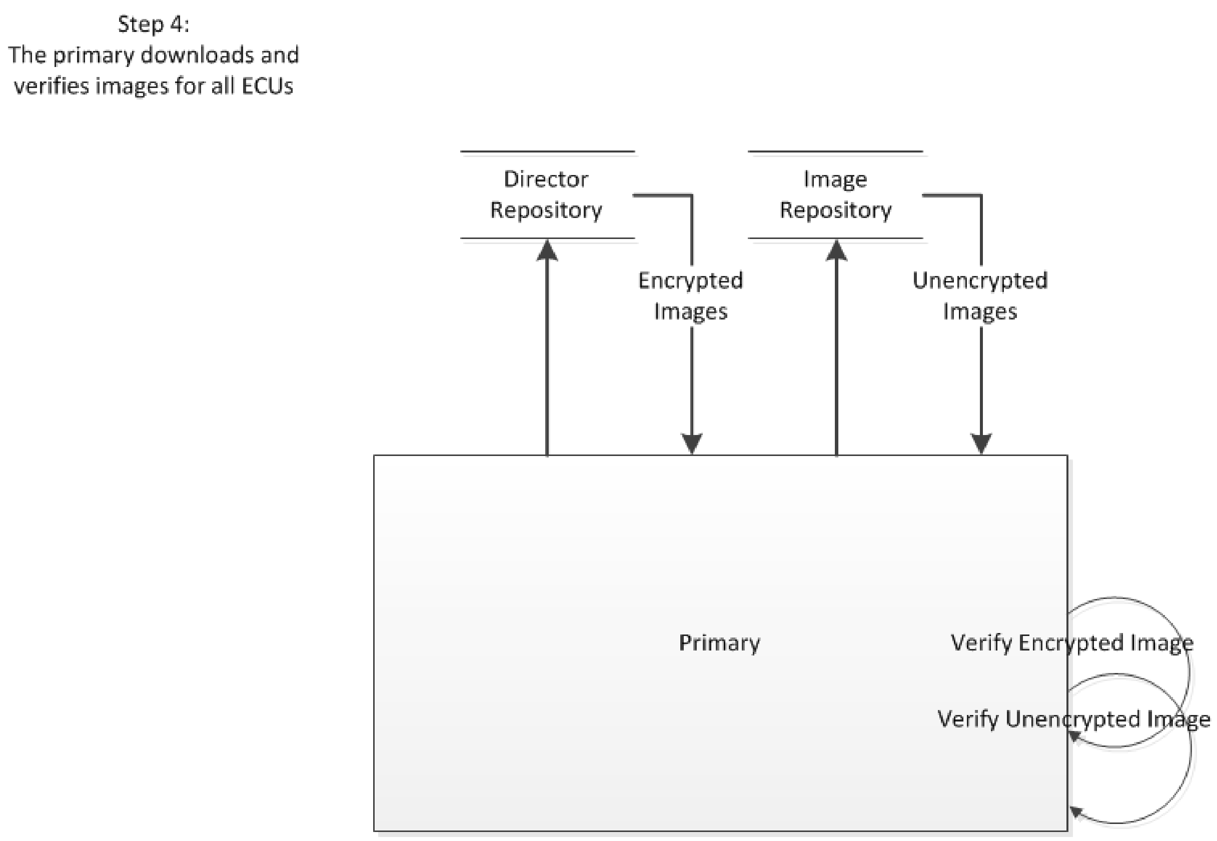

Figure 8.1d: The primary downloads and verifies images for itself and all of its secondaries.

Fourth, the primary SHALL download and verify images for itself and all of its secondaries (Figure 8.1d). The targets metadata from the director repository SHALL determine which images are encrypted. Encrypted images SHALL be downloaded from the director repository, whereas unencrypted images SHALL be downloaded from the image repository. Images SHALL be verified using only the targets metadata from the director repository. In order for a secondary without additional storage to request its image from the primary (Section 8.2), the primary SHALL associate every downloaded image with each possible filename (Section 3.7).

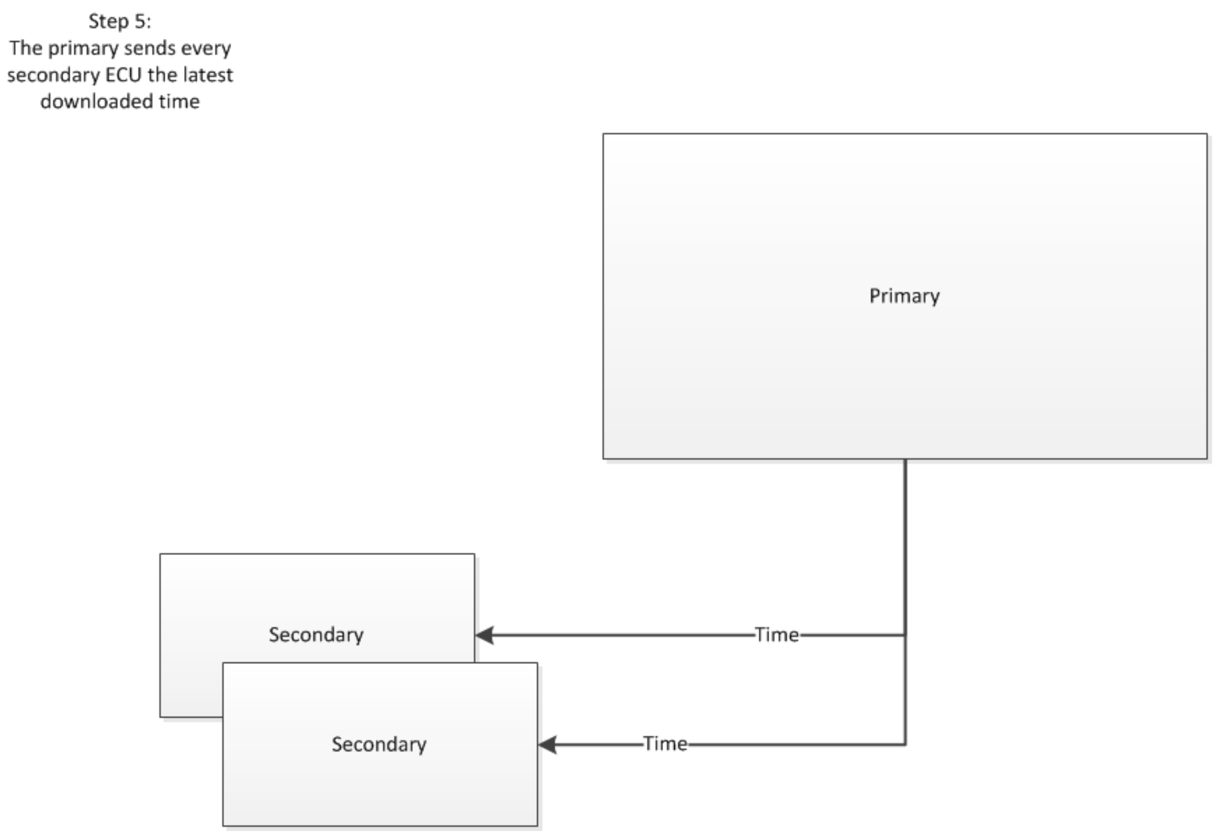
Step 5:
The primary sends every secondary ECU the latest downloaded time
Primary
Secondary
Time
Secondary
Time

Figure 8.1e: The primary sends the latest downloaded time to all ECUs.

Fifth, the primary SHALL send the latest downloaded time to all ECUs. If the primary has previously sent the latest downloaded time at that time, then the secondary SHALL overwrite that time with this one instead (Figure 8.1e).

For example, the primary MAY send each secondary a `TimeServerModule.CurrentTime` message (Table 7.2a) signed by the time server.

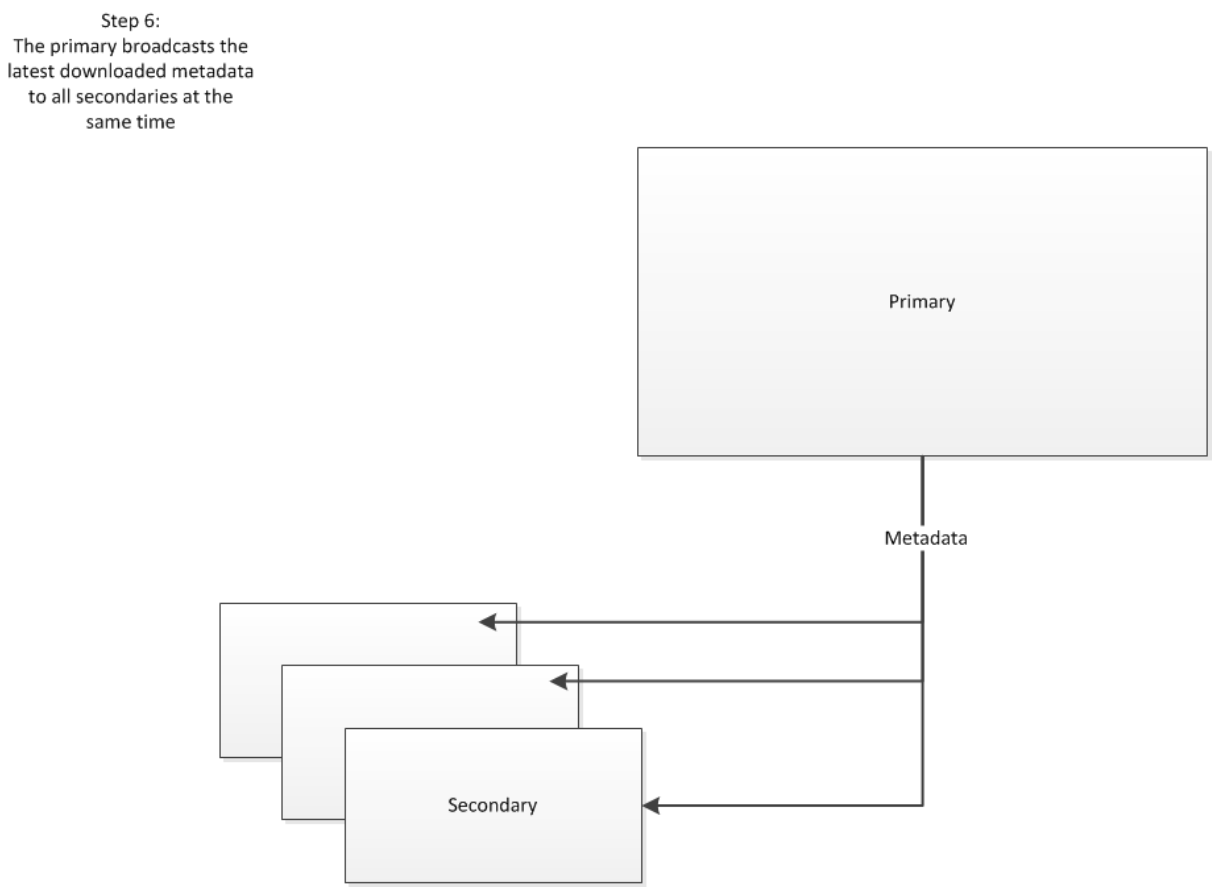
Step 6:
The primary broadcasts the latest downloaded metadata to all secondaries at the same time
Primary
Metadata
Secondary

Figure 8.1f: The primary sends metadata to all of its secondaries.

Sixth, the primary SHALL send the latest downloaded metadata to all of its secondaries (Figure 8.1f). The exact implementation details are left to the OEM.

For example, in order to accommodate a large amount of metadata, the primary MAY send the metadata over multiple messages. The `ECUModule.MetadataFiles` message (Table 8.1a) specifies: (1) a globally unique identifier (GUID) that identifies the following set of metadata, and (2) the number *n* of metadata files. For example, this GUID MAY be the current time.

Each of the following *n* `ECUModule.MetadataFile` messages (Table 8.1a) specifies: (1) the aforementioned GUID, (2) the file number, (3) the filename, and (4) the metadata itself. For example, the metadata filename "http://supplier.repository.com/timestamp.ext" MAY be associated with the timestamp metadata from the image repository, whereas the metadata filename "http://director.repository.com/timestamp.ext" MAY be associated with the timestamp metadata from the director repository.

A partial verification secondary would listen for an `ECUModule.MetadataFile` message containing the latest downloaded targets metadata file from the director repository. Note that this file may contain metadata about images for all ECUs on the vehicle. In order to help a partial verification secondary to quickly find metadata about its own image, an `ECUModule.MetadataFile` message MAY contain an offset for each such secondary that points to its metadata.

In response, a secondary SHALL prepare to receive and write the metadata. A full verification secondary SHALL keep a complete copy of all metadata, whereas a partial verification secondary SHALL keep only the targets metadata file from the director repository. A partial verification secondary SHALL NOT update any metadata file besides the targets metadata file from the director repository. If the primary has previously sent the latest downloaded metadata at that time, then the secondary SHALL overwrite that metadata with this one instead.

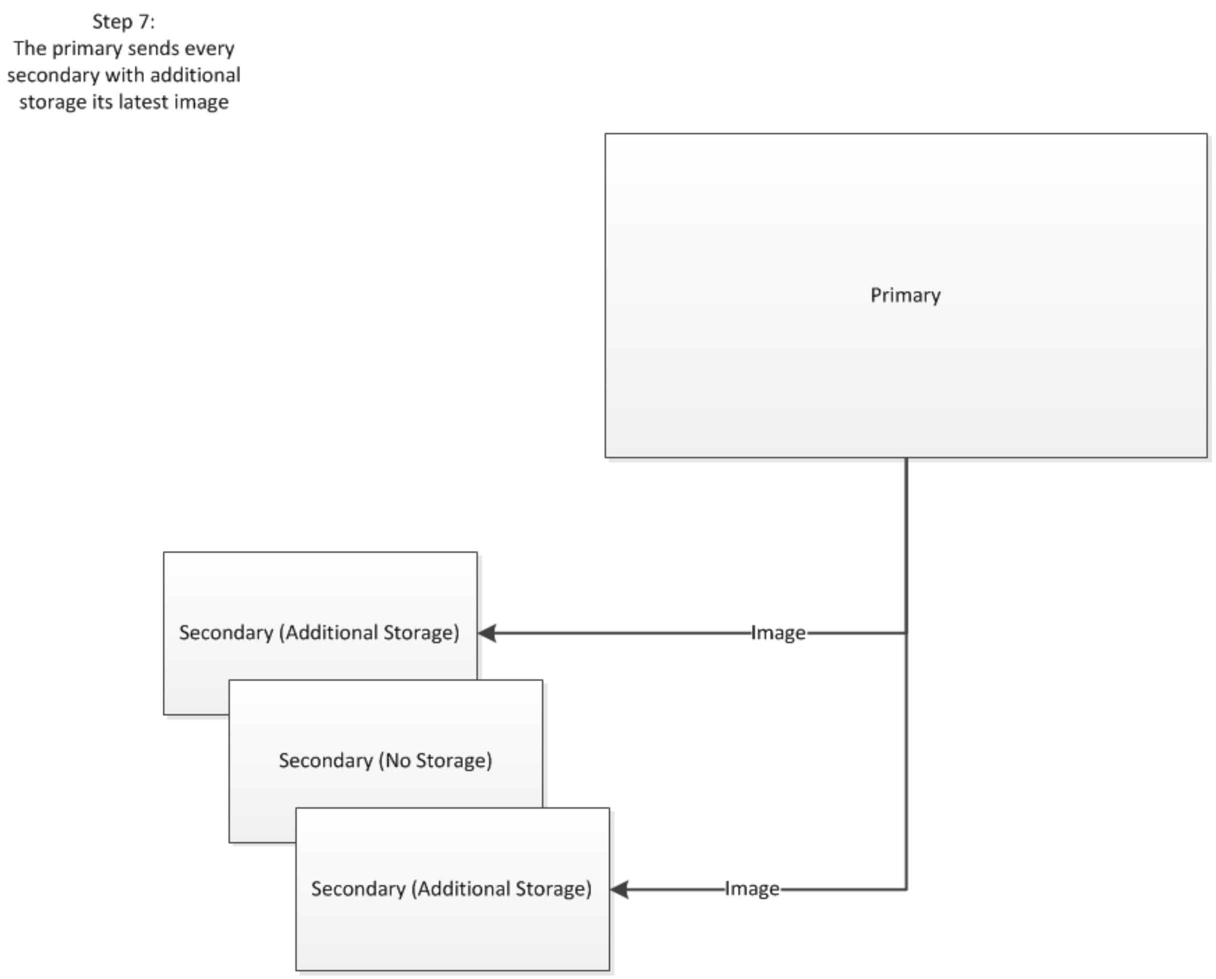
Step 7:
The primary sends every secondary with additional storage its latest image
Primary
Secondary (Additional Storage)
Image
Secondary (No Storage)
Secondary (Additional Storage)
Image

Figure 8.1g: The primary sends its image to each secondary with additional storage.

Seventh, the primary SHALL send each of its secondaries with additional storage its latest image (Figure 8.1g). The exact implementation details are left to the OEM.

For example, in order to send a large image, the primary MAY send it over multiple messages. The `ECUModule.ImageFile` message (Table 8.1a) specifies: (1) the image filename, (2) the number *n* of blocks that constitute the file, and (3) the size of each block. Each of the following *n* `ECUModule.ImageBlock` messages (Table 8.1a) specifies: (1) the aforementioned filename, (2) the block number, and (3) the content of the block itself.

In response, each such secondary SHALL prepare to receive and write the image. If the primary has previously sent the latest downloaded image at that time, then the secondary SHALL overwrite that image with this one instead.

```
ECUModule DEFINITIONS AUTOMATIC TAGS ::= BEGIN

  IMPORTS BinaryData,
          Filename,
          Identifier,
          Length,
          Positive,
          Signatures,
          UTCDateTime FROM CommonModule
          Metadata    FROM MetadataModule
          Target      FROM TargetsModule
          Token       FROM TimeServerModule;

  -- What a primary sends the director repository.
  VehicleVersionManifest ::= SEQUENCE {
    -- The signed portion of the manifest.
    signed    VehicleVersionManifestSigned,
    -- The signature on the hash of the signed portion.
    numberOfSignatures  Length,
    signatures          Signatures
  }
  VehicleVersionManifestSigned ::= SEQUENCE {
    -- A unique identifier for the vehicle.
    vehicleIdentifier           Identifier,
    -- A unique identifier for the primary sending this manifest.
    primaryIdentifier           Identifier,
    numberOfECUVersionManifests Length,
    -- An ECU version manifest per secondary.
    ecuVersionManifests         ECUVersionManifests,
    -- A message about a detected security attack, if any.
    securityAttack  VisibleString (SIZE(1..1024)) OPTIONAL,
    -- https://tools.ietf.org/html/rfc6025#section-2.4.2
    ...
  }
  -- Adjust length of SEQUENCE OF to your needs.
  ECUVersionManifests ::= SEQUENCE (SIZE(1..128)) OF ECUVersionManifest

  -- What a secondary sends its primary after installation.
  VersionReport ::= SEQUENCE {
    -- Token for the time server.
    tokenForTimeServer Token,
    -- The ECU version manifest for the director.
    ecuVersionManifest ECUVersionManifest,
    ...
  }
  ECUVersionManifest ::= SEQUENCE {
    -- The signed portion of the manifest.
    signed      ECUVersionManifestSigned,
    -- The signature on the hash of the signed portion.
    numberOfSignatures  Length,
    signatures          Signatures
  }
  ECUVersionManifestSigned ::= SEQUENCE {
    -- A unique identifier for the ECU.
    ecuIdentifier   Identifier,
    -- The previous time seen on the time server.
    previousTime    UTCDateTime,
    -- The latest known time seen on the time server.
    currentTime     UTCDateTime,
    -- A message about a detected security attack, if any.
    securityAttack  VisibleString (SIZE(1..1024)) OPTIONAL,
    -- Metadata about the installed image.
    installedImage  Target,
    ...
  }

  -- A signal from a primary to its secondaries that new metadata will be sent.
  MetadataFiles ::= SEQUENCE {
    -- A globally unique identifier (e.g., a monotonically increasing counter)
    -- for this set of metadata files.
    setGUID INTEGER,
    -- The number of metadata files that will be sent in this set.
    numberOfMetadataFiles Length
  }
  MetadataFile ::= SEQUENCE {
    -- The GUID associated with this set of metadata files.
    setGUID INTEGER,
    -- The file number in this set.
    fileNumber    Positive,
    -- The metadata filename.
    filename      Filename,
    -- The metadata itself.
    metadata      Metadata
  }

  -- What a secondary without additional storage sends its primary to request an
  -- image.
  ImageRequest ::= SEQUENCE {
    filename Filename,
    -- https://tools.ietf.org/html/rfc6025#section-2.4.2
    ...
  }
  ImageFile ::= SEQUENCE {
    -- An image filename.
    filename        Filename,
    -- The number of blocks in this image file.
    numberOfBlocks  Positive,
    -- The size of each block.
    blockSize       Positive
  }
  ImageBlock ::= SEQUENCE {
    -- The filename of the image associated with this block.
    filename    Filename,
    -- The block number of the image file.
    blockNumber Positive,
    -- The image block itself.
    block       BinaryData
  }

END
```

Table 8.1a: An ASN.1 modules that specifies messages that MAY be used to exchange the time, metadata, and images between primaries and secondaries.

8.2 What an ECU shall do to install a new image

Before installing a new image, every ECU SHALL perform the following five steps. Triggers MAY be used to decide when a new image SHOULD be installed (e.g., the vehicle is parked). If an ECU does not have additional storage, then it SHOULD be careful not to allow an update from disrupting the update process itself.

For example, the routines used to perform the following steps MAY be loaded unto and run from volatile memory, so that the update process is not interrupted by overwriting the previous working image with the latest downloaded image on non-volatile memory.

First, the ECU SHALL verify the latest downloaded time as follows. It SHALL verify that: (1) signatures are valid, (2) the list of tokens includes the token sent in the previous version report (see the fifth step), and (3) the current time is greater than the previous time. If so, then the previous downloaded time SHALL be overwritten with the latest downloaded time, and a new token SHALL be generated for the next request to the time server. Otherwise, the ECU MAY reuse the previous token for its next request to the time server, and SHALL jump to the fifth step. (Reusing this token is useful when the response from the time server has been delayed. If it is not reused, the ECU will reject the response when it finally arrives.)

Second, the ECU SHALL verify the latest downloaded metadata using either full or partial verification (Section 8.3). If the latest metadata is not successfully verified for any reason (e.g., a security attack is detected), then it SHALL jump to the fifth step.

Third, if a secondary does not have additional storage, then it SHOULD download its latest downloaded image from its primary. Otherwise, it SHALL jump to the fourth step.

To request the image from its primary, the secondary MAY send the `ECUModule.ImageRequest` message (Table 8.1a). The primary MAY respond using the `ECUModule.ImageFile` and `ECUModule.ImageBlock` messages (Table 8.1a).

The filename used to identify the latest known image SHALL be determined as follows: (1) load the targets metadata file from the director repository, (2) find the targets metadata associated with this ECU identifier, and (3) construct the image filename using the rule in Section 3.7 (i.e., HASH.FILENAME.EXT instead of FILENAME.EXT, where HASH is a hash value of either the encrypted image, if any, or the unencrypted image). Before downloading the latest known image, the ECU MAY first keep a backup of its previous working image elsewhere (e.g., the primary). Then, the ECU SHALL overwrite the previous image with the latest downloaded image. If this step fails for any reason, it SHALL jump to the fifth step.

Fourth, the ECU SHALL verify that the latest image matches the latest metadata.

The ECU SHALL: (1) load the targets metadata file from the director repository, (2) find the target metadata associated with this ECU identifier, (3) check that the hardware identifier in the latest metadata is equal to its own hardware identifier, and (4) check that the release counter of its previous image in the previous metadata, if any, is less than or equal to the release counter in the latest metadata.

If the image is encrypted, then the ECU SHALL verify the encrypted image as follows: (1) load the targets metadata file from the director repository, (2) find the target metadata associated with this ECU identifier, and (3) check whether the encrypted image matches the metadata. There are three ways to obtain the decryption key. First, if the ECU key is symmetric, then that is the decryption key. Second, if the ECU key is asymmetric, and the director has not specified an encrypted symmetric key, then the ECU key is also the decryption key. Third, if the ECU key is asymmetric, and the director has specified an encrypted symmetric key, then the decryption key is retrieved as follows: (1) load the targets metadata file from the director repository, (2) find the target metadata associated with this ECU identifier, (3) extract the encrypted symmetric key from the metadata, and (4) decrypt the symmetric key using its asymmetric ECU key. Using one of these three ways to obtain the decryption key, the ECU SHALL decrypt the image.

If the image is not encrypted, or it has been decrypted, then the ECU SHALL verify the unencrypted image as follows: (1) load the targets metadata file from the director repository, (2) find the target metadata associated with this ECU identifier, and (3) check whether the unencrypted image matches the metadata.

If the latest images match the latest metadata, then the ECU SHALL overwrite the previous with the latest metadata. If it has additional storage, it SHALL overwrite the previous image with the latest downloaded image. Otherwise, if the images do not match the metadata, it SHALL jump to the fifth step. If the ECU does not have additional storage, then it MAY restore the previous working image using the backup from the third step.

Fifth, a secondary SHALL create a *version report* to send to its primary. There are two important pieces of information in the report. The first piece contains the next token that will be sent to the time server. The second piece contains the *ECU version manifest*, or signed information about what is currently installed.

There are four types of information in an ECU version manifest. First, it lists the ECU identifier. Second, it lists the previous and current time known by the ECU. This helps the director to ascertain whether a freeze attack was accidental (e.g., because the vehicle has been offline for a long time). Third, it lists information about a security attack (e.g., whether it was a rollback or an arbitrary software attack), if any was detected. Finally, it lists metadata about the unencrypted image currently installed on the ECU.

The version report MAY be written using the `ECUModule.VersionReport` message (Table 8.1a).

After installing the latest image, a secondary SHALL submit its version report to its primary. The primary SHALL write the version report to disk, and associate it with the secondary. The primary SHALL include its own ECU version manifest in the vehicle version manifest.

8.3 How an ECU shall verify metadata

A primary SHALL always perform *full verification* of metadata (Section 8.3.2). A secondary SHOULD perform either full or *partial verification* (Section 8.3.1). In order to verify metadata, we assume that the ECU has been prepared with some necessary information (see Section A.2 of the Deployment Considerations document).

8.3.1 Partial verification of metadata

In order to perform partial verification, a secondary SHALL:

1. Load the latest downloaded time from the time server.
2. Load the latest top-level targets metadata file from the director repository.
    a. Check for an arbitrary software attack. This metadata file must have been signed by a threshold of keys specified in the previous root metadata file.
    b. Check for a rollback attack. The version number in the previous targets metadata file, if any, must be less than or equal to the version number in this targets metadata file.
    c. Check for a freeze attack. The latest downloaded time should be lower than the expiration timestamp in this metadata file.
    d. Check that there are no delegations.
    e. Check that any ECU's identifier has been represented at most once.
3. Return an error code indicating whether or not a security attack was detected.

8.3.2 Full verification of metadata

Full verification of metadata means that a primary, or full verification secondary, checks that the targets metadata about unencrypted images from the director repository matches the targets metadata about the same images from the image repository. It provides substantial resilience to a key compromise in the system.

Note that, in the following discussion, instructions to download a file SHALL apply only to a primary. A full verification secondary SHALL always receive files from its primary (Section 8.1). A primary SHALL download metadata and target files following the rules specified in TAP 5. A primary SHALL also download these files following the file renaming rules specified in Section 3.7.

To begin full verification, an ECU SHALL load the map file. Using the information specified in the map file, the ECU SHALL first download and verify metadata from the director repository as follows:

1. Load the latest downloaded time from the time server.
2. Download and check the root metadata file.
    a. Check signatures. It must have been signed by a threshold of keys specified in the previous root metadata file.
    b. Check for a rollback attack. The version number of the previous root metadata file must be less than or equal to the version number of this root metadata file.
    c. Check for a freeze attack. The latest downloaded time should be lower than the expiration timestamp in this metadata file.
    d. If the the timestamp and / or snapshot keys have been rotated, then delete the previous timestamp and snapshot metadata files. This is done in order to recover from *fast-forward attacks* after the repository has been compromised and recovered. A fast-forward attack happens when attackers arbitrarily increase the version numbers of: (1) the timestamp metadata, (2) the snapshot metadata, and / or (3) the targets, or a delegated targets, metadata file in the snapshot metadata. Please see the Mercury paper for more details (to be published soon).
3. Download and check the timestamp metadata file.
    a. Check signatures. It must have been signed by a threshold of keys specified in the root metadata file.
    b. Check for a rollback attack. The version number of the previous timestamp metadata file, if any, must be less than or equal to the version number of this timestamp metadata file.
    c. Check for a freeze attack. The latest downloaded time should be lower than the expiration timestamp in this metadata file.
4. Download and check the snapshot metadata file.
    a. Check signatures. It must have been signed by a threshold of keys specified in the latest root metadata file.
    b. Check for a rollback attack.
        i. The version number of the previous snapshot metadata file, if any, must be less than or equal to the version number of this snapshot metadata file.
        ii. The version number of the targets metadata file, and all delegated targets metadata files (if any), in the previous snapshot metadata file, if any, must be less than or equal to its version number in this snapshot metadata file. Furthermore, any targets metadata filename that was listed in the previous snapshot metadata file, if any, must continue to be listed in this snapshot metadata file.
    c. Check for a freeze attack. The latest downloaded time should be lower than the expiration timestamp in this metadata file.
5. Download and check the top-level targets metadata file.

   a. Check for an arbitrary software attack. This metadata file must have been signed by a threshold of keys specified in the latest root metadata file.
   b. Check for a rollback attack. The version number of the previous targets metadata file, if any, must be less than or equal to the version number of this targets metadata file.
   c. Check for a freeze attack. The latest downloaded time should be lower than the expiration timestamp in this metadata file.
   d. Check for a mix-and-match attack. The version number in this targets metadata file must match the snapshot metadata.
   e. Check that there are no delegations.
   f. Check that every ECU identifier has been represented at most once.
6. If applicable, return an error code indicating a security attack.

To finish full verification, the ECU SHALL then download and verify metadata from the image repository as follows:

1. Load the latest downloaded time from the time server.
2. Repeat steps 2-4 as for the director repository to download and check the root, timestamp, and snapshot metadata files.
3. For each image listed in the targets metadata file from the director repository, perform the following preorder depth-first search for an image with exactly the same file name, beginning with the top-level targets metadata file from the image repository:
   a. Repeats steps 5a-5d as for the director repository to check for arbitrary software, rollback, freeze, and mix-and-match attacks.
   b. If the current targets role has signed metadata about the image, then return the metadata.
   c. Otherwise, recursively search the list of delegations in order of appearance.
      i. If it is a multi-role delegation, recursively visit each role, and check that each has signed exactly the same non-custom metadata (i.e., length and hashes) about the image (or the lack of it).
      ii. If it is a terminating delegation, then jump to step 3d.
      iii. Otherwise, if it is a non-terminating delegation, continue processing the next delegation, if any. Stop the search, and jump to step 3d as soon as a delegation returns a result.
   d. If there is metadata about the image, then:
      i. Check that the non-custom metadata (i.e., length and hashes) matches the non-custom metadata about the unencrypted image from the director repository.
      ii. Check that certain custom metadata (hardware identifier, and release counter) matches the custom metadata about the image from the director repository.
      iii. Check that the release counter in the previous targets metadata file is less than or equal to the release counter in this targets metadata file.
      iv. Return the metadata.
   e. Otherwise, if there is no such metadata, then return an error code indicating that the image is missing.

4. If applicable, return an error code indicating a security attack (e.g., rollback, endless data, or arbitrary software attack).

A primary SHOULD perform additional checks, such as:

1. Ensuring that the ECU identifiers present in the targets metadata from the director repository are a subset of the actual ECU identifiers of ECUs in the vehicle.

8.4 Notes about some implementation details

We have left the following implementation details to the OEM.

Although we specify high-level ASN.1 messages that primaries and secondaries can use to exchange the time, metadata, and images, we do not specify the low-level network protocol used to transfer these messages. This is because the network protocol is OEM-specific: one OEM may use CAN, while another may use Ethernet. Nevertheless, our reference implementation points to how ASN.1 messages may be transferred over CAN.

We do not explicitly specify how to handle exceptions in sending and receiving messages between primaries and secondaries. An OEM is free to: (1) require acknowledgements of sent messages, (2) retry sending messages that have not been acknowledged in a sufficient amount of time, (3) require a finite waiting time for a response, and so on.

We assume that ECUs have a notion of files and directories. While this assumption may be true for primaries and full verification secondaries, it may not hold true for partial verification secondaries. Note that partial verification secondaries are not required to actually have a filesystem. Instead, they are free to implement the notion of files and directories using alternate means (e.g., reading from and writing directly to memory) as long as: (1) “files” and “directories” are somehow addressable using file and directory names respectively, (2) they are kept separate from each other, and (3) they persist on non-volatile memory.

Acknowledgements

We would like to thank Lois Anne DeLong for her efforts on this document. Our work on Uptane is supported by U.S. Department of Homeland Security grants D15PC00239 and D15PC00302.