---

title: Uptane IEEE-ISTO Standard for Design and Implementation abbrev: UPTANE docname: uptane-standard-design date: 2018-08-28 category: info

ipr: noDerivativesTrust200902 area: TODO workgroup: TODO keyword: Internet-Draft

stand_alone: yes pi: [toc, sortrefs, symrefs]

author: - ins: J. Cappos name: Justin Cappos organization: NYU Tandon School of Engineering email: redacted@nyu.edu street: todo country: USA region: NY city: New York code: todo

normative: # Keyword def (MUST, SHALL, MAY, etc.) RFC2119: # HTTP/1.1 RFC2616: # X.509 RFC3280: # PKCS:RSA RFC3447: # X.509 PKI spec RFC3647: # SHA RFC4634: # base64 RFC4648: # RSA updates RFC5756: # JSON RFC7159: # TAP 3 at rev d0818e5 TAP-3: target: https://github.com/theupdateframework/taps/commit/d0818e580c322815a473520f2e8cc5f5eb8df499 title: The Update Framework TAP 3 - Multi-role delegations author: - ins: T.K. Kuppusamy - ins: S. Awwad - ins: E. Cordell - ins: V. Diaz - ins: J. Moshenko - ins: J. Cappos date: 2018-01-18 # TAP 4 at rev 2cb67d9 TAP-4: target: https://github.com/theupdateframework/taps/commit/2cb67d913ec19424d1e354b38f862886fbfd4105 title: The Update Framework TAP 4 - Multiple repository consensus on entrusted targets author: - ins: T.K. Kuppusamy - ins: S. Awwad - ins: E. Cordell - ins: V. Diaz - ins: J. Moshenko - ins: J. Cappos date: 2017-12-15 # TAP 5 at rev 01726d2 TAP-5: target: https://github.com/theupdateframework/taps/blob/01726d203c9b9c029d26f6612069ce3180500d9a/tap5.md#do metadata-and-target-files title: The Update Framework TAP 5 - Setting URLs for roles in the root metadata file author: - ins: T.K. Kuppusamy - ins: S. Awwad - ins: E. Cordell - ins: V. Diaz - ins: J. Moshenko - ins: J. Cappos date: 2018-01-22 # TUF at rev 2b4e184 TUF-spec: target: https://github.com/theupdateframework/specification/blob/2b4e18472fe25d5b57f36f6fa50104967c8faeaa/tuf-spec.md title: The Update Framework Specification author: - ins: J. Samuel - ins: N. Mathewson - ins: G. Condra - ins: V. Diaz - ins: T.K. Kuppusamy - ins: S. Awwad - ins: S. Tobias - ins: J. Wright - ins: H. Mehnert - ins: E. Tryzelaar - ins: J. Cappos - ins: R. Dingledine date: 2018-09-19

informative: # MD5 RFC1321: ED25519: title: ‘“High-Speed High-Security Signatures”, Journal of Cryptographic Engineering, Vol. 2’ author: - ins: D. J. Bernstein - ins: N. Duif - ins: T. Lange - ins: P. Schwabe - ins: B-Y. Yang date: 2011-09-26 MERCURY: target: https://www.usenix.org/system/files/conference/atc17/atc17-kuppusamy.pdf title: “Mercury: Bandwidth-Effective Prevention of Rollback Attacks Against Community Repositories” author: - ins: T.K. Kuppusamy - ins: V. Diaz - ins: J. Cappos seriesinfo: ISBN: 978-1-931971-38-6 date: 2017-07-12 PEP-458: target: https://www.python.org/dev/peps/pep-0458/ title: “PEP 458 – Surviving a Compromise of PyPI” author: - ins: T.K. Kuppusamy - ins: V. Diaz - ins: D. Stufft - ins: J. Cappos date: 2013-09-27 # TODO add TUF-CCS-2010 #

TODO add DIPLOMAT-NSDI-2016 # TODO add DER USATODAY: target: https://www.usatoday.com/story/tech/columnist/2016/06/28/your-average-car-lot-more-code-driven-than-you-think/86437052/ title: Your average car is a lot more code-driven than you think author: - ins: B. O'Donnell date: 2016-06-28 CR-OTA: target: https://www.consumerreports.org/automotive-technology/automakers-embrace-over-the-air-updates-can-we-trust-digital-car-repair/ title: Automakers Embrace Over-the-Air Updates, but Can We Trust Digital Car Repair? author: - ins: K. Barry date: 2018-04-20 IN-TOTO: target: https://in-toto.github.io/ title: "in-toto: A framework to secure the integrity of software supply chains" date: 2018-10-29

— abstract

This document describes a framework for securing automotive software update systems.

— middle

# Introduction

Uptane is a secure software update framework for automobiles. This document describes procedures to enable programmers for OEMs and suppliers to design and implement this framework to better protect connected units on cars. Integrating Uptane as outlined in the sections that follow can reduce the ability of attackers to compromise critical systems. It also assures a faster and easier recovery process should a compromise occur.

These instructions specify the components necessary for a compliant implementation. Individual implementors can make their own technological choices within those requirements. This flexibility makes Uptane adaptable to the many customized update solutions used by manufacturers.

# Terminology

## Conformance Terminology

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 {{RFC2119}}.

In order to be considered "Uptane-compliant," an implementation MUST follow all of these rules as specified in the document.

## Automotive Terminology

*Bus*: An internal communications network that interconnects components within a vehicle. A car can have a number of buses that will vary in terms of power, speed and resources.
*Image*: File containing software for an ECU to install. May contain a binary image to flash, installation instructions, and other necessary information for the ECU to properly apply the update. Each ECU typically holds only one image, although this may vary in some cases.
*Primary/Secondary ECUs*: Terms used to describe the control units within an automobile. A primary ECU downloads from a repository and verifies update images and metadata for itself and for secondary ECUs, and distributes images and metadata to secondaries. Thus, it requires extra storage space and a connection to the internet. Secondary ECUs receive their update images and metadata from the primary, and only need to verify and install their own metadata and images.
*Repository*: A server containing metadata about images. May also contain the images themselves.
*Suppliers*: Independent companies to which auto manufacturers may outsource the production of ECUs. Tier-1 suppliers directly serve the manufacturers. Tier-2 suppliers are those that perform outsourced work for Tier-1 suppliers.
*Vehicle Version Manifest*: A compilation of all ECU version manifests on a vehicle. It serves as a master list of all images currently running on all ECUs in the vehicle.

## Uptane Role Terminology

These terms are defined in greater detail in {{roles}}.

*Delegations*: Designating the responsibility of signing metadata about images to another party.
*Roles*: The roles mechanism of Uptane allows the system to distribute signing responsibilities so that the compromise of one key does not necessarily impact the security of the entire system.

- *Root Role*: Distributes and revokes public keys used to verify the root, timestamp, snapshot, and targets role metadata.
- *Snapshot Role*: Indicates which images the repository has released at the same time.
- *Targets Role*: Holds the metadata used to verify the image, such as cryptographic hashes and file size.
- *Timestamp Role*: Indicates if there are any new metadata or image on the repository.

### Acronyms and Abbreviations

*CAN Bus*: Controller Area Network bus standard.
*ECUs*: Electronic Control Units, the computing units on vehicle.
*LIN Bus*: Local Interconnect Bus.
*SOTA*: Software Updates Over-the-Air.
*VIN*: Vehicle Identification Number.

## Rationale for and Scope of Uptane Standards

This Standards document clarifies the essential components and best practices for the secure design implementation and deployment of Uptane by OEMs and suppliers. These practices contribute to compromise resilience, or the ability to minimize the extent of the threat posed by any given attack.

### Why Uptane requires standards

A standards document that can guide the safe design, integration and deployment of Uptane in cars is needed at this time because:

- The number of connected units on the average vehicle continues to grow, with mainstream cars now containing up to 100 million lines of code. {{USATODAY}}
- The expanded use of software over-the-air strategies creates new attack surfaces for malicious parties. {{CR-OTA}}
- Legacy update strategies, such as SSL/TLS or GPG/RSA, are not feasible for use on automotive ECUs because they force manufacturers to chose between enhanced security and customizability.
- Conventional strategies are also complicated by the differing resources of the ECUs, which can vary greatly in memory, storage space, and Internet connectivity.
- The design of Uptane makes it possible to offer improved design flexibility, without sacrificing security.
- This added design flexibility, however, could be a liability if the framework is implemented incorrectly.
- Standardization of crucial steps in the design, implementation and use of Uptane can assure that customizability does not impact security or functionality.

### Scope of Standards Coverage

This document sets guidelines for implementing Uptane in most systems capable of updating software on connected units in cars. In this section, we define the

scope of that applicability by providing sample use cases and possible exceptions, aspects of update security that are not applicable to Uptane, and the design requirements governing the preparation of these standards.

### Use Cases

The following use cases provide a number of scenarios illustrating the manner in which software updates could be accomplished using Uptane.

#### OEMs initializing Uptane at the factory using SOTA

Bob, who works for an OEM, is overseeing the installation of Uptane on new vehicles at a manufacturing plant. He starts with preparing the ECUs by adding the following components: code to perform full and partial verification, the latest copy of the relevant metadata, the public keys, and the latest time, signed by the time server. His implementation would be considered Uptane-compliant if:

1. all primaries perform full verification;
2. all secondaries that are updated via OTA perform full or partial verification; and
3. all other ECUs that do not perform verification cannot be updated via OTA.

#### Updating one ECU with a complete image

Alice, a Tier-1 supplier, completes work on a revised image for an electronic brake control module. This module will control the brakes on all models of an SUV produced by the OEM for whom Clark is in charge of electronic systems. Alice signs the image, then delivers it and all of its metadata, including delegations, and associated images to Clark. Clark adds these metadata and images to the image repository, along with information about any dependencies and conflicts between this image and those on other ECUs. Clark also updates the inventory database, so that the director repository can instruct the ECU on how to install these updated images.

#### Dealership updating individual ECUs on demand

Dana runs a dealership for a major OEM. The OEM has issued a recall to address a problem with a keyless entry device that has been locking people out of their cars. Individual owners are bringing in a revised image on a flash drive that was sent to them from the manufacturer via courier mail. To carry out this update, the OEM would first have to delegate to Dana the authority to sign the metadata that would need to accompany the image on the flashdrive. He would then follow the same procedures used by Clark in the example above.

**Update one ECU with multiple deltas**

Frances needs to update an On-Board Diagnostics port and has several new images to download. To save bandwidth costs, she uses delta images that contain only the code and/or data that has changed from the previous image installed by the ECU. To do so, she must first modify the director repository using the vehicle version manifest and dependency resolution to determine the differences between the previous and latest images. Frances then adds the following to the custom targets metadata used by the director repository: (1) the algorithm used to apply a delta image and (2) the targets metadata about the delta image. Frances would also check whether the delta images match the targets metadata from the director repository.

## Exceptions

There are a number of factors that could impede the completion of the above scenarios: * ECUs may be lacking the necessary resources to function as designated. These resources could include weaknesses, in terms of CPU or RAM, that prevent performance of public key cryptography; or they may lack sufficient storage to undo installation of bad software; or they simply may reside on a low-speed network (e.g., LIN) * ECUs may reside on different network segments, and may not be able to directly reach each other, requiring a gateway to facilitate communication. * A user may replace OEM-installed ECUs with aftermarket ECUs instead. * A vehicle may be able to download only a limited amount of data via a cellular channel (due to limits on a data plan). * A system may lack sufficient power to download or install software updates. * Vehicles may be offline for extended periods of time, thus missing required updates (e.g., key rotations). * OEMs may be unwilling to implement costly security or hardware requirements.

## Out of Scope

The following topics will not be addressed in this document, as they represent threats outside the scope of Uptane:

- Physical attacks, such as manual tampering with ECUs outside the vehicle.
- Compromise of the supply chain (e.g., build system, version control system, packaging process). A number of strategies already (e.g., git signing, TPMs, in-toto {{IN-TOTO}}) exist to address this problem. Therefore, there is no need duplicate those techniques here.
- Problems associated with OBD or UDS programming of ECUs, such as authentication of communications between ECUs.

### Design Requirements

The design requirements for this document are governed by three principal parameters:

- to clearly mandate the design and implementation steps that are security critical and must be followed as is, while offering flexibility in the implementation of non-critical steps. In this manner, users can adapt to support different use models and deployment scenarios.
- to delineate best practices to ensure that, should a vehicle be attacked, an attacker is forced to compromise many different systems.
- to ensure that, if implemented, the security practices mandated or suggested in this document do not interfere with the functionality of ECUs, vehicles, or the manufacturing systems on which they run.

## Threat Model and Attack Strategies

The overarching goal of Uptane is to provide a system that is resilient in the face of various types of compromise. In this section, we describe the goals that an attacker may have ({{attacker_goals}}) and the capabilities they may have or develop ({{capabilities}}). We then describe and classify types of attack on the system according to the attacker's goals ({{threats}}).

### Attacker goals

We assume that attackers may want to achieve one or more of the following goals, in increasing order of severity:

- Read the contents of updates to discover confidential information or reverse-engineer firmware
- Deny installation of updates to prevent vehicles from fixing software problems
- Cause one or more ECUs in the vehicle to fail, denying use of the vehicle or of certain functions
- Control the vehicle or ECUs within the vehicle

### Attacker capabilities

Uptane is designed with resilience to compromise in mind. We assume that attackers may develop one or more of the following capabilities:

- Read and analyze the contents of previous and/or current versions of software, as well as the update sequence and instructions

- Intercept and modify network traffic (i.e., perform man-in-the-middle attacks). This capability may be developed in two domains:
  - Outside the vehicle, intercepting and modifying traffic between the vehicle and software repositories
  - Inside the vehicle, intercepting and modifying traffic on one or more vehicle buses (e.g. via an OBD port or using a compromised ECU as a vector)
- Compromise and control one or more ECUs within a vehicle
- Compromise signing or encryption keys
- Compromise and control software repository servers (and any keys stored on the repository)

## Description of threats

Uptane's threat model considers the following types of attack, organized according to the attacker goals listed in {{attacker_goals}}.

### Read updates

- *Eavesdrop attack:* Read the unencrypted contents of an update sent from a repository to a vehicle.

### Deny installation of updates

An attacker seeking to deny installation of updates may attempt one or more of the following strategies:

- *Drop-request attack:* Block network traffic outside or inside the vehicle.
- *Slow retrieval attack:* Slow down network traffic, in the extreme case sending barely enough packets to avoid a timeout. Similar to a drop-request attack, except that both the sender and receiver of the traffic still think network traffic is unimpeded.
- *Freeze attack:* Continue to send a previously known update to an ECU, even if a newer update exists.
- *Partial bundle installation attack:* Install updates to some ECUs, but freeze updates on others.

### Interfere with ECU functionality

Attackers seeking to interfere with the functionality of vehicle ECUs in order to cause an operational failure or unexpected behaviour may do so in one of the following ways:

- *Rollback attack:* Cause an ECU to install a previously-valid software revision that is older than the currently-installed version.
- *Endless data attack:* Send a large amount of data to an ECU, until it runs out of storage, possibly causing the ECU to fail to operate.
- *Mix-and-match attack:* Install a set of images on ECUs in the vehicle that are incompatible with each other. This may be accomplished even if all of the individual images being installed are valid, as long as there exist valid versions that are mutually incompatible.

#### Control an ECU or vehicle

Full control of a vehicle, or one or more ECUs within a vehicle, is the most severe threat.

- *Arbitrary software attack:* Cause an ECU to install and run arbitrary code of the attacker's choice.

## Detailed Design of Uptane

Uptane is a secure software update framework for automobiles. We do not specify implementation details. Instead, we describe the components necessary for a compliant implementation, and leave it up to individual implementors to make their own technological choices within those requirements.

At a high level, Uptane requires:

- Two software repositories:
  - An image repository containing binary images for install, and signed metadata about those images
  - A director repository connected to an inventory database that can sign metadata on demand for images in the image repository
- Repository tools for generating Uptane-specific metadata about images
- A public key infrastructure supporting the required metadata production/signing roles on each repository:
  - Root - Certificate authority for the repo. Distributes public keys for verifying all the other roles' metadata
  - Timestamp - Indicates whether there are new metadata or images
  - Snapshot - Indicates images released by the repository at a point in time, via signing metadata about targets metadata
  - Targets - Indicates metadata about images, such as hashes and file sizes
- A time server to deliver cryptographically verifiable time to ECUs
- An in-vehicle client on a primary ECU capable of verifying the signatures on all update metadata, handling all server communication, and downloading updates on behalf of secondary ECUs

- A client or library on each secondary ECU capable of performing either full or partial verification of metadata

## Roles on repositories

A repository contains images and metadata. Each role has a particular type of metadata associated with it, as described in {{meta_syntax}}.

### The Root role

The Root role SHALL be responsible for a Certificate Authority as defined in {{RFC3647}}. The Root role SHALL produce and sign Root metadata as described in {{root_meta}}. The Root role SHALL sign the public keys used to verfy the metadata produced by the Timestamp, Snapshot, and Targets roles. The Root role SHALL revoke keys for the other roles, in case of compromise.

### The Targets role

The Targets role SHALL produce and sign metadata about images and delegations as described in {{targets_meta}}.

### Delegations

The Targets role on the Image repository MAY delegate the responsibility of signing metadata to other, custom-defined roles. If it does, it MUST do so as specified in {{delegations_meta}}.

Responsibility for signing images or a subset of images MAY be delegated to more than one role, and therefore it is possible for two different roles to be trusted for signing a particular image. For this reason, delegations MUST be prioritized.

A particular delegation for a subset of images MAY be designated as **terminating**. For terminating delegations, the client SHALL NOT search the any further if it does not find validly signed metadata about those images in the terminating delegation. Delegations SHOULD NOT be terminating by default; terminating delegations SHOULD only be used when there is a compelling technical reason to do so.

A delegation for a subset of images MAY be a multi-role delegation {{TAP-3}}. A multi-role delegation indicates that each of the delegatee roles MUST sign the same metadata.

Delegations only apply to the Image repository. The Targets role on the Director repository MUST NOT delegate metadata signing responsibility.

#### The Snapshot role

The Snapshot role SHALL produce and sign metadata about all Targets metadata the repository releases, including the current version number and hash of the main Targets metadata and the version numbers and hashes of all delegated targets metadata, as described in {{snapshot_meta}}.

#### The Timestamp role

The Timestamp role SHALL produce and sign metadata indicating whether there are new metadata or images on the repository. It MUST do so by signing the metadata about the Snapshot metadata file.

### Metadata abstract syntax

#### Common Metadata Structures and Formats

#### Root Metadata

#### Targets Metadata

#### Metadata about Images

#### Metadata about Delegations

#### Snapshot Metadata

#### Timestamp Metadata

#### The map file

#### Rules for filenames in repositories and metadata

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 the TUF specification {{TUF-spec}} and PEP 458 {{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 {{common_metadata}}.

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 {{director_repository}}, 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 {{metadata_verification}}, 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 {{targets_meta}}. 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.

#### Vehicle version manifest

#### ECU version report

## Server / repository implementation requirements

An Uptane implementation SHALL make the following services available to vehicles:

- Image repository
- Director repository
- Time server

### Image Repository

The Image repository exists to allow an OEM and/or its suppliers to upload images and their associated metadata. It makes these images and their metadata available to vehicles. The Image repository is designed to be primarily controlled by human actors, and updated relatively infrequently.

The Image repository SHALL expose an interface permitting the download of metadata and images. This interface SHOULD be public.

The Image repository SHALL require authorization for writing metadata and images.

The Image repository SHALL provide a method for authorized users to upload images and their associated metadata. It SHALL check that a user writing metadata and images is authorized to do so for that specific image by checking the chain of delegations for the image as described in {{delegations_meta}}.

The Image repository SHALL implement storage which permits authorized users to write an image file using a unique filename, and later read the same file using the same name. It MAY use any filesystem, key-value store, or database that fulfills this requirement.

The Image repository MAY require authentication for read access.

### Director Repository

The Director repository instructs ECUs as to which images should be installed by producing signed metadata on demand. Unlike the Image repository, it is mostly controlled by automated, online processes. It also consults a private inventory database containing information on vehicles, ECUs, and software revisions.

The Directory repository SHALL expose an interface for primaries to upload vehicle version manifests ({{vehicle_version_manifest}}) and download metadata. This interface SHOULD be public. The Director MAY encrypt images for ECUs that require it, either by encrypting on-the-fly or by storing encrypted images in the repository.

The Director repository SHALL implement storage which permits an automated service to write generated metadata files. It MAY use any filesystem, key-value store, or database that fulfills this requirement.

### Directing installation of images on vehicles

A Director repository MUST conform to the following six-step process for directing the installation of software images on a vehicle.

1. When the Director receives a vehicle version manifest sent by a primary (as described in {{construct_manifest_primary}}), it decodes the manifest, and determines the unique vehicle identifier.
2. Using the vehicle identifier, the Director queries its inventory database (as described in {{inventory_db}}) for relevant information about each ECU in the vehicle.
3. The Director checks the manifest for accuracy compared to the information in the inventory database. If any of the required checks fail, the Director drops the request. An implementor MAY make whatever additional checks they wish. At a minimum, the following checks are required:
   - Each ECU recorded in the inventory database is also represented in the manifest.
   - The signature of the manifest matches the ECU key of the primary that sent it.
   - The signature of each secondary's contribution to the manifest matches the ECU key of that secondary.
4. The Director extracts information about currently installed images from the vehicle version manifest. Using this information, it determines if the vehicle is already up-to-date, and if not, determines a set of images that should be installed. The exact process by which this determination takes place is out of scope of this standard. However, it MUST take into account *dependencies* and *conflicts* between images, and SHOULD consult well-established techniques for dependency resolution.
5. The Director MAY encrypt images for ECUs that require it.
6. The Director generates new metadata representing the desired set of images to be installed in the vehicle, based on the dependency resolution in step 4. This includes targets ({{targets_meta}}), snapshot ({{snapshot_meta}}), and timestamp ({{timestamp_meta}}) metadata. It then sends this metadata to the primary as described in {{download_meta_primary}}.

### Inventory Database

The Director SHALL use a private inventory database to store information about ECUs and vehicles. An implementor MAY use any durable database for this purpose.

The inventory database MUST record the following pieces of information:

- Per vehicle:
  - A unique identifier (such as a VIN)
- Per ECU:
  - A unique identifier (such as a serial number)
  - The vehicle identifier the ECU is associated with
  - A public key
  - The format of the public key
  - Whether the ECU is a primary or a secondary

The inventory database MAY record other information about ECUs and vehicles.

#### Time Server

The Time Server exists to inform vehicles about the current time in cryptographically secure way, since many ECUs in a vehicle will not have a reliable source of time. It receives lists of tokens from vehicles, and returns back a signed sequence that includes the token and the current time.

The Time Server SHALL receive a sequence of tokens from a vehicle representing all of its ECUs. In response, it SHALL sign each token together with the current time.

The Time Server SHALL expose a public interface allowing primaries to communicate with it. This communication MAY occur over FTP, FTPS, SFTP, HTTP, or HTTPS.

### In-vehicle implementation requirements

An Uptane-compliant ECU SHALL be able to download and verify the time, metadata, and image binaries before installing a new image.

Each ECU in a vehicle receiving over-the-air updates is either a primary or a secondary ECU. A primary ECU collects and delivers to the Director vehicle manifests ({{vehicle_version_manifest}}) containing information about which images have been installed on ECUs in the vehicle. It also downloads and verifies the latest time, metadata, and images for itself and for its secondaries. A secondary ECU downloads and verifies the latest time, metadata, and images for itself from its associated primary ECU. It also sends signed information about its installed images to its associated primary.

All ECUs MUST verify image metadata as specified in {{metadata_verification}} before installing an image or making it available to other ECUs. A primary ECU MUST perform full verification ({{full_verification}}). A secondary ECU SHOULD perform full verification if possible, and MUST perform full verification if it is safety-critical. If it is not safety-critical, it MAY perform partial verification ({{partial_verification}}) instead.

### Build-time prerequisite requirements for ECUs

For an ECU to be capable of receiving Uptane-secured updates, it MUST have the following data provisioned at the time it is manufactured or installed in the vehicle:

1. The latest copy of required Uptane metadata at the time of manufacture or install.
   - Partial verification ECUs MUST have the root and targets metadata from the director repository.
   - Full verification ECUs MUST have a complete set of metadata from both repositories (root, targets, snapshot, and timestamp), as well as the repository map file {{TAP-4}}.
2. The public key(s) of the time server.
3. An attestation of time downloaded from the time server.
4. An **ECU key**. This is a private key, unique to the ECU, used to sign ECU version manifests and decrypt images. An ECU key MAY be either a symmetric key or an asymmetric key. If it is an asymmetric key, there MAY be separate keys for encryption and signing. For the purposes of this standard, the set of private keys that an ECU uses is referred to as the ECU key (singular), even if it is actually multiple keys used for different purposes.

### Downloading and distributing updates on a primary ECU

A primary downloads, verifies, and distributes the latest time, metadata and images. To do so, it SHALL perform the following seven steps:

1. Construct and send vehicle version manifest ({{construct_manifest_primary}})
2. Download and check current time ({{check_time_primary}})
3. Download and verify metadata ({{download_meta_primary}})
4. Download and verify images ({{download_images_primary}})
5. Send latest time to secondaries ({{send_time_primary}})
6. Send metadata to secondaries ({{send_metadata_primary}})
7. Send images to secondaries ({{send_images_primary}})

### Construct and send vehicle version manifest

The primary SHALL build a *vehicle version manifest* as described in {{vehicle_version_manifest}}.

Once it has the complete manifest built, it MAY send the manifest to the director repository. However, it is not strictly required that the primary send the manifest until step three.

Secondaries MAY send their version report at any time, so that it is stored on the primary already when it wishes to check for updates. Alternatively, the

primary MAY request a version report from each secondary at the time of the update check.

#### Download and check current time

The primary SHALL download the current time from the time server, for distribution to its secondaries.

The version report from each secondary ECU (as described in {{version_report}}) contains a nonce, plus a signed ECU version report. The primary SHALL gather each of these nonces from the secondary ECUs, then send them to the time server to fetch the current time. The time server responds as described in {{time_server}}, providing a cryptographic attestation of the last known time. The primary SHALL verify that the signatures are valid, and that the time the server attests is greater than the previous attested time.

#### Download and verify metadata

The primary SHALL download metadata for all targets and perform a full verification on it as specified in {{full_verification}}.

#### Download and verify images

The primary SHALL download and verify images for itself and for all of its associated secondaries. Images SHALL be verified by checking that the hash of the image file matches the hash specified in the director's targets metadata for that image.

There may be several different filenames that all refer to the same image binary, as described in {{targets_meta}}. The primary SHALL associate each image binary with each of its possible filenames.

#### Send latest time to secondaries

The primary SHALL send the time server's latest attested time to each ECU. The secondary SHALL verify the time message, then overwrite its current time with the received time.

#### Send metadata to secondaries

The primary SHALL send the latest metadata it has downloaded to all of its associated secondaries.

Full verification secondaries SHALL keep a complete copy of all metadata. A partial verification secondary MAY keep *only* the targets metadata file from the director repository.

### Send images to secondaries

The primary SHALL send the latest image to each of its associated secondaries that have storage to receive it.

For secondaries without storage, the primary SHOULD wait for a request from the secondary to stream the new image file to it. The secondary will send the request once it has verified the metadata sent in the previous step.

### Installing images on ECUs

Before installing a new image, an ECU SHALL perform the following five steps:

1. Verify latest attested time ({{verify_time}})
2. Verify metadata ({{verify_metadata}})
3. Download latest image ({{download_image}})
4. Verify image ({{verify_image}})
5. Create and send version report ({{create_version_report}})

### Verify latest attested time

The ECU SHALL verify the latest downloaded time. To do so, it must:

1. Verify that the signatures on the downloaded time are valid,
2. Verify that the list of nonces/tokens in the downloaded time includes the token that the ECU sent in its previous version report
3. Verify that the time downloaded is greater than the previous time

If all three steps complete without error, the ECU SHALL overwrite its current attested time with the time it has just downloaded and generate a new nonce/token for the next request to the time server.

If any check fails, the ECU SHALL NOT overwrite its current attested time, and SHALL jump to the fifth step ({{create_version_report}}). The ECU MUST reuse its previous token for the next request to the time server.

### Verify metadata

The ECU SHALL verify the latest downloaded metadata ({{metadata_verification}}) using either full or partial verification. If the metadata verification fails for any reason, the ECU SHALL jump to the fifth step ({{create_version_report}}).

### Download latest image

If the ECU does not have secondary storage, it SHALL download the latest image from the primary. (If the ECU has secondary storage, it will already have the

latest image in its secondary storage as specified in {{send_images_primary}}, and should skip to the next step.) The ECU MAY first create a backup of its previous working image and store it elsewhere (e.g., the primary).

The filename used to identify the latest known image (i.e., the file to request from the primary) 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.
3. Construct the image filename using the rule in {{metadata_filename_rules}}.

When the primary responds to the download request, the ECU SHALL overwrite its current image with the downloaded image from the primary.

If any part of this step fails, the ECU SHALL jump to the fifth step ({{create_version_report}}).

#### Verify image

The ECU SHALL verify that the latest image matches the latest metadata as follows:

1. Load the latest targets metadata file from the director.
2. Find the target metadata associated with this ECU identifier.
3. Check that the hardware identifier in the metadata matches the ECUs hardware identifier.
4. Check that the release counter of the image in the previous metadata, if it exists, is less than or equal to the release counter in the latest metadata.
5. If the image is encrypted, decrypt the image with a decryption key to be chosen as follows:
    - If the ECU key is a symmetric key, the ECU SHALL use the ECU key for image decryption.
    - If the ECU key is asymmetric, the ECU SHALL check the target metadata for an encrypted symmetric key. If such a key is found, the ECU SHALL decrypt the symmetric key using its ECU key, and use the decrypted symmetric key for image decryption.
    - If the ECU key is asymmetric and there is no symmetric key in the target metadata, the ECU SHALL use its ECU key for image decryption.
6. Check that the hash of the image matches the hash in the metadata.

If the ECU has secondary storage, the checks SHOULD be performed on the image in secondary storage, before it is installed.

If any step fails, the ECU SHALL jump to the fifth step ({{create_version_report}}). If the ECU does not have secondary storage, a step fails, and the ECU created a backup of its previous working image, the ECU SHOULD now install the backup image.

### Create and send version report

The ECU SHALL create a version report as described in {{version_report}}, and send it to the primary (or simply save it to disk, if the ECU is a primary). The primary SHOULD write the version reports it receives to disk and associate them with the secondaries that sent them.

### Metadata verification

A primary ECU MUST perform full verification of metadata. A secondary ECU SHOULD perform full verification of metadata, but MAY perform partial verification instead.

If a step in the following workflows does not succeed (e.g., the update is aborted because a new metadata file was not signed), an ECU SHOULD still be able to update again in the future. Errors raised during the update process SHOULD NOT leave ECUs in an unrecoverable state.

### Partial verification

In order to perform partial verification, an ECU SHALL perform the following steps:

1. Load the latest attested time from the time server.
2. Load the latest top-level targets metadata file from the director repository.
3. Check that the metadata file has been signed by a threshold of keys specified in the previous root metadata file. If not, return an error code indicating an arbitrary software attack.
4. Check that the version number in the previous targets metadata file, if any, is less than or equal to the version number in this targets metadata file. If not, return an error code indicating a rollback attack.
5. Check that the latest attested time is lower than the expiration timestamp in this metadata file. If not, return an error code indicating a freeze attack.
6. Check that there are no delegations. If there are, return an error code.
7. Check that each ECU identifier appears only once. If not, return an error code.
8. Return an indicator of success.

### Full verification

Full verification of metadata means that the ECU checks that the targets metadata about images from the director repository matches the targets metadata about the same images from the image repository. This provides resilience to a key compromise in the system.

Full verification MAY be performed either by primary or secondary ECUs. The procedure is the same, except that secondary ECUs receive their metadata from

the primary instead of downloading it directly. In the following instructions, whenever an ECU is directed to download metadata, it applies only to primary ECUs.

A primary ECU SHALL download metadata and images following the rules specified in {{TAP-5}}, and the metadata file renaming rules specified in {{metadata_filename_rules}}.

In order to perform full verification, an ECU SHALL perform the following steps:

1. Load the map file {{TAP-4}}. If necessary, use the information therein to determine where to download metadata from.
2. Load the latest attested time from the time server.
3. Download and check the root metadata file from the director repository:
    1. Load the previous root metadata file.
    2. Update to the latest root metadata file.
    3. Let N denote the version number of the latest root metadata file (which at first could be the same as the previous root metadata file).
    4. Try downloading a new version N+1 of the root metadata file, up to some X number of bytes. The value for X is set by the implementor. For example, X may be tens of kilobytes. The filename used to download the root metadata file is of the fixed form VERSION_NUMBER.FILENAME.EXT (e.g., 42.root.json). If this file is not available, then go to step 3.5.
    5. Version N+1 of the root metadata file MUST have been signed by: (1) a threshold of keys specified in the latest root metadata file (version N), and (2) a threshold of keys specified in the new root metadata file being validated (version N+1). If version N+1 is not signed as required, discard it, abort the update cycle, and report the signature failure. On the next update cycle, begin at step 0 and version N of the root metadata file. (Checks for an arbitrary software attack.)
    6. The version number of the latest root metadata file (version N) must be less than or equal to the version number of the new root metadata file (version N+1). Effectively, this means checking that the version number signed in the new root metadata file is indeed N+1. If the version of the new root metadata file is less than the latest metadata file, discard it, abort the update cycle, and report the rollback attack. On the next update cycle, begin at step 0 and version N of the root metadata file. (Checks for a rollback attack.)
    7. Set the latest root metadata file to the new root metadata file.
    8. Repeat steps 1 to 6.
    9. Check that the latest attested time is lower than the expiration timestamp in the latest root metadata file. (Checks for a freeze attack.)
    10. If the the timestamp and / or snapshot keys have been rotated, delete the previous timestamp and snapshot metadata files. (Checks for recovery from fast-forward attacks {{MERCURY}}.)

4. Download and check the timestamp metadata file from the director repository:
    1. Download up to Y number of bytes. The value for Y is set by the implementor. For example, Y may be tens of kilobytes. The filename used to download the timestamp metadata file is of the fixed form FILENAME.EXT (e.g., timestamp.json).
    2. Check that it has been signed by the threshold of keys specified in the latest root metadata file. If the new timestamp metadata file is not properly signed, discard it, abort the update cycle, and report the signature failure. (Checks for an arbitrary software attack.)
    3. Check that the version number of the previous timestamp metadata file, if any, is less than or equal to the version number of this timestamp metadata file. If the new timestamp metadata file is older than the trusted timestamp metadata file, discard it, abort the update cycle, and report the potential rollback attack. (Checks for a rollback attack.)
    4. Check that the latest attested time is lower than the expiration timestamp in this timestamp metadata file. If the new timestamp metadata file has expired, discard it, abort the update cycle, and report the potential freeze attack. (Checks for a freeze attack.)
5. Download and check the snapshot metadata file from the director repository:
    1. Download up to the number of bytes specified in the timestamp metadata file. If consistent snapshots are not used {{metadata_filename_rules}}, then the filename used to download the snapshot metadata file is of the fixed form FILENAME.EXT (e.g., snapshot.json). Otherwise, the filename is of the form VERSION_NUMBER.FILENAME.EXT (e.g., 42.snapshot.json), where VERSION_NUMBER is the version number of the snapshot metadata file listed in the timestamp metadata file. In either case, the ECU MUST write the file to non-volatile storage as FILENAME.EXT.
    2. The hashes and version number of the new snapshot metadata file MUST match the hashes and version number listed in timestamp metadata. If hashes and version do not match, discard the new snapshot metadata, abort the update cycle, and report the failure. (Checks for a mix-and-match attack.)
    3. Check that it has been signed by the threshold of keys specified in the latest root metadata file. If the new snapshot metadata file is not signed as required, discard it, abort the update cycle, and report the signature failure. (Checks for an arbitrary software attack.)
    4. Check that the version number of the previous snapshot metadata file, if any, is less than or equal to the version number of this snapshot metadata file. If this snapshot metadata file is older than the previous snapshot metadata file, discard it, abort the update cycle, and report the potential rollback attack. (Checks for a rollback attack.)

5. Check that the version number the previous snapshot metadata file lists for each targets metadata file is less than or equal to the its version number in this snapshot metadata file. If this condition is not met, discard the new snaphot metadadata file, abort the update cycle, and report the failure. (Checks for a rollback attack.)
6. Check that each targets metadata filename listed in the previous snapshot metadata file is also listed in this snapshot metadata file. If this condition is not met, discard the new snaphot metadadata file, abort the update cycle, and report the failure. (Checks for a rollback attack.)
7. Check that the latest attested time is lower than the expiration timestamp in this snapshot metadata file. If the new snapshot metadata file is expired, discard it, abort the update cycle, and report the potential freeze attack. (Checks for a freeze attack.)

6. Download and check the targets metadata file from the director repository:
   1. Download up to either the number of bytes specified in the snapshot metadata file, or some Z number of bytes. The value for Z is set by the implementor. For example, Z may be tens of kilobytes. If consistent snapshots are not used {{metadata_filename_rules}}, then the filename used to download the targets metadata file is of the fixed form FILENAME.EXT (e.g., targets.json). Otherwise, the filename is of the form VERSION_NUMBER.FILENAME.EXT (e.g., 42.targets.json), where VERSION_NUMBER is the version number of the targets metadata file listed in the snapshot metadata file. In either case, the ECU MUST write the file to non-volatile storage as FILENAME.EXT.
   2. The hashes (if any), and version number of the new targets metadata file MUST match the latest snapshot metadata. If the new targets metadata file does not match, discard it, abort the update cycle, and report the failure. (Checks for a mix-and-match attack.)
   3. Check that it has been signed by the threshold of keys specified in the latest root metadata file. (Checks for an arbitrary software attack.)
   4. Check that the version number of the previous targets metadata file, if any, is less than or equal to the version number of this targets metadata file. (Checks for a rollback attack.)
   5. Check that the latest attested time is lower than the expiration timestamp in this targets metadata file. (Checks for a freeze attack.)
   6. Check that there are no delegations. (Targets metadata from the director MUST NOT contain delegations.)
   7. Check that no ECU identifier is represented more than once.
7. Download and check the root metadata file from the image repository as in Step 3.
8. Download and check the timestamp metadata file from the image repository as in Step 4.
9. Download and check the snapshot metadata file from the image repository as in Step 5.

10. Download and check the top-level targets metadata file from the image repository as in Step 6 (except for Steps 6.6-6.7).
11. For each image listed in the targets metadata file from the director repository, locate a targets metadata file that contains an image with exactly the same file name. For each delegated targets metadata file that is found to contain metadata for the image currently being processed, perform all of the checks in step 10. Use the following process to locate image metadata:
    1. If the top-level targets metadata file contains signed metadata about the image, return the metadata to be checked and skip to step 11.3.
    2. Recursively search the list of delegations, in order of appearance:
        1. If it is a multi-role delegation {{TAP-3}}, recursively visit each role, and check that each has signed exactly the same non-custom metadata (i.e., length and hashes) about the image. If it is all the same, return the metadata to be checked and skip to step 11.3.
        2. If it is a terminating delegation and it contains signed metadata about the image, return the metadata to be checked and skip to step 11.3. If metadata about an image is not found in a terminating delegation, return an error code indicating that the image is missing.
        3. Otherwise, continue processing the next delegation, if any. As soon as a delegation is found that contains signed metadata about the image, return the metadata to be checked and skip to step 11.3.
        4. If no signed metadata about the image can be found anywhere in the delegation graph, return an error code indicating that the image is missing.
    3. Check that the targets metadata from the image repository matches the targets metadata from the director repository:
        1. Check that the non-custom metadata (i.e., length and hashes) of the unencrypted image are the same in both sets of metadata.
        2. Check that the custom metadata (e.g., hardware identifier and release counter) are the same in both sets of metadata.
        3. 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.
12. Verify the desired image against its targets metadata.
13. If there is no targets metadata about this image, abort the update cycle and report that there is no such image.
14. Otherwise, download the image (up to the number of bytes specified in the targets metadata), and verify that its hashes match the targets metadata. (We download up to this number of bytes, because in some cases, the exact number is unknown. This may happen, for example, if an external program is used to compute the root hash of a tree of targets files, and this program does not provide the total size of all of these files.) If consistent snapshots are not used

{{metadata_filename_rules}}, then the filename used to download the image file is of the fixed form FILENAME.EXT (e.g., foobar.tar.gz). Otherwise, the filename is of the form HASH.FILENAME.EXT (e.g., c14aeb4ac9f4a8fc0d83d12482b9197452f6adf3eb710e3b1e2b79e8d14cb681.foobar.tar.gz), where HASH is one of the hashes of the targets file listed in the targets metadata file found earlier in step 4. In either case, the client MUST write the file to non-volatile storage as FILENAME.EXT.

If any step fails, the ECU MUST return an error code indicating the failure. If a check for a specific type of security attack fails (e.g. rollback, freeze, arbitrary software, etc.), the ECU SHOULD return an error code that indicates the type of attack.

If the ECU performing the verification is the primary ECU, it SHOULD also ensure 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.