

SUIT
Internet-Draft
Intended status: Informational
Expires: July 31, 2021

B. Moran
H. Tschofenig
Arm Limited
D. Brown
Linaro
M. Meriac
Consultant
January 27, 2021

A Firmware Update Architecture for Internet of Things
draft-ietf-suit-architecture-16

Abstract

Vulnerabilities in Internet of Things (IoT) devices have raised the need for a reliable and secure firmware update mechanism suitable for devices with resource constraints. Incorporating such an update mechanism is a fundamental requirement for fixing vulnerabilities but it also enables other important capabilities such as updating configuration settings as well as adding new functionality.

In addition to the definition of terminology and an architecture this document motivates the standardization of a manifest format as a transport-agnostic means for describing and protecting firmware updates.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on July 31, 2021.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

| | |
|--|----|
| 1. Introduction | 2 |
| 2. Conventions and Terminology | 5 |
| 2.1. Terms | 5 |
| 2.2. Stakeholders | 6 |
| 2.3. Functions | 7 |
| 3. Architecture | 8 |
| 4. Invoking the Firmware | 13 |
| 4.1. The Bootloader | 14 |
| 5. Types of IoT Devices | 15 |
| 5.1. Single MCU | 16 |
| 5.2. Single CPU with Secure - Normal Mode Partitioning | 16 |
| 5.3. Symmetric Multiple CPUs | 16 |
| 5.4. Dual CPU, shared memory | 16 |
| 5.5. Dual CPU, other bus | 17 |
| 6. Manifests | 17 |
| 7. Securing Firmware Updates | 19 |
| 8. Example | 20 |
| 9. IANA Considerations | 25 |
| 10. Security Considerations | 25 |
| 11. Acknowledgements | 25 |
| 12. Informative References | 26 |
| Authors' Addresses | 28 |

1. Introduction

Firmware updates can help to fix security vulnerabilities, and performing updates is an important building block in securing IoT devices. Due to rising concerns about insecure IoT devices the Internet Architecture Board (IAB) organized a 'Workshop on Internet of Things (IoT) Software Update (IOTSU)' [RFC8240] to take a look at the bigger picture. The workshop revealed a number of challenges for

developers and led to the formation of the IETF Software Updates for Internet of Things (SUIT) working group.

Developing secure Internet of Things (IoT) devices is not an easy task and supporting a firmware update solution requires skillful engineers. Once devices are deployed, firmware updates play a critical part in their lifecycle management, particularly when devices have a long lifetime, or are deployed in remote or inaccessible areas where manual intervention is cost prohibitive or otherwise difficult. Firmware updates for IoT devices are expected to work automatically, i.e. without user involvement. Conversely, non-IoT devices are expected to account for user preferences and consent when scheduling updates. Automatic updates that do not require human intervention are key to a scalable solution for fixing software vulnerabilities.

Firmware updates are done not only to fix bugs, but also to add new functionality and to reconfigure the device to work in new environments or to behave differently in an already deployed context.

The manifest specification has to allow that

- The firmware image is authenticated and integrity protected. Attempts to flash a maliciously modified firmware image or an image from an unknown, untrusted source must be prevented. In examples this document uses asymmetric cryptography because it is the preferred approach by many IoT deployments. The use of symmetric credentials is also supported and can be used by very constrained IoT devices.
- The firmware image can be confidentiality protected so that attempts by an adversary to recover the plaintext binary can be mitigated or at least made more difficult. Obtaining the firmware is often one of the first steps to mount an attack since it gives the adversary valuable insights into the software libraries used, configuration settings and generic functionality. Even though reverse engineering the binary can be a tedious process modern reverse engineering frameworks have made this task a lot easier.

Authentication and integrity protection of firmware images must be used in a deployment but the confidential protection of firmware is optional.

While the standardization work has been informed by and optimized for firmware update use cases of Class 1 devices (according to the device class definitions in RFC 7228 [RFC7228]), there is nothing in the architecture that restricts its use to only these constrained IoT devices. Moreover, this architecture is not limited to managing

firmware and software updates, but can also be applied to managing the delivery of arbitrary data, such as configuration information and keys. Unlike higher end devices, like laptops and desktop PCs, many IoT devices do not have user interfaces; and support for unattended updates is, therefore, essential for the design of a practical solution. Constrained IoT devices often use a software engineering model where a developer is responsible for creating and compiling all software running on the device into a single, monolithic firmware image. On higher end devices application software is, on the other hand, often downloaded separately and even obtained from developers different to the developers of the lower level software. The details for how to obtain those application layer software binaries then depends heavily on the platform, programming language used and the sandbox in which the software is executed.

While the IETF standardization work has been focused on the manifest format, a fully interoperable solution needs more than a standardized manifest. For example, protocols for transferring firmware images and manifests to the device need to be available as well as the status tracker functionality. Devices also require a mechanism to discover the status tracker(s) and/or firmware servers, for example using pre-configured hostnames or DNS-SD [RFC6763]. These building blocks have been developed by various organizations under the umbrella of an IoT device management solution. The LwM2M protocol [LwM2M] is one IoT device management protocol.

There are, however, several areas that (partially) fall outside the scope of the IETF and other standards organizations but need to be considered by firmware authors, as well as device and network operators. Here are some of them, as highlighted during the IOTSU workshop:

- Installing firmware updates in a robust fashion so that the update does not break the device functionality of the environment this device operates in. This requires proper testing and offering recovery strategies when a firmware update is unsuccessful.
- Making firmware updates available in a timely fashion considering the complexity of the decision making process for updating devices, potential re-certification requirements, the length of a supply chain an update needs to go through before it reaches the end customer, and the need for user consent to install updates.
- Ensuring an energy efficient design of a battery-powered IoT device because a firmware update, particularly radio communication and writing the firmware image to flash, is an energy-intensive task for a device.

- Creating incentives for device operators to use a firmware update mechanism and to demand the integration of it from IoT device vendors.
- Ensuring that firmware updates addressing critical flaws can be obtained even after a product is discontinued or a vendor goes out of business.

This document starts with a terminology followed by the description of the architecture. We then explain the bootloader and how it integrates with the firmware update mechanism. Subsequently, we offer a categorization of IoT devices in terms of their hardware capabilities relevant for firmware updates. Next, we talk about the manifest structure and how to use it to secure firmware updates. We conclude with a more detailed example.

2. Conventions and Terminology

2.1. Terms

This document uses the following terms:

- **Firmware Image:** The firmware image, or simply the "image", is a binary that may contain the complete software of a device or a subset of it. The firmware image may consist of multiple images, if the device contains more than one microcontroller. Often it is also a compressed archive that contains code, configuration data, and even the entire file system. The image may consist of a differential update for performance reasons.

The terms, firmware image, firmware, and image, are used in this document and are interchangeable. We use the term application firmware image to differentiate it from a firmware image that contains the bootloader. An application firmware image, as the name indicates, contains the application program often including all the necessary code to run it (such as protocol stacks, and embedded operating system).

- **Manifest:** The manifest contains meta-data about the firmware image. The manifest is protected against modification and provides information about the author.
- **Microcontroller (MCU for microcontroller unit):** An MCU is a compact integrated circuit designed for use in embedded systems. A typical microcontroller includes a processor, memory (RAM and flash), input/output (I/O) ports and other features connected via some bus on a single chip. The term 'system on chip (SoC)' is

often used interchangeably with MCU, but MCU tends to imply more limited peripheral functions.

- Rich Execution Environment (REE): An environment that is provided and governed by a typical OS (e.g., Linux, Windows, Android, iOS), potentially in conjunction with other supporting operating systems and hypervisors; it is outside of the TEE. This environment and applications running on it are considered un-trusted.
- Software: Similar to firmware, but typically dynamically loaded by an Operating System. Used interchangeably with firmware in this document.
- System on Chip (SoC): An SoC is an integrated circuit that contains all components of a computer, such as CPU, memory, input/output ports, secondary storage, a bus to connect the components, and other hardware blocks of logic.
- Trust Anchor: A trust anchor, as defined in [RFC6024], represents an authoritative entity via a public key and associated data. The public key is used to verify digital signatures, and the associated data is used to constrain the types of information for which the trust anchor is authoritative.
- Trust Anchor Store: A trust anchor store, as defined in [RFC6024], is a set of one or more trust anchors stored in a device. A device may have more than one trust anchor store, each of which may be used by one or more applications. A trust anchor store must resist modification against unauthorized insertion, deletion, and modification.
- Trusted Applications (TAs): An application component that runs in a TEE.
- Trusted Execution Environments (TEEs): An execution environment that runs alongside of, but is isolated from, an REE. For more information about TEEs see [I-D.ietf-teeep-architecture].

2.2. Stakeholders

The following stakeholders are used in this document:

- Author: The author is the entity that creates the firmware image. There may be multiple authors involved in producing firmware running on an IoT device. Section 5 talks about those IoT device deployment cases.

- Device Operator: The device operator is responsible for the day-to-day operation of a fleet of IoT devices. Customers of IoT devices, as the owners of IoT devices – such as enterprise customers or end users – interact with their IoT devices indirectly through the device operator via web or smart phone apps.
- Network Operator: The network operator is responsible for the operation of a network to which IoT devices connect.
- Trust Provisioning Authority (TPA): The TPA distributes trust anchors and authorization policies to devices and various stakeholders. The TPA may also delegate rights to stakeholders. Typically, the Original Equipment Manufacturer (OEM) or Original Design Manufacturer (ODM) will act as a TPA, however complex supply chains may require a different design. In some cases, the TPA may decide to remain in full control over the firmware update process of their products.
- User: The end-user of a device. The user may interact with devices via web or smart phone apps, as well as through direct user interfaces.

2.3. Functions

- (IoT) Device: A device refers to the entire IoT product, which consists of one or many MCUs, sensors and/or actuators. Many IoT devices sold today contain multiple MCUs and therefore a single device may need to obtain more than one firmware image and manifest to successfully perform an update.
- Status Tracker: The status tracker has a client and a server component and performs three tasks: 1) It communicates the availability of a new firmware version. This information will flow from the server to the client.
2) It conveys information about software and hardware characteristics of the device. The information flow is from the client to the server.
3) It can remotely trigger the firmware update process. The information flow is from the server to the client.

For example, a device operator may want to read the installed firmware version number running on the device and information about available flash memory. Once an update has been triggered, the device operator may want to obtain information about the state of the firmware update. If errors occurred, the device operator may want to troubleshoot problems by first obtaining diagnostic information (typically using a device management protocol).

We make no assumptions about where the server-side component is deployed. The deployment of status trackers is flexible: they may be found at cloud-based servers or on-premise servers, or they may be embedded in edge computing devices. A status tracker server component may even be deployed on an IoT device. For example, if the IoT device contains multiple MCUs, then the main MCU may act as a status tracker towards the other MCUs. Such deployment is useful when updates have to be synchronized across MCUs.

The status tracker may be operated by any suitable stakeholder; typically the Author, Device Operator, or Network Operator.

- **Firmware Consumer:** The firmware consumer is the recipient of the firmware image and the manifest. It is responsible for parsing and verifying the received manifest and for storing the obtained firmware image. The firmware consumer plays the role of the update component on the IoT device, typically running in the application firmware. It interacts with the firmware server and with the status tracker client (locally).
- **Firmware Server:** The firmware server stores firmware images and manifests and distributes them to IoT devices. Some deployments may require a store-and-forward concept, which requires storing the firmware images/manifests on more than one entity before they reach the device. There is typically some interaction between the firmware server and the status tracker and these two entities are often physically separated on different devices for scalability reasons.
- **Bootloader:** A bootloader is a piece of software that is executed once a microcontroller has been reset. It is responsible for deciding what code to execute.

3. Architecture

More devices today than ever before are connected to the Internet, which drives the need for firmware updates to be provided over the Internet rather than through traditional interfaces, such as USB or RS-232. Sending updates over the Internet requires the device to fetch the new firmware image as well as the manifest.

Hence, the following components are necessary on a device for a firmware update solution:

- the Internet protocol stack for firmware downloads. Because firmware images are often multiple kilobytes, sometimes exceeding one hundred kilobytes, for low-end IoT devices and even several megabytes for IoT devices running full-fledged operating systems

like Linux, the protocol mechanism for retrieving these images needs to offer features like congestion control, flow control, fragmentation and reassembly, and mechanisms to resume interrupted or corrupted transfers.

- the capability to write the received firmware image to persistent storage (most likely flash memory).
- a manifest parser with code to verify a digital signature or a message authentication code.
- the ability to unpack, to decompress and/or to decrypt the received firmware image.
- a status tracker.

The features listed above are most likely offered by code in the application firmware image running on the device rather than by the bootloader itself. Note that cryptographic algorithms will likely run in a trusted execution environment, on a separate MCU, in a hardware security module, or in a secure element rather than in the same context with the application code.

Figure 1 shows the architecture where a firmware image is created by an author, and made available to a firmware server. For security reasons, the author will not have the permissions to upload firmware images to the firmware server and to initiate an update directly. Instead, authors will make firmware images available to the device operators. Note that there may be a longer supply chain involved to pass software updates from the author all the way to the party that can then finally make a decision to deploy it with IoT devices.

As a first step in the firmware update process, the status tracker server needs to inform the status tracker client that a new firmware update is available. This can be accomplished via polling (client-initiated), push notifications (server-initiated), or more complex mechanisms (such as a hybrid approach):

- Client-initiated updates take the form of a status tracker client proactively checking (polling) for updates.
- With Server-initiated updates the server-side component of the status tracker learns about a new firmware version and determines which devices qualify for a firmware update. Once the relevant devices have been selected, the status tracker informs these devices and the firmware consumers obtain those images and manifests. Server-initiated updates are important because they allow a quick response time. Note that in this mode the client-

side status tracker needs to be reachable by the server-side component. This may require devices to keep reachability information on the server-side up-to-date and state at NATs and stateful packet filtering firewalls alive.

- Using a hybrid approach the server-side of the status tracker pushes notifications of availability of an update to the client side and requests the firmware consumer to pull the manifest and the firmware image from the firmware server.

Once the device operator triggers an update via the status tracker, it will keep track of the update process on the device. This allows the device operator to know what devices have received an update and which of them are still pending an update.

Firmware images can be conveyed to devices in a variety of ways, including USB, UART, WiFi, BLE, low-power WAN technologies, mesh networks and many more. At the application layer a variety of protocols are also available: MQTT, CoAP, and HTTP are the most popular application layer protocols used by IoT devices. This architecture does not make assumptions about how the firmware images are distributed to the devices and therefore aims to support all these technologies.

In some cases it may be desirable to distribute firmware images using a multicast or broadcast protocol. This architecture does not make recommendations for any such protocol. However, given that broadcast may be desirable for some networks, updates must cause the least disruption possible both in metadata and firmware transmission. For an update to be broadcast friendly, it cannot rely on link layer, network layer, or transport layer security. A solution has to rely on security protection applied to the manifest and firmware image instead. In addition, the same manifest must be deliverable to many devices, both those to which it applies and those to which it does not, without a chance that the wrong device will accept the update. Considerations that apply to network broadcasts apply equally to the use of third-party content distribution networks for payload distribution.

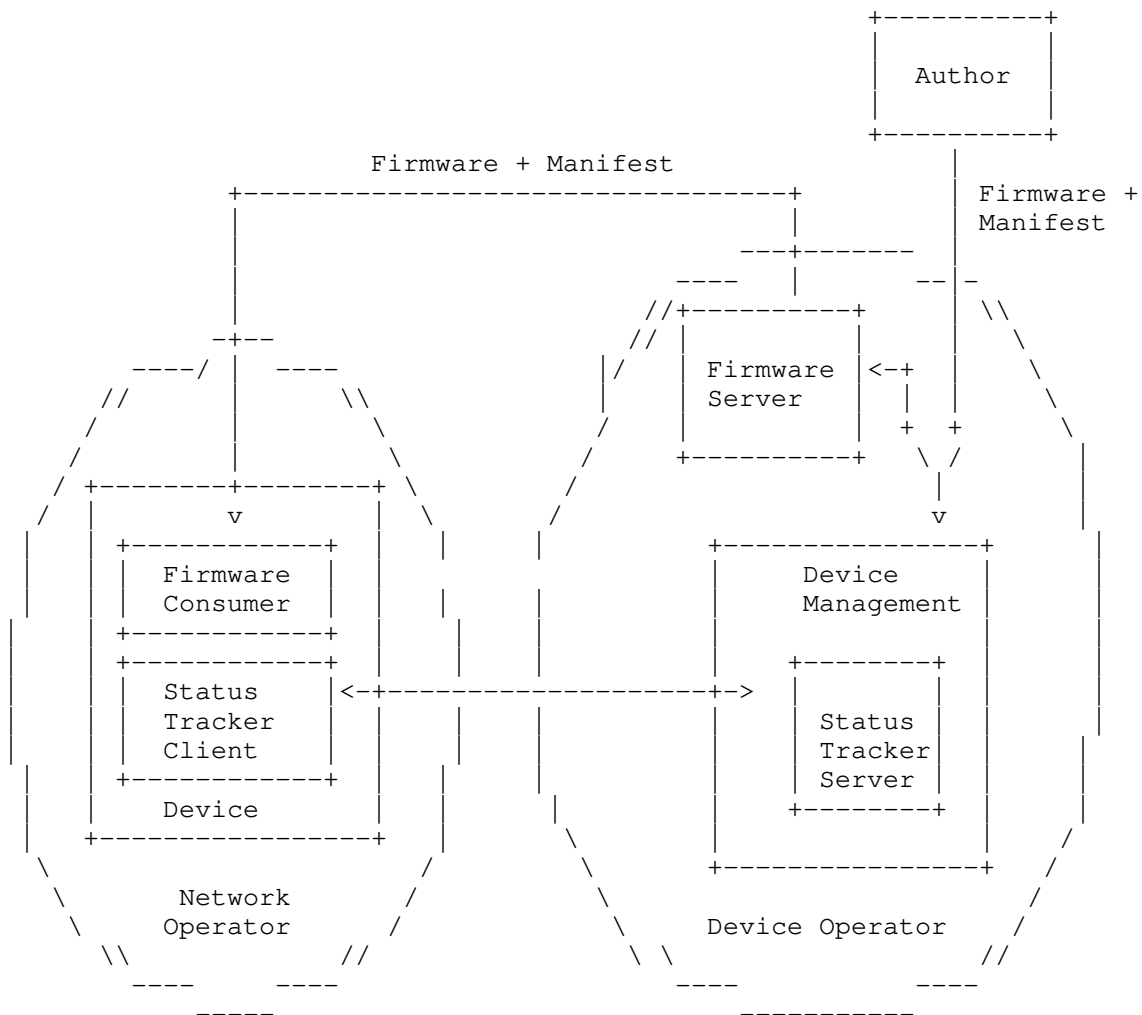


Figure 1: Architecture.

Firmware images and manifests may be conveyed as a bundle or detached. The manifest format must support both approaches.

For distribution as a bundle, the firmware image is embedded into the manifest. This is a useful approach for deployments where devices are not connected to the Internet and cannot contact a dedicated firmware server for the firmware download. It is also applicable when the firmware update happens via a USB sticks or short range radio technologies (such as Bluetooth Smart).

Alternatively, the manifest is distributed detached from the firmware image. Using this approach, the firmware consumer is presented with the manifest first and then needs to obtain one or more firmware images as dictated in the manifest.

The pre-authorisation step involves verifying whether the entity signing the manifest is indeed authorized to perform an update. The firmware consumer must also determine whether it should fetch and process a firmware image, which is referenced in a manifest.

A dependency resolution phase is needed when more than one component can be updated or when a differential update is used. The necessary dependencies must be available prior to installation.

The download step is the process of acquiring a local copy of the firmware image. When the download is client-initiated, this means that the firmware consumer chooses when a download occurs and initiates the download process. When a download is server-initiated, this means that the status tracker tells the device when to download or that it initiates the transfer directly to the firmware consumer. For example, a download from an HTTP/1.1-based firmware server is client-initiated. Pushing a manifest and firmware image to the Package resource of the LwM2M Firmware Update object [LwM2M] is server-initiated update.

If the firmware consumer has downloaded a new firmware image and is ready to install it, to initiate the installation, it may

- either need to wait for a trigger from the status tracker,
- or trigger the update automatically,
- or go through a more complex decision making process to determine

the appropriate timing for an update. Sometimes the final decision may require confirmation of the user of the device for safety reasons.

Installation is the act of processing the payload into a format that the IoT device can recognize and the bootloader is responsible for then booting from the newly installed firmware image. This process is different when a bootloader is not involved. For example, when an application is updated in a full-featured operating system, the updater may halt and restart the application in isolation. Devices must not fail when a disruption, such as a power failure or network interruption, occurs during the update process.

4. Invoking the Firmware

Section 3 describes the steps for getting the firmware image and the manifest from the author to the firmware consumer on the IoT device. Once the firmware consumer has retrieved and successfully processed the manifest and the firmware image it needs to invoke the new firmware image. This is managed in many different ways, depending on the type of device, but it typically involves halting the current version of the firmware, handing control over to a firmware with a higher privilege/trust level (the firmware verifier), verifying the new firmware's authenticity & integrity, and then invoking it.

In an execute-in-place microcontroller, this is often done by rebooting into a bootloader (simultaneously halting the application & handing over to the higher privilege level) then executing a secure boot process (verifying and invoking the new image).

In a rich OS, this may be done by halting one or more processes, then invoking new applications. In some OSs, this implicitly involves the kernel verifying the code signatures on the new applications.

The invocation process is security sensitive. An attacker will typically try to retrieve a firmware image from the device for reverse engineering or will try to get the firmware verifier to execute an attacker-modified firmware image. The firmware verifier will therefore have to perform security checks on the firmware image before it can be invoked. These security checks by the firmware verifier happen in addition to the security checks that took place when the firmware image and the manifest were downloaded by the firmware consumer.

The overlap between the firmware consumer and the firmware verifier functionality comes in two forms, namely

- A firmware verifier must verify the firmware image it boots as part of the secure boot process. Doing so requires meta-data to be stored alongside the firmware image so that the firmware verifier can cryptographically verify the firmware image before booting it to ensure it has not been tampered with or replaced. This meta-data used by the firmware verifier may well be the same manifest obtained with the firmware image during the update process.
- An IoT device needs a recovery strategy in case the firmware update / invocation process fails. The recovery strategy may include storing two or more application firmware images on the device or offering the ability to invoke a recovery image to perform the firmware update process again using firmware updates

over serial, USB or even wireless connectivity like Bluetooth Smart. In the latter case the firmware consumer functionality is contained in the recovery image and requires the necessary functionality for executing the firmware update process, including manifest parsing.

While this document assumes that the firmware verifier itself is distinct from the role of the firmware consumer and therefore does not manage the firmware update process, this is not a requirement and these roles may be combined in practice.

Using a bootloader as the firmware verifier requires some special considerations, particularly when the bootloader implements the robustness requirements identified by the IOTSU workshop [RFC8240].

4.1. The Bootloader

In most cases the MCU must restart in order to hand over control to the bootloader. Once the MCU has initiated a restart, the bootloader determines whether a newly available firmware image should be executed. If the bootloader concludes that the newly available firmware image is invalid, a recovery strategy is necessary. There are only two approaches for recovering from an invalid firmware: either the bootloader must be able to select a different, valid firmware, or it must be able to obtain a new, valid firmware. Both of these approaches have implications for the architecture of the update system.

Assuming the first approach, there are (at least) three firmware images available on the device:

- First, the bootloader is also firmware. If a bootloader is updatable then its firmware image is treated like any other application firmware image.
- Second, the firmware image that has to be replaced is still available on the device as a backup in case the freshly downloaded firmware image does not boot or operate correctly.
- Third, there is the newly downloaded firmware image.

Therefore, the firmware consumer must know where to store the new firmware. In some cases, this may be implicit, for example replacing the least-recently-used firmware image. In other cases, the storage location of the new firmware must be explicit, for example when a device has one or more application firmware images and a recovery image with limited functionality, sufficient only to perform an update.

Since many low end IoT devices do not use position-independent code, either the bootloader needs to copy the newly downloaded application firmware image into the location of the old application firmware image and vice versa or multiple versions of the firmware need to be prepared for different locations.

In general, it is assumed that the bootloader itself, or a minimal part of it, will not be updated since a failed update of the bootloader poses a reliability risk.

For a bootloader to offer a secure boot functionality it needs to implement the following functionality:

- The bootloader needs to fetch the manifest from nonvolatile storage and parse its contents for subsequent cryptographic verification.
- Cryptographic libraries with hash functions, digital signatures (for asymmetric crypto), message authentication codes (for symmetric crypto) need to be accessible.
- The device needs to have a trust anchor store to verify the digital signature. (Alternatively, access to a key store for use with the message authentication code.)
- There must be an ability to expose boot process-related data to the application firmware (such as to the status tracker). This allows sharing information about the current firmware version, and the status of the firmware update process and whether errors have occurred.
- Produce boot measurements as part of an attestation solution. See [I-D.ietf-rats-architecture] for more information. (optional)
- The bootloader must be able to decrypt firmware images, in case confidentiality protection was applied. This requires a solution for key management. (optional)

5. Types of IoT Devices

There are billions of MCUs used in devices today produced by a large number of silicon manufacturers. While MCUs can vary significantly in their characteristics, there are a number of similiarities allowing us to categorize in groups.

The firmware update architecture, and the manifest format in particular, needs to offer enough flexibility to cover these common deployment cases.

5.1. Single MCU

The simplest, and currently most common, architecture consists of a single MCU along with its own peripherals. These SoCs generally contain some amount of flash memory for code and fixed data, as well as RAM for working storage. A notable characteristic of these SoCs is that the primary code is generally execute in place (XIP). Due to the non-relocatable nature of the code, the firmware image needs to be placed in a specific location in flash since the code cannot be executed from an arbitrary location in flash. Hence, when the firmware image is updated it is necessary to swap the old and the new image.

5.2. Single CPU with Secure - Normal Mode Partitioning

Another configuration consists of a similar architecture to the previous, with a single CPU. However, this CPU supports a security partitioning scheme that allows memory (in addition to other things) to be divided into secure and normal mode. There will generally be two images, one for secure mode, and one for normal mode. In this configuration, firmware upgrades will generally be done by the CPU in secure mode, which is able to write to both areas of the flash device. In addition, there are requirements to be able to update either image independently, as well as to update them together atomically, as specified in the associated manifests.

5.3. Symmetric Multiple CPUs

In more complex SoCs with symmetric multi-processing support, advanced operating systems, such as Linux, are often used. These SoCs frequently use an external storage medium, such as raw NAND flash or eMMC. Due to the higher quantity of resources, these devices are often capable of storing multiple copies of their firmware images and selecting the most appropriate one to boot. Many SoCs also support bootloaders that are capable of updating the firmware image, however this is typically a last resort because it requires the device to be held in the bootloader while the new firmware is downloaded and installed, which results in down-time for the device. Firmware updates in this class of device are typically not done in-place.

5.4. Dual CPU, shared memory

This configuration has two or more heterogeneous CPUs in a single SoC that share memory (flash and RAM). Generally, there will be a mechanism to prevent one CPU from unintentionally accessing memory currently allocated to the other. Upgrades in this case will

typically be done by one of the CPUs, and is similar to the single CPU with secure mode.

5.5. Dual CPU, other bus

This configuration has two or more heterogeneous CPUs, each having their own memory. There will be a communication channel between them, but it will be used as a peripheral, not via shared memory. In this case, each CPU will have to be responsible for its own firmware upgrade. It is likely that one of the CPUs will be considered the primary CPU, and will direct the other CPU to do the upgrade. This configuration is commonly used to offload specific work to other CPUs. Firmware dependencies are similar to the other solutions above, sometimes allowing only one image to be upgraded, other times requiring several to be upgraded atomically. Because the updates are happening on multiple CPUs, upgrading the two images atomically is challenging.

6. Manifests

In order for a firmware consumer to apply an update, it has to make several decisions using manifest-provided information and data available on the device itself. For more detailed information and a longer list of information elements in the manifest consult the information model specification [I-D.ietf-suit-information-model], which offers justifications for each element, and the manifest specification [I-D.ietf-suit-manifest] for details about how this information is included in the manifest.

Table 1 provides examples of decisions to be made.

| Decision | Information Elements |
|--|--|
| Should I trust the author of the firmware? | Trust anchors and authorization policies on the device |
| Has the firmware been corrupted? | Digital signature and MAC covering the firmware image |
| Does the firmware update apply to this device? | Conditions with Vendor ID, Class ID and Device ID |
| Is the update older than the active firmware? | Sequence number in the manifest (1) |
| When should the device apply the update? | Wait directive |
| How should the device apply the update? | Manifest commands |
| What kind of firmware binary is it? | Unpack algorithms to interpret a format. |
| Where should the update be obtained? | Dependencies on other manifests and firmware image URI in Manifest |
| Where should the firmware be stored? | Storage Location and Component Identifier |

Table 1: Firmware Update Decisions.

(1): A device presented with an old, but valid manifest and firmware must not be tricked into installing such firmware since a vulnerability in the old firmware image may allow an attacker to gain control of the device.

Keeping the code size and complexity of a manifest parsers small is important for constrained IoT devices. Since the manifest parsing code may also be used by the bootloader it can be part of the trusted computing base.

A manifest may be used to protect not only firmware images but also configuration data such as network credentials or personalization data related to firmware or software. Personalization data demonstrates the need for confidentiality to be maintained between two or more stakeholders that both deliver images to the same device.

Personalization data is used with Trusted Execution Environments (TEEs), which benefit from a protocol for managing the lifecycle of trusted applications (TAs) running inside a TEE. TEEs may obtain TAs from different authors and those TAs may require personalization data, such as payment information, to be securely conveyed to the TEE. The TA's author does not want to expose the TA's code to any other stakeholder or third party. The user does not want to expose the payment information to any other stakeholder or third party.

7. Securing Firmware Updates

Using firmware updates to fix vulnerabilities in devices is important but securing this update mechanism is equally important since security problems are exacerbated by the update mechanism: update is essentially authorized remote code execution, so any security problems in the update process expose that remote code execution system. Failure to secure the firmware update process will help attackers to take control over devices.

End-to-end security mechanisms are used to protect the firmware image and the manifest. The following assumptions are made to allow the firmware consumer to verify the received firmware image and manifest before updating software:

- Authentication ensures that the device can cryptographically identify the author(s) creating firmware images and manifests. Authenticated identities may be used as input to the authorization process. Not all entities creating and signing manifests have the same permissions. A device needs to determine whether the requested action is indeed covered by the permission of the party that signed the manifest. Informing the device about the permissions of the different parties also happens in an out-of-band fashion and is a duty of the Trust Provisioning Authority.
- Integrity protection ensures that no third party can modify the manifest or the firmware image. To accept an update, a device needs to verify the signature covering the manifest. There may be one or multiple manifests that need to be validated, potentially signed by different parties. The device needs to be in possession of the trust anchors to verify those signatures. Installing trust anchors to devices via the Trust Provisioning Authority happens in an out-of-band fashion prior to the firmware update process.
- For confidentiality protection of the firmware image, it must be done in such a way that the intended firmware consumer(s), other authorized parties, and no one else can decrypt it. The information that is encrypted individually for each device/recipient must be done in a way that is usable with Content

Distribution Networks, bulk storage, and broadcast protocols. For confidentiality protection of firmware images the author needs to be in possession of the certificate/public key or a pre-shared key of a device. The use of confidentiality protection of firmware images is optional.

A manifest specification must support different cryptographic algorithms and algorithm extensibility. Moreover, since RSA- and ECC-based signature schemes may become vulnerable to quantum-accelerated key extraction in the future, unchangeable bootloader code in ROM is recommended to use post-quantum secure signature schemes such as hash-based signatures [RFC8778]. A bootloader author must carefully consider the service lifetime of their product and the time horizon for quantum-accelerated key extraction. The worst-case estimate, at time of writing, for the time horizon to key extraction with quantum acceleration is approximately 2030, based on current research [quantum-factorization].

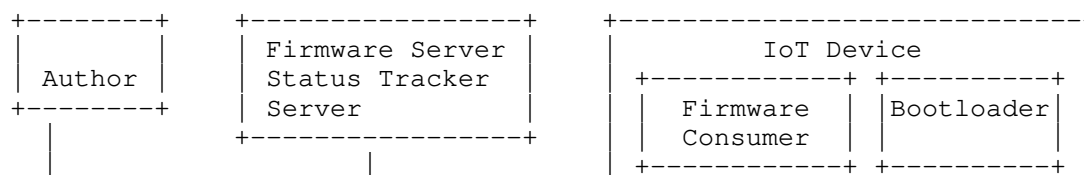
When a device obtains a monolithic firmware image from a single author without any additional approval steps, the authorization flow is relatively simple. There are, however, other cases where more complex policy decisions need to be made before updating a device.

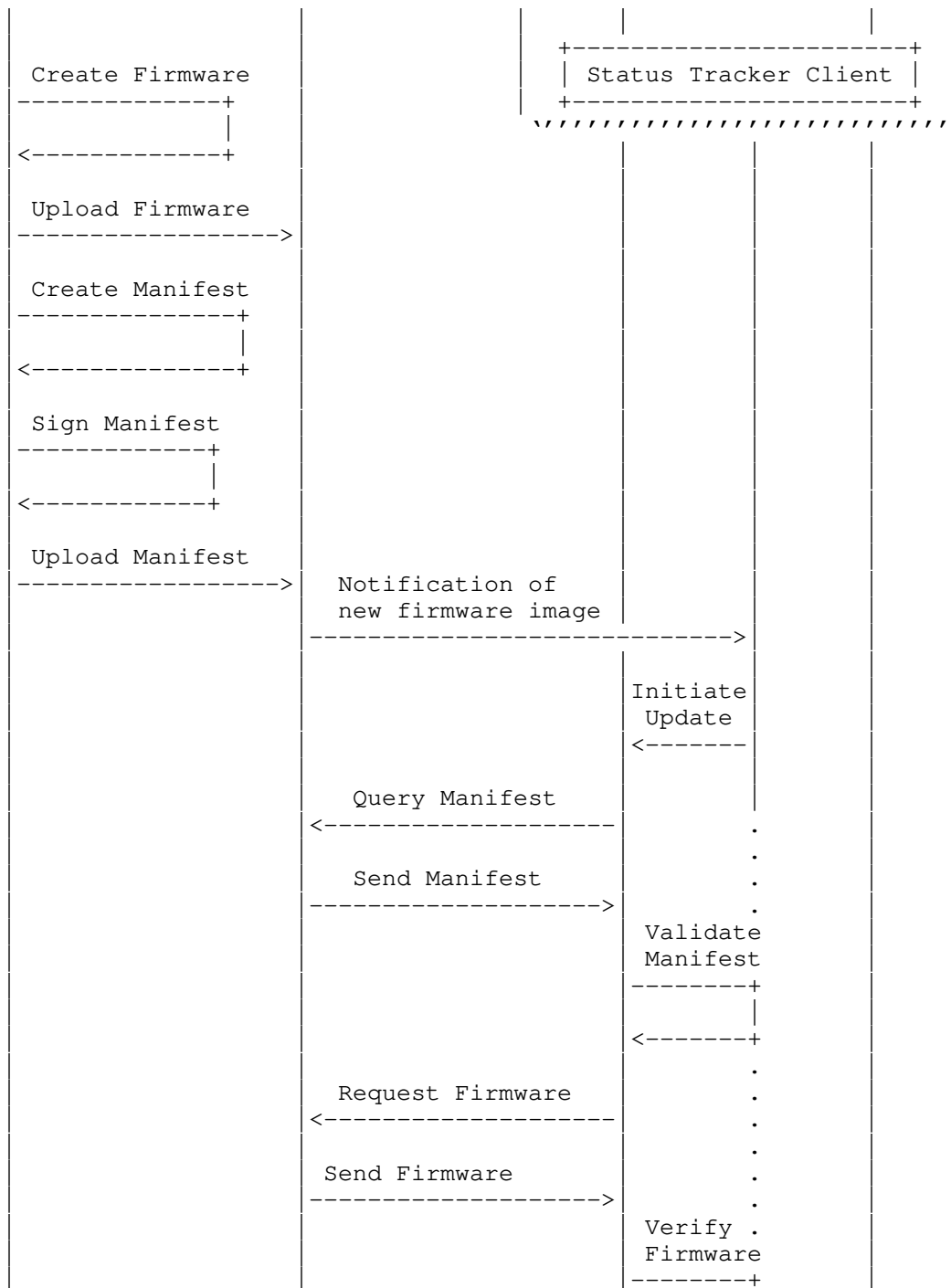
In this architecture the authorization policy is separated from the underlying communication architecture. This is accomplished by separating the entities from their permissions. For example, an author may not have the authority to install a firmware image on a device in critical infrastructure without the authorization of a device operator. In this case, the device may be programmed to reject firmware updates unless they are signed both by the firmware author and by the device operator.

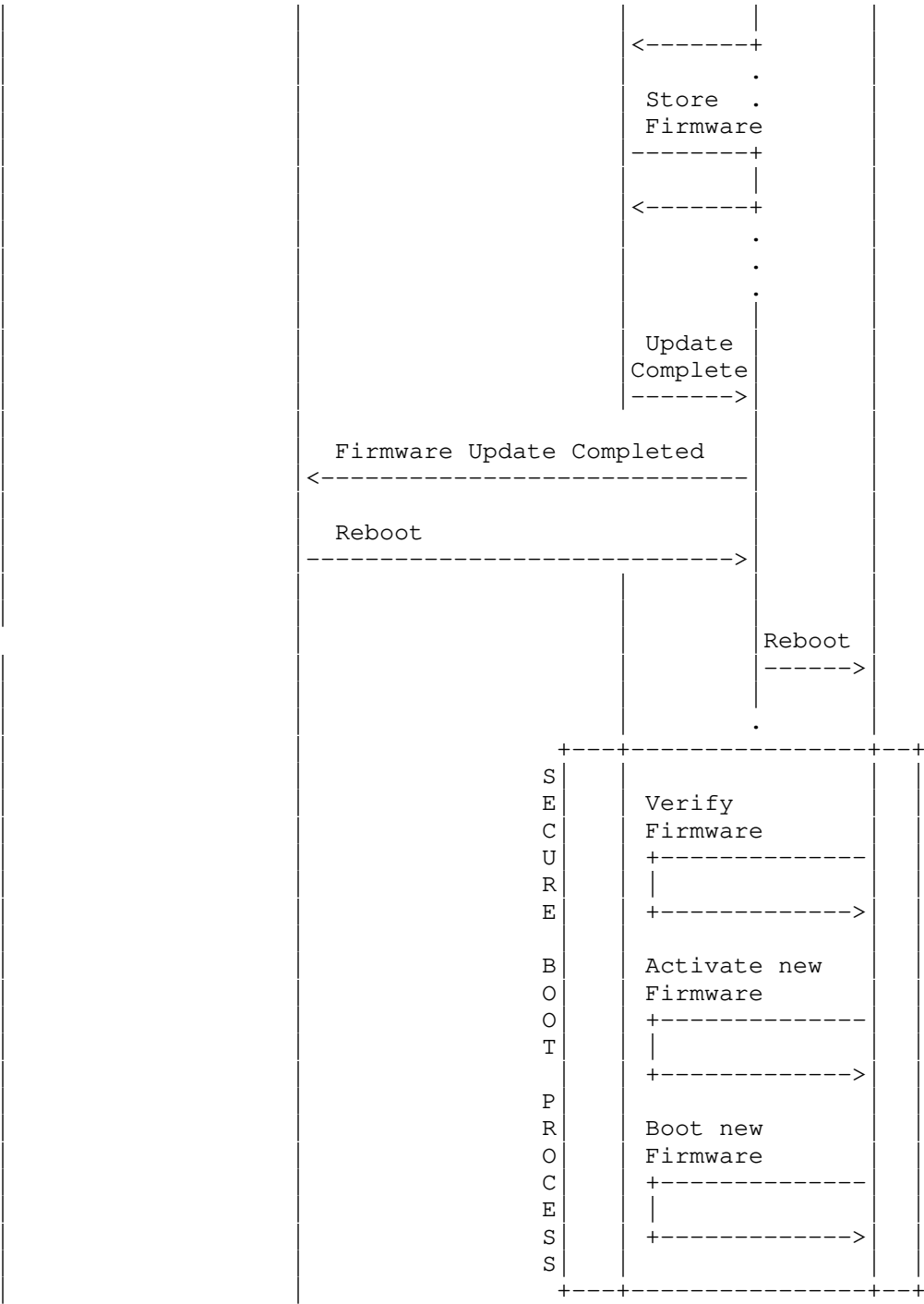
Alternatively, a device may trust precisely one entity, which does all permission management and coordination. This entity allows the device to offload complex permissions calculations for the device.

8. Example

Figure 2 illustrates an example message flow for distributing a firmware image to a device. The firmware and manifest are stored on the same firmware server and distributed in a detached manner.







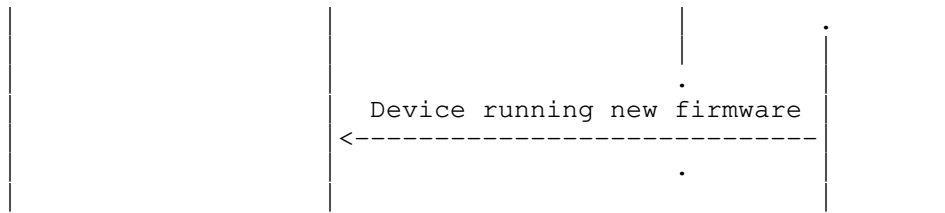
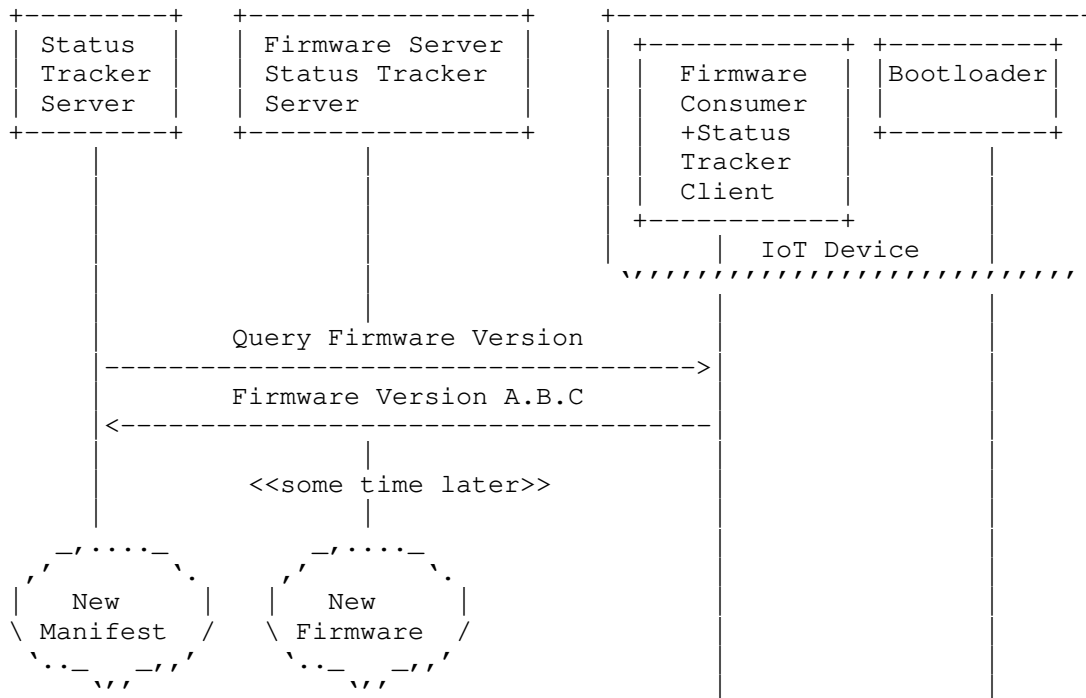


Figure 2: First Example Flow for a Firmware Update.

Figure 3 shows an exchange that starts with the status tracker querying the device for its current firmware version. Later, a new firmware version becomes available and since this device is running an older version the status tracker server interacts with the device to initiate an update.

The manifest and the firmware are stored on different servers in this example. When the device processes the manifest it learns where to download the new firmware version. The firmware consumer downloads the firmware image with the newer version X.Y.Z after successful validation of the manifest. Subsequently, a reboot is initiated and the secure boot process starts. Finally, the device reports the successful boot of the new firmware version.



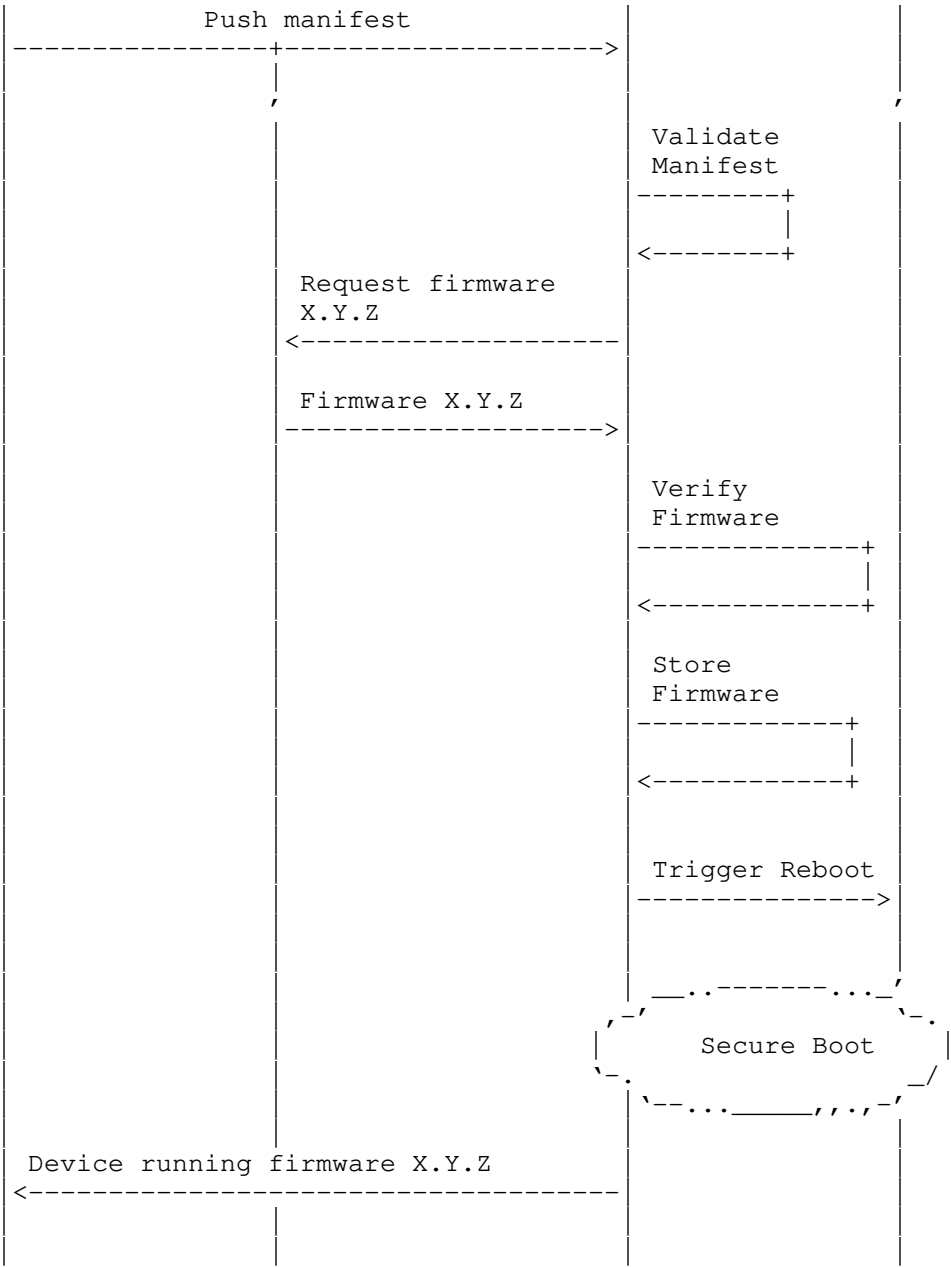


Figure 3: Second Example Flow for a Firmware Update.

9. IANA Considerations

This document does not require any actions by IANA.

10. Security Considerations

This document describes terminology, requirements and an architecture for firmware updates of IoT devices. The content of the document is thereby focused on improving security of IoT devices via firmware update mechanisms and informs the standardization of a manifest format.

An in-depth examination of the security considerations of the architecture is presented in [I-D.ietf-suit-information-model].

11. Acknowledgements

We would like to thank the following persons for their feedback:

- Geraint Luff
- Amyas Phillips
- Dan Ros
- Thomas Eichinger
- Michael Richardson
- Emmanuel Baccelli
- Ned Smith
- Jim Schaad
- Carsten Bormann
- Cullen Jennings
- Olaf Bergmann
- Suhas Nandakumar
- Phillip Hallam-Baker
- Marti Bolivar
- Andrzej Puzdrowski

- Markus Gueller
- Henk Birkholz
- Jintao Zhu
- Takeshi Takahashi
- Jacob Beningo
- Kathleen Moriarty
- Bob Briscoe
- Roman Danyliw
- Brian Carpenter
- Theresa Enghardt
- Rich Salz
- Mohit Sethi
- Eric Vyncke
- Alvaro Retana
- Barry Leiba
- Benjamin Kaduk
- Martin Duke
- Robert Wilton

We would also like to thank the WG chairs, Russ Housley, David Waltermire, and Dave Thaler, for their support and their reviews.

12. Informative References

- [I-D.ietf-rats-architecture]
Birkholz, H., Thaler, D., Richardson, M., Smith, N., and
W. Pan, "Remote Attestation Procedures Architecture",
draft-ietf-rats-architecture-08 (work in progress),
December 2020.

- [I-D.ietf-suit-information-model]
Moran, B., Tschofenig, H., and H. Birkholz, "An Information Model for Firmware Updates in IoT Devices", draft-ietf-suit-information-model-08 (work in progress), October 2020.
- [I-D.ietf-suit-manifest]
Moran, B., Tschofenig, H., Birkholz, H., and K. Zandberg, "A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest", draft-ietf-suit-manifest-11 (work in progress), December 2020.
- [I-D.ietf-teep-architecture]
Pei, M., Tschofenig, H., Thaler, D., and D. Wheeler, "Trusted Execution Environment Provisioning (TEEP) Architecture", draft-ietf-teep-architecture-13 (work in progress), November 2020.
- [LwM2M] OMA, ., "Lightweight Machine to Machine Technical Specification, Version 1.0.2", February 2018, <http://www.openmobilealliance.org/release/LightweightM2M/V1_0_2-20180209-A/OMA-TS-LightweightM2M-V1_0_2-20180209-A.pdf>.
- [quantum-factorization]
Jiang, S., Britt, K., McCaskey, A., Humble, T., and S. Kais, "Quantum Annealing for Prime Factorization", December 2018, <<https://www.nature.com/articles/s41598-018-36058-z>>.
- [RFC6024] Reddy, R. and C. Wallace, "Trust Anchor Management Requirements", RFC 6024, DOI 10.17487/RFC6024, October 2010, <<https://www.rfc-editor.org/info/rfc6024>>.
- [RFC6763] Cheshire, S. and M. Krochmal, "DNS-Based Service Discovery", RFC 6763, DOI 10.17487/RFC6763, February 2013, <<https://www.rfc-editor.org/info/rfc6763>>.
- [RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <<https://www.rfc-editor.org/info/rfc7228>>.
- [RFC8240] Tschofenig, H. and S. Farrell, "Report from the Internet of Things Software Update (IoTSU) Workshop 2016", RFC 8240, DOI 10.17487/RFC8240, September 2017, <<https://www.rfc-editor.org/info/rfc8240>>.

[RFC8778] Housley, R., "Use of the HSS/LMS Hash-Based Signature Algorithm with CBOR Object Signing and Encryption (COSE)", RFC 8778, DOI 10.17487/RFC8778, April 2020, <<https://www.rfc-editor.org/info/rfc8778>>.

Authors' Addresses

Brendan Moran
Arm Limited

EMail: Brendan.Moran@arm.com

Hannes Tschofenig
Arm Limited

EMail: hannes.tschofenig@arm.com

David Brown
Linaro

EMail: david.brown@linaro.org

Milosch Meriac
Consultant

EMail: milosch@meriac.com

SUIT
Internet-Draft
Intended status: Informational
Expires: January 9, 2022

B. Moran
H. Tschofenig
Arm Limited
H. Birkholz
Fraunhofer SIT
July 08, 2021

A Manifest Information Model for Firmware Updates in IoT Devices
draft-ietf-suit-information-model-13

Abstract

Vulnerabilities with Internet of Things (IoT) devices have raised the need for a reliable and secure firmware update mechanism that is also suitable for constrained devices. Ensuring that devices function and remain secure over their service life requires such an update mechanism to fix vulnerabilities, to update configuration settings, as well as adding new functionality.

One component of such a firmware update is a concise and machine-processable meta-data document, or manifest, that describes the firmware image(s) and offers appropriate protection. This document describes the information that must be present in the manifest.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 9, 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

| | |
|--|----|
| 1. Introduction | 5 |
| 2. Requirements and Terminology | 6 |
| 2.1. Requirements Notation | 6 |
| 2.2. Terminology | 6 |
| 3. Manifest Information Elements | 6 |
| 3.1. Version ID of the Manifest Structure | 7 |
| 3.2. Monotonic Sequence Number | 7 |
| 3.3. Vendor ID | 7 |
| 3.4. Class ID | 8 |
| 3.4.1. Example 1: Different Classes | 9 |
| 3.4.2. Example 2: Upgrading Class ID | 10 |
| 3.4.3. Example 3: Shared Functionality | 10 |
| 3.4.4. Example 4: White-labelling | 10 |
| 3.5. Precursor Image Digest Condition | 11 |
| 3.6. Required Image Version List | 11 |
| 3.7. Expiration Time | 11 |
| 3.8. Payload Format | 12 |
| 3.9. Processing Steps | 12 |
| 3.10. Storage Location | 12 |
| 3.10.1. Example 1: Two Storage Locations | 13 |
| 3.10.2. Example 2: File System | 13 |
| 3.10.3. Example 3: Flash Memory | 13 |
| 3.11. Component Identifier | 13 |
| 3.12. Payload Indicator | 13 |
| 3.13. Payload Digests | 14 |
| 3.14. Size | 14 |
| 3.15. Manifest Envelope Element: Signature | 14 |
| 3.16. Additional Installation Instructions | 15 |
| 3.17. Manifest text information | 15 |
| 3.18. Aliases | 15 |
| 3.19. Dependencies | 15 |
| 3.20. Encryption Wrapper | 16 |
| 3.21. XIP Address | 16 |
| 3.22. Load-time Metadata | 16 |
| 3.23. Run-time metadata | 17 |
| 3.24. Payload | 17 |

| | |
|--|----|
| 3.25. Manifest Envelope Element: Delegation Chain | 17 |
| 4. Security Considerations | 18 |
| 4.1. Threat Model | 18 |
| 4.2. Threat Descriptions | 19 |
| 4.2.1. THREAT.IMG.EXPIRED: Old Firmware | 19 |
| 4.2.2. THREAT.IMG.EXPIRED.OFFLINE : Offline device + Old Firmware | 19 |
| 4.2.3. THREAT.IMG.INCOMPATIBLE: Mismatched Firmware | 20 |
| 4.2.4. THREAT.IMG.FORMAT: The target device misinterprets the type of payload | 20 |
| 4.2.5. THREAT.IMG.LOCATION: The target device installs the payload to the wrong location | 21 |
| 4.2.6. THREAT.NET.REDIRECT: Redirection to inauthentic payload hosting | 21 |
| 4.2.7. THREAT.NET.ONPATH: Traffic interception | 21 |
| 4.2.8. THREAT.IMG.REPLACE: Payload Replacement | 21 |
| 4.2.9. THREAT.IMG.NON_AUTH: Unauthenticated Images | 22 |
| 4.2.10. THREAT.UPD.WRONG_PRECURSOR: Unexpected Precursor images | 22 |
| 4.2.11. THREAT.UPD.UNAPPROVED: Unapproved Firmware | 22 |
| 4.2.12. THREAT.IMG.DISCLOSURE: Reverse Engineering Of Firmware Image for Vulnerability Analysis | 24 |
| 4.2.13. THREAT.MFST.OVERRIDE: Overriding Critical Manifest Elements | 25 |
| 4.2.14. THREAT.MFST.EXPOSURE: Confidential Manifest Element Exposure | 25 |
| 4.2.15. THREAT.IMG.EXTRA: Extra data after image | 25 |
| 4.2.16. THREAT.KEY.EXPOSURE: Exposure of signing keys | 25 |
| 4.2.17. THREAT.MFST.MODIFICATION: Modification of manifest or payload prior to signing | 26 |
| 4.2.18. THREAT.MFST.TOCTOU: Modification of manifest between authentication and use | 26 |
| 4.3. Security Requirements | 26 |
| 4.3.1. REQ.SEC.SEQUENCE: Monotonic Sequence Numbers | 27 |
| 4.3.2. REQ.SEC.COMPATIBLE: Vendor, Device-type Identifiers | 27 |
| 4.3.3. REQ.SEC.EXP: Expiration Time | 27 |
| 4.3.4. REQ.SEC.AUTHENTIC: Cryptographic Authenticity | 28 |
| 4.3.5. REQ.SEC.AUTH.IMG_TYPE: Authenticated Payload Type | 28 |
| 4.3.6. Security Requirement REQ.SEC.AUTH.IMG_LOC: Authenticated Storage Location | 28 |
| 4.3.7. REQ.SEC.AUTH.REMOTE_LOC: Authenticated Remote Payload | 29 |
| 4.3.8. REQ.SEC.AUTH.EXEC: Secure Execution | 29 |
| 4.3.9. REQ.SEC.AUTH.PRECURSOR: Authenticated precursor images | 29 |
| 4.3.10. REQ.SEC.AUTH.COMPATIBILITY: Authenticated Vendor and Class IDs | 29 |
| 4.3.11. REQ.SEC.RIGHTS: Rights Require Authenticity | 29 |
| 4.3.12. REQ.SEC.IMG.CONFIDENTIALITY: Payload Encryption | 30 |

| | | |
|---------|--|----|
| 4.3.13. | REQ.SEC.ACCESS_CONTROL: Access Control | 30 |
| 4.3.14. | REQ.SEC.MFST.CONFIDENTIALITY: Encrypted Manifests . . . | 31 |
| 4.3.15. | REQ.SEC.IMG.COMPLETE_DIGEST: Whole Image Digest . . . | 31 |
| 4.3.16. | REQ.SEC.REPORTING: Secure Reporting | 31 |
| 4.3.17. | REQ.SEC.KEY.PROTECTION: Protected storage of signing keys | 31 |
| 4.3.18. | REQ.SEC.KEY.ROTATION: Protected storage of signing keys | 32 |
| 4.3.19. | REQ.SEC.MFST.CHECK: Validate manifests prior to deployment | 32 |
| 4.3.20. | REQ.SEC.MFST.TRUSTED: Construct manifests in a trusted environment | 32 |
| 4.3.21. | REQ.SEC.MFST.CONST: Manifest kept immutable between check and use | 33 |
| 4.4. | User Stories | 33 |
| 4.4.1. | USER_STORY.INSTALL.INSTRUCTIONS: Installation Instructions | 33 |
| 4.4.2. | USER_STORY.MFST.FAIL_EARLY: Fail Early | 34 |
| 4.4.3. | USER_STORY.OVERRIDE: Override Non-Critical Manifest Elements | 34 |
| 4.4.4. | USER_STORY.COMPONENT: Component Update | 34 |
| 4.4.5. | USER_STORY.MULTI_AUTH: Multiple Authorizations . . . | 35 |
| 4.4.6. | USER_STORY.IMG.FORMAT: Multiple Payload Formats . . . | 35 |
| 4.4.7. | USER_STORY.IMG.CONFIDENTIALITY: Prevent Confidential Information Disclosures | 35 |
| 4.4.8. | USER_STORY.IMG.UNKNOWN_FORMAT: Prevent Devices from Unpacking Unknown Formats | 35 |
| 4.4.9. | USER_STORY.IMG.CURRENT_VERSION: Specify Version Numbers of Target Firmware | 36 |
| 4.4.10. | USER_STORY.IMG.SELECT: Enable Devices to Choose Between Images | 36 |
| 4.4.11. | USER_STORY.EXEC.MFST: Secure Execution Using Manifests | 36 |
| 4.4.12. | USER_STORY.EXEC.DECOMPRESS: Decompress on Load . . . | 36 |
| 4.4.13. | USER_STORY.MFST.IMG: Payload in Manifest | 36 |
| 4.4.14. | USER_STORY.MFST.PARSE: Simple Parsing | 37 |
| 4.4.15. | USER_STORY.MFST.DELEGATION: Delegated Authority in Manifest | 37 |
| 4.4.16. | USER_STORY.MFST.PRE_CHECK: Update Evaluation | 37 |
| 4.4.17. | USER_STORY.MFST.ADMINISTRATION: Administration of manifests | 37 |
| 4.5. | Usability Requirements | 37 |
| 4.5.1. | REQ.USE.MFST.PRE_CHECK: Pre-Installation Checks . . . | 37 |
| 4.5.2. | REQ.USE.MFST.TEXT: Descriptive Manifest Information . | 38 |
| 4.5.3. | REQ.USE.MFST.OVERRIDE_REMOTE: Override Remote Resource Location | 38 |
| 4.5.4. | REQ.USE.MFST.COMPONENT: Component Updates | 38 |
| 4.5.5. | REQ.USE.MFST.MULTI_AUTH: Multiple authentications . . | 39 |

| | | |
|---------|--|----|
| 4.5.6. | REQ.USE.IMG.FORMAT: Format Usability | 39 |
| 4.5.7. | REQ.USE.IMG.NESTED: Nested Formats | 40 |
| 4.5.8. | REQ.USE.IMG.VERSIONS: Target Version Matching | 40 |
| 4.5.9. | REQ.USE.IMG.SELECT: Select Image by Destination . . . | 40 |
| 4.5.10. | REQ.USE.EXEC: Executable Manifest | 41 |
| 4.5.11. | REQ.USE.LOAD: Load-Time Information | 41 |
| 4.5.12. | REQ.USE.PAYLOAD: Payload in Manifest Envelope | 41 |
| 4.5.13. | REQ.USE.PARSE: Simple Parsing | 42 |
| 4.5.14. | REQ.USE.DELEGATION: Delegation of Authority in Manifest | 42 |
| 5. | IANA Considerations | 42 |
| 6. | Acknowledgements | 42 |
| 7. | References | 43 |
| 7.1. | Normative References | 43 |
| 7.2. | Informative References | 44 |
| | Authors' Addresses | 44 |

1. Introduction

Vulnerabilities with Internet of Things (IoT) devices have raised the need for a reliable and secure firmware update mechanism that is also suitable for constrained devices. Ensuring that devices function and remain secure over their service life requires such an update mechanism to fix vulnerabilities, to update configuration settings, as well as adding new functionality.

One component of such a firmware update is a concise and machine-processable meta-data document, or manifest, that describes the firmware image(s) and offers appropriate protection. This document describes the information that must be present in the manifest.

This document describes all the information elements required in a manifest to secure firmware updates of IoT devices. Each information element is motivated by user stories and threats it aims to mitigate. These threats and user stories are not intended to be an exhaustive list of the threats against IoT devices, nor of the possible user stories that describe how to conduct a firmware update. Instead they are intended to describe the threats against firmware updates in isolation and provide sufficient motivation to specify the information elements that cover a wide range of user stories.

To distinguish information elements from their encoding and serialization over the wire this document presents an information model. RFC 3444 [RFC3444] describes the differences between information and data models.

Because this document covers a wide range of user stories and a wide range of threats, not all information elements apply to all

scenarios. As a result, various information elements are optional to implement and optional to use, depending on which threats exist in a particular domain of application and which user stories are important for deployments.

2. Requirements and Terminology

2.1. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Unless otherwise stated these words apply to the design of the manifest format, not its implementation or application. Hence, whenever an information is declared as "REQUIRED" this implies that the manifest format document has to include support for it.

2.2. Terminology

This document uses terms defined in [I-D.ietf-suit-architecture]. The term 'Operator' refers to both Device and Network Operator.

Secure time and secure clock refer to a set of requirements on time sources. For local time sources, this primarily means that the clock must be monotonically increasing, including across power cycles, firmware updates, etc. For remote time sources, the provided time must be both authenticated and guaranteed to be correct to within some predetermined bounds, whenever the time source is accessible.

The term Envelope is used to describe an encoding that allows the bundling of a manifest with related information elements that are not directly contained within the manifest.

The term Payload is used to describe the data that is delivered to a device during an update. This is distinct from a "firmware image", as described in [I-D.ietf-suit-architecture], because the payload is often in an intermediate state, such as being encrypted, compressed and/or encoded as a differential update. The payload, taken in isolation, is often not the final firmware image.

3. Manifest Information Elements

Each manifest information element is anchored in a security requirement or a usability requirement. The manifest elements are described below, justified by their requirements.

3.1. Version ID of the Manifest Structure

An identifier that describes which iteration of the manifest format is contained in the structure. This allows devices to identify the version of the manifest data model that is in use.

This element is REQUIRED.

3.2. Monotonic Sequence Number

A monotonically increasing (unsigned) sequence number to prevent malicious actors from reverting a firmware update against the policies of the relevant authority. This number must not wrap around.

For convenience, the monotonic sequence number may be a UTC timestamp. This allows global synchronisation of sequence numbers without any additional management.

This element is REQUIRED.

Implements: REQ.SEC.SEQUENCE (Section 4.3.1)

3.3. Vendor ID

The Vendor ID element helps to distinguish between identically named products from different vendors. Vendor ID is not intended to be a human-readable element. It is intended for binary match/mismatch comparison only.

Recommended practice is to use [RFC4122] version 5 UUIDs with the vendor's domain name and the DNS name space ID. Other options include type 1 and type 4 UUIDs.

Fixed-size binary identifiers are preferred because they are simple to match, unambiguous in length, explicitly non-parsable, and require no issuing authority. Guaranteed unique integers are preferred because they are small and simple to match, however they may not be fixed length and they may require an issuing authority to ensure uniqueness. Free-form text is avoided because it is variable-length, prone to error, and often requires parsing outside the scope of the manifest serialization.

If human-readable content is required, it SHOULD be contained in a separate manifest information element: Manifest text information (Section 3.17)

This element is RECOMMENDED.

Implements: REQ.SEC.COMPATIBLE (Section 4.3.2),
REQ.SEC.AUTH.COMPATIBILITY (Section 4.3.10).

Here is an example for a domain name-based UUID. Vendor A creates a UUID based on a domain name it controls, such as `vendorId = UUID5(DNS, "vendor-a.example")`

Because the DNS infrastructure prevents multiple registrations of the same domain name, this UUID is (with very high probability) guaranteed to be unique. Because the domain name is known, this UUID is reproducible. Type 1 and type 4 UUIDs produce similar guarantees of uniqueness, but not reproducibility.

This approach creates a contention when a vendor changes its name or relinquishes control of a domain name. In this scenario, it is possible that another vendor would start using that same domain name. However, this UUID is not proof of identity; a device's trust in a vendor must be anchored in a cryptographic key, not a UUID.

3.4. Class ID

A device "Class" is a set of different device types that can accept the same firmware update without modification. It thereby allows devices to determine applicability of a firmware in an unambiguous way. Class IDs must be unique within the scope of a Vendor ID. This is to prevent similarly, or identically named devices colliding in their customer's infrastructure.

Recommended practice is to use [RFC4122] version 5 UUIDs with as much information as necessary to define firmware compatibility. Possible information used to derive the class UUID includes:

- o model name or number
- o hardware revision
- o runtime library version
- o bootloader version
- o ROM revision
- o silicon batch number

The Class ID UUID should use the Vendor ID as the name space identifier. Classes may be more fine-grained granular than is required to identify firmware compatibility. Classes must not be

less granular than is required to identify firmware compatibility. Devices may have multiple Class IDs.

Class ID is not intended to be a human-readable element. It is intended for binary match/mismatch comparison only. A manifest serialization SHOULD NOT permit free-form text content to be used for Class ID. A fixed-size binary identifier SHOULD be used.

Some organizations desire to keep the same product naming across multiple, incompatible hardware revisions for ease of user experience. If this naming is propagated into the firmware, then matching a specific hardware version becomes a challenge. An opaque, non-readable binary identifier has no naming implications and so is more likely to be usable for distinguishing among incompatible device groupings, regardless of naming.

Fixed-size binary identifiers are preferred because they are simple to match, unambiguous in length, opaque and free from naming implications, and explicitly non-parsable. Free-form text is avoided because it is variable-length, prone to error, often requires parsing outside the scope of the manifest serialization, and may be homogenized across incompatible device groupings.

If Class ID is not implemented, then each logical device class must use a unique trust anchor for authorization.

This element is RECOMMENDED.

Implements: Security Requirement REQ.SEC.COMPATIBLE (Section 4.3.2), REQ.SEC.AUTH.COMPATIBILITY (Section 4.3.10).

3.4.1. Example 1: Different Classes

Vendor A creates product Z and product Y. The firmware images of products Z and Y are not interchangeable. Vendor A creates UUIDs as follows:

- o vendorId = UUID5(DNS, "vendor-a.example")
- o ZclassId = UUID5(vendorId, "Product Z")
- o YclassId = UUID5(vendorId, "Product Y")

This ensures that Vendor A's Product Z cannot install firmware for Product Y and Product Y cannot install firmware for Product Z.

3.4.2. Example 2: Upgrading Class ID

Vendor A creates product X. Later, Vendor A adds a new feature to product X, creating product X v2. Product X requires a firmware update to work with firmware intended for product X v2.

Vendor A creates UUIDs as follows:

- o vendorId = UUID5(DNS, "vendor-a.example")
- o XclassId = UUID5(vendorId, "Product X")
- o Xv2classId = UUID5(vendorId, "Product X v2")

When product X receives the firmware update necessary to be compatible with product X v2, part of the firmware update changes the class ID to Xv2classId.

3.4.3. Example 3: Shared Functionality

Vendor A produces two products, product X and product Y. These components share a common core (such as an operating system), but have different applications. The common core and the applications can be updated independently. To enable X and Y to receive the same common core update, they require the same class ID. To ensure that only product X receives application X and only product Y receives application Y, product X and product Y must have different class IDs. The vendor creates Class IDs as follows:

- o vendorId = UUID5(DNS, "vendor-a.example")
- o XclassId = UUID5(vendorId, "Product X")
- o YclassId = UUID5(vendorId, "Product Y")
- o CommonClassId = UUID5(vendorId, "common core")

Product X matches against both XclassId and CommonClassId. Product Y matches against both YclassId and CommonClassId.

3.4.4. Example 4: White-labelling

Vendor A creates a product A and its firmware. Vendor B sells the product under its own name as Product B with some customised configuration. The vendors create the Class IDs as follows:

- o vendorIdA = UUID5(DNS, "vendor-a.example")

```
o classIdA = UUID5(vendorIdA, "Product A-Unlabelled")
o vendorIdB = UUID5(DNS, "vendor-b.example")
o classIdB = UUID5(vendorIdB, "Product B")
```

The product will match against each of these class IDs. If Vendor A and Vendor B provide different components for the device, the implementor may choose to make ID matching scoped to each component. Then, the vendorIdA, classIdA match the component ID supplied by Vendor A, and the vendorIdB, classIdB match the component ID supplied by Vendor B.

3.5. Precursor Image Digest Condition

This element provides information about the payload that needs to be present on the device for an update to apply. This may, for example, be the case with differential updates.

This element is OPTIONAL.

Implements: REQ.SEC.AUTH.PRECURSOR (Section 4.3.9)

3.6. Required Image Version List

Payloads may only be applied to a specific firmware version or firmware versions. For example, a payload containing a differential update may be applied only to a specific firmware version.

When a payload applies to multiple versions of a firmware, the required image version list specifies which firmware versions must be present for the update to be applied. This allows the update author to target specific versions of firmware for an update, while excluding those to which it should not or cannot be applied.

This element is OPTIONAL.

Implements: REQ.USE.IMG.VERSIONS (Section 4.5.8)

3.7. Expiration Time

This element tells a device the time at which the manifest expires and should no longer be used. This element should be used where a secure source of time is provided and firmware is intended to expire predictably. This element may also be displayed (e.g. via an app) for user confirmation since users typically have a reliable knowledge of the date.

Special consideration is required for end-of-life if a firmware will not be updated again, for example if a business stops issuing updates to a device. In this case the last valid firmware should not have an expiration time.

This element is OPTIONAL.

Implements: REQ.SEC.EXP (Section 4.3.3)

3.8. Payload Format

This element describes the payload format within the signed metadata. It is used to enable devices to decode payloads correctly.

This element is REQUIRED.

Implements: REQ.SEC.AUTH.IMG_TYPE (Section 4.3.5), REQ.USE.IMG.FORMAT (Section 4.5.6)

3.9. Processing Steps

A representation of the Processing Steps required to decode a payload, in particular those that are compressed, packed, or encrypted. The representation must describe which algorithms are used and must convey any additional parameters required by those algorithms.

A Processing Step may indicate the expected digest of the payload after the processing is complete.

This element is RECOMMENDED.

Implements: REQ.USE.IMG.NESTED (Section 4.5.7)

3.10. Storage Location

This element tells the device where to store a payload within a given component. The device can use this to establish which permissions are necessary and the physical storage location to use.

This element is REQUIRED.

Implements: REQ.SEC.AUTH.IMG_LOC (Section 4.3.6)

3.10.1. Example 1: Two Storage Locations

A device supports two components: an OS and an application. These components can be updated independently, expressing dependencies to ensure compatibility between the components. The Author chooses two storage identifiers:

- o "OS"
- o "APP"

3.10.2. Example 2: File System

A device supports a full-featured filesystem. The Author chooses to use the storage identifier as the path at which to install the payload. The payload may be a tarball, in which case, it unpacks the tarball into the specified path.

3.10.3. Example 3: Flash Memory

A device supports flash memory. The Author chooses to make the storage identifier the offset where the image should be written.

3.11. Component Identifier

In a device with more than one storage subsystem, a storage identifier is insufficient to identify where and how to store a payload. To resolve this, a component identifier indicates to which part of the storage subsystem the payload shall be placed.

A serialization may choose to combine Component Identifier and Storage Location (Section 3.10).

This element is OPTIONAL.

Implements: REQ.USE.MFST.COMPONENT (Section 4.5.4)

3.12. Payload Indicator

This element provides the information required for the device to acquire the payload. This functionality is only needed when the target device does not intrinsically know where to find the payload.

This can be encoded in several ways:

- o One URI
- o A list of URIs

- o A prioritised list of URIs

- o A list of signed URIs

This element is OPTIONAL.

Implements: REQ.SEC.AUTH.REMOTE_LOC (Section 4.3.7)

3.13. Payload Digests

This element contains one or more digests of one or more payloads. This allows the target device to ensure authenticity of the payload(s) when combined with the Signature (Section 3.15) element. A manifest format must provide a mechanism to select one payload from a list based on system parameters, such as Execute-In-Place Installation Address.

This element is REQUIRED. Support for more than one digest is OPTIONAL.

Implements: REQ.SEC.AUTHENTIC (Section 4.3.4), REQ.USE.IMG.SELECT (Section 4.5.9)

3.14. Size

The size of the payload in bytes, which informs the target device how big of a payload to expect. Without it, devices are exposed to some classes of denial of service attack.

This element is REQUIRED.

Implements: REQ.SEC.AUTH.EXEC (Section 4.3.8)

3.15. Manifest Envelope Element: Signature

The Signature element contains all the information necessary to protect the contents of the manifest against modification and to offer authentication of the signer. Because the Signature element authenticates the manifest, it cannot be contained within the manifest. Instead, the manifest is either contained within the signature element, or the signature element is a member of the Manifest Envelope and bundled with the manifest.

The Signature element represents the foundation of all security properties of the manifest. Manifests, which are included as dependencies by another manifests, should include a signature so that the recipient can distinguish between different actors with different permissions.

The Signature element must support multiple signers and multiple signing algorithms. A manifest format may allow multiple manifests to be covered by a single Signature element.

This element is REQUIRED in non-dependency manifests.

Implements: REQ.SEC.AUTHENTIC (Section 4.3.4), REQ.SEC.RIGHTS (Section 4.3.11), REQ.USE.MFST.MULTI_AUTH (Section 4.5.5)

3.16. Additional Installation Instructions

Additional installation instructions are machine-readable commands the device should execute when processing the manifest. This information is distinct from the information necessary to process a payload. Additional installation instructions include information such as update timing (for example, install only on Sunday, at 0200), procedural considerations (for example, shut down the equipment under control before executing the update), pre- and post-installation steps (for example, run a script). Other installation instructions could include requesting user confirmation before installing.

This element is OPTIONAL.

Implements: REQ.USE.MFST.PRE_CHECK (Section 4.5.1)

3.17. Manifest text information

Textual information pertaining to the update described by the manifest. This information is for human consumption only. It MUST NOT be the basis of any decision made by the recipient.

Implements: REQ.USE.MFST.TEXT (Section 4.5.2)

3.18. Aliases

A mechanism for a manifest to augment or replace URIs or URI lists defined by one or more of its dependencies.

This element is OPTIONAL.

Implements: REQ.USE.MFST.OVERRIDE_REMOTE (Section 4.5.3)

3.19. Dependencies

A list of other manifests that are required by the current manifest. Manifests are identified an unambiguous way, such as a cryptographic digest.

This element is REQUIRED to support deployments that include both multiple authorities and multiple payloads.

Implements: REQ.USE.MFST.COMPONENT (Section 4.5.4)

3.20. Encryption Wrapper

Encrypting firmware images requires symmetric content encryption keys. The encryption wrapper provides the information needed for a device to obtain or locate a key that it uses to decrypt the firmware.

This element is REQUIRED for encrypted payloads.

Implements: REQ.SEC.IMG.CONFIDENTIALITY (Section 4.3.12)

3.21. XIP Address

In order to support execute in place (XIP) systems with multiple possible base addresses, it is necessary to specify which address the payload is linked for.

For example a microcontroller may have a simple bootloader that chooses one of two images to boot. That microcontroller then needs to choose one of two firmware images to install, based on which of its two images is older.

This element is OPTIONAL.

Implements: REQ.USE.IMG.SELECT (Section 4.5.9)

3.22. Load-time Metadata

Load-time metadata provides the device with information that it needs in order to load one or more images. This metadata may include any of:

- o the source (e.g. non-volatile storage)
- o the destination (e.g. an address in RAM)
- o cryptographic information
- o decompression information
- o unpacking information

Typically, loading is done by copying an image from its permanent storage location into its active use location. The metadata allows operations such as decryption, decompression, and unpacking to be performed during that copy.

This element is OPTIONAL.

Implements: REQ.USE.LOAD (Section 4.5.11)

3.23. Run-time metadata

Run-time metadata provides the device with any extra information needed to boot the device. This may include the entry-point of an XIP image or the kernel command-line to boot a Linux image.

This element is OPTIONAL.

Implements: REQ.USE.EXEC (Section 4.5.10)

3.24. Payload

The Payload element is contained within the manifest or manifest envelope and enables the manifest and payload to be delivered simultaneously. This is used for delivering small payloads, such as cryptographic keys or configuration data.

This element is OPTIONAL.

Implements: REQ.USE.PAYLOAD (Section 4.5.12)

3.25. Manifest Envelope Element: Delegation Chain

The delegation chain offers enhanced authorization functionality via authorization tokens, such as CBOR Web Tokens [RFC8392] with Proof of Possession Key Semantics [RFC8747]. Each token itself is protected and does not require another layer of protection. Each authorization token typically includes a public key or a public key fingerprint, however this is dependent on the tokens used. Each token MAY include additional metadata, such as key usage information. Because the delegation chain is needed to verify the signature, it must be placed in the Manifest Envelope, rather than the Manifest.

The first token in any delegation chain MUST be authenticated by the recipient's Trust Anchor. Each subsequent token MUST be authenticated using the previous token. This allows a recipient to discard each antecedent token after it has authenticated the subsequent token. The final token MUST enable authentication of the manifest. More than one delegation chain MAY be used if more than

one signature is used. Note that no restriction is placed on the encoding order of these tokens, the order of elements is logical only.

This element is OPTIONAL.

Implements: REQ.USE.DELEGATION (Section 4.5.14), REQ.SEC.KEY.ROTATION (Section 4.3.18)

4. Security Considerations

The following sub-sections describe the threat model, user stories, security requirements, and usability requirements. This section also provides the motivations for each of the manifest information elements.

Note that it is worthwhile to recall that a firmware update is, by definition, remote code execution. Hence, if a device is configured to trust an entity to provide firmware, it trusts this entity to do the "right thing". Many classes of attacks can be mitigated by verifying that a firmware update came from a trusted party and that no rollback is taking place. However, if the trusted entity has been compromised and distributes attacker-provided firmware to devices then the possibilities for deference are limited.

4.1. Threat Model

The following sub-sections aim to provide information about the threats that were considered, the security requirements that are derived from those threats and the fields that permit implementation of the security requirements. This model uses the S.T.R.I.D.E. [STRIDE] approach. Each threat is classified according to:

- o Spoofing identity
- o Tampering with data
- o Repudiation
- o Information disclosure
- o Denial of service
- o Elevation of privilege

This threat model only covers elements related to the transport of firmware updates. It explicitly does not cover threats outside of

the transport of firmware updates. For example, threats to an IoT device due to physical access are out of scope.

4.2. Threat Descriptions

Many of the threats detailed in this section contain a "threat escalation" description. This explains how the described threat might fit together with other threats and produce a high severity threat. This is important because some of the described threats may seem low severity but could be used with others to construct a high severity compromise.

4.2.1. THREAT.IMG.EXPIRED: Old Firmware

Classification: Elevation of Privilege

An attacker sends an old, but valid manifest with an old, but valid firmware image to a device. If there is a known vulnerability in the provided firmware image, this may allow an attacker to exploit the vulnerability and gain control of the device.

Threat Escalation: If the attacker is able to exploit the known vulnerability, then this threat can be escalated to ALL TYPES.

Mitigated by: REQ.SEC.SEQUENCE (Section 4.3.1)

4.2.2. THREAT.IMG.EXPIRED.OFFLINE : Offline device + Old Firmware

Classification: Elevation of Privilege

An attacker targets a device that has been offline for a long time and runs an old firmware version. The attacker sends an old, but valid manifest to a device with an old, but valid firmware image. The attacker-provided firmware is newer than the installed one but older than the most recently available firmware. If there is a known vulnerability in the provided firmware image then this may allow an attacker to gain control of a device. Because the device has been offline for a long time, it is unaware of any new updates. As such it will treat the old manifest as the most current.

The exact mitigation for this threat depends on where the threat comes from. This requires careful consideration by the implementor. If the threat is from a network actor, including an on-path attacker, or an intruder into a management system, then a user confirmation can mitigate this attack, simply by displaying an expiration date and requesting confirmation. On the other hand, if the user is the attacker, then an online confirmation system (for example a trusted timestamp server) can be used as a mitigation system.

Threat Escalation: If the attacker is able to exploit the known vulnerability, then this threat can be escalated to ALL TYPES.

Mitigated by: REQ.SEC.EXP (Section 4.3.3), REQ.USE.MFST.PRE_CHECK (Section 4.5.1),

4.2.3. THREAT.IMG.INCOMPATIBLE: Mismatched Firmware

Classification: Denial of Service

An attacker sends a valid firmware image, for the wrong type of device, signed by an actor with firmware installation permission on both types of device. The firmware is verified by the device positively because it is signed by an actor with the appropriate permission. This could have wide-ranging consequences. For devices that are similar, it could cause minor breakage, or expose security vulnerabilities. For devices that are very different, it is likely to render devices inoperable.

Mitigated by: REQ.SEC.COMPATIBLE (Section 4.3.2)

For example, suppose that two vendors, Vendor A and Vendor B, adopt the same trade name in different geographic regions, and they both make products with the same names, or product name matching is not used. This causes firmware from Vendor A to match devices from Vendor B.

If the vendors are the firmware authorities, then devices from Vendor A will reject images signed by Vendor B since they use different credentials. However, if both devices trust the same Author, then, devices from Vendor A could install firmware intended for devices from Vendor B.

4.2.4. THREAT.IMG.FORMAT: The target device misinterprets the type of payload

Classification: Denial of Service

If a device misinterprets the format of the firmware image, it may cause a device to install a firmware image incorrectly. An incorrectly installed firmware image would likely cause the device to stop functioning.

Threat Escalation: An attacker that can cause a device to misinterpret the received firmware image may gain elevation of privilege and potentially expand this to all types of threat.

Mitigated by: REQ.SEC.AUTH.IMG_TYPE (Section 4.3.5)

4.2.5. THREAT.IMG.LOCATION: The target device installs the payload to the wrong location

Classification: Denial of Service

If a device installs a firmware image to the wrong location on the device, then it is likely to break. For example, a firmware image installed as an application could cause a device and/or an application to stop functioning.

Threat Escalation: An attacker that can cause a device to misinterpret the received code may gain elevation of privilege and potentially expand this to all types of threat.

Mitigated by: REQ.SEC.AUTH.IMG_LOC (Section 4.3.6)

4.2.6. THREAT.NET.REDIRECT: Redirection to inauthentic payload hosting

Classification: Denial of Service

If a device is tricked into fetching a payload for an attacker controlled site, the attacker may send corrupted payloads to devices.

Mitigated by: REQ.SEC.AUTH.REMOTE_LOC (Section 4.3.7)

4.2.7. THREAT.NET.ONPATH: Traffic interception

Classification: Spoofing Identity, Tampering with Data

An attacker intercepts all traffic to and from a device. The attacker can monitor or modify any data sent to or received from the device. This can take the form of: manifests, payloads, status reports, and capability reports being modified or not delivered to the intended recipient. It can also take the form of analysis of data sent to or from the device, either in content, size, or frequency.

Mitigated by: REQ.SEC.AUTHENTIC (Section 4.3.4), REQ.SEC.IMG.CONFIDENTIALITY (Section 4.3.12), REQ.SEC.AUTH.REMOTE_LOC (Section 4.3.7), REQ.SEC.MFST.CONFIDENTIALITY (Section 4.3.14), REQ.SEC.REPORTING (Section 4.3.16)

4.2.8. THREAT.IMG.REPLACE: Payload Replacement

Classification: Elevation of Privilege

An attacker replaces a newly downloaded firmware after a device finishes verifying a manifest. This could cause the device to

execute the attacker's code. This attack likely requires physical access to the device. However, it is possible that this attack is carried out in combination with another threat that allows remote execution. This is a typical Time Of Check/Time Of Use (TICTOC) attack.

Threat Escalation: If the attacker is able to exploit a known vulnerability, or if the attacker can supply their own firmware, then this threat can be escalated to ALL TYPES.

Mitigated by: REQ.SEC.AUTH.EXEC (Section 4.3.8)

4.2.9. THREAT.IMG.NON_AUTH: Unauthenticated Images

Classification: Elevation of Privilege / All Types

If an attacker can install their firmware on a device, for example by manipulating either payload or metadata, then they have complete control of the device.

Mitigated by: REQ.SEC.AUTHENTIC (Section 4.3.4)

4.2.10. THREAT.UPD.WRONG_PRECURSOR: Unexpected Precursor images

Classification: Denial of Service / All Types

Modifications of payloads and metadata allow an attacker to introduce a number of denial of service attacks. Below are some examples.

An attacker sends a valid, current manifest to a device that has an unexpected precursor image. If a payload format requires a precursor image (for example, delta updates) and that precursor image is not available on the target device, it could cause the update to break.

An attacker that can cause a device to install a payload against the wrong precursor image could gain elevation of privilege and potentially expand this to all types of threat. However, it is unlikely that a valid differential update applied to an incorrect precursor would result in a functional, but vulnerable firmware.

Mitigated by: REQ.SEC.AUTH.PRECURSOR (Section 4.3.9)

4.2.11. THREAT.UPD.UNAPPROVED: Unapproved Firmware

Classification: Denial of Service, Elevation of Privilege

This threat can appear in several ways, however it is ultimately about ensuring that devices retain the behaviour required by their

owner, or operator. The owner or operator of a device typically requires that the device maintain certain features, functions, capabilities, behaviours, or interoperability constraints (more generally, behaviour). If these requirements are broken, then a device will not fulfill its purpose. Therefore, if any party other than the device's owner or the owner's contracted Device Operator has the ability to modify device behaviour without approval, then this constitutes an elevation of privilege.

Similarly, a Network Operator may require that devices behave in a particular way in order to maintain the integrity of the network. If devices behaviour on a network can be modified without the approval of the Network Operator, then this constitutes an elevation of privilege with respect to the network.

For example, if the owner of a device has purchased that device because of features A, B, and C, and a firmware update is issued by the manufacturer, which removes feature A, then the device may not fulfill the owner's requirements any more. In certain circumstances, this can cause significantly greater threats. Suppose that feature A is used to implement a safety-critical system, whether the manufacturer intended this behaviour or not. When unapproved firmware is installed, the system may become unsafe.

In a second example, the owner or operator of a system of two or more interoperating devices needs to approve firmware for their system in order to ensure interoperability with other devices in the system. If the firmware is not qualified, the system as a whole may not work. Therefore, if a device installs firmware without the approval of the device owner or operator, this is a threat to devices or the system as a whole.

Similarly, the operator of a network may need to approve firmware for devices attached to the network in order to ensure favourable operating conditions within the network. If the firmware is not qualified, it may degrade the performance of the network. Therefore, if a device installs firmware without the approval of the Network Operator, this is a threat to the network itself.

Threat Escalation: If the firmware expects configuration that is present in devices deployed in Network A, but not in devices deployed in Network B, then the device may experience degraded security, leading to threats of All Types.

Mitigated by: REQ.SEC.RIGHTS (Section 4.3.11), REQ.SEC.ACCESS_CONTROL (Section 4.3.13)

4.2.11.1. Example 1: Multiple Network Operators with a Single Device Operator

In this example, assume that Device Operators expect the rights to create firmware but that Network Operators expect the rights to qualify firmware as fit-for-purpose on their networks. Additionally, assume that Device Operators manage devices that can be deployed on any network, including Network A and B in our example.

An attacker may obtain a manifest for a device on Network A. Then, this attacker sends that manifest to a device on Network B. Because Network A and Network B are under control of different Operators, and the firmware for a device on Network A has not been qualified to be deployed on Network B, the target device on Network B is now in violation of the Operator B's policy and may be disabled by this unqualified, but signed firmware.

This is a denial of service because it can render devices inoperable. This is an elevation of privilege because it allows the attacker to make installation decisions that should be made by the Operator.

4.2.11.2. Example 2: Single Network Operator with Multiple Device Operators

Multiple devices that interoperate are used on the same network and communicate with each other. Some devices are manufactured and managed by Device Operator A and other devices by Device Operator B. A new firmware is released by Device Operator A that breaks compatibility with devices from Device Operator B. An attacker sends the new firmware to the devices managed by Device Operator A without approval of the Network Operator. This breaks the behaviour of the larger system causing denial of service and possibly other threats. Where the network is a distributed SCADA system, this could cause misbehaviour of the process that is under control.

4.2.12. THREAT.IMG.DISCLOSURE: Reverse Engineering Of Firmware Image for Vulnerability Analysis

Classification: All Types

An attacker wants to mount an attack on an IoT device. To prepare the attack he or she retrieves the provided firmware image and performs reverse engineering of the firmware image to analyze it for specific vulnerabilities.

Mitigated by: REQ.SEC.IMG.CONFIDENTIALITY (Section 4.3.12)

4.2.13. THREAT.MFST.OVERRIDE: Overriding Critical Manifest Elements

Classification: Elevation of Privilege

An authorized actor, but not the Author, uses an override mechanism (USER_STORY.OVERRIDE (Section 4.4.3)) to change an information element in a manifest signed by the Author. For example, if the authorized actor overrides the digest and URI of the payload, the actor can replace the entire payload with a payload of their choice.

Threat Escalation: By overriding elements such as payload installation instructions or firmware digest, this threat can be escalated to all types.

Mitigated by: REQ.SEC.ACCESS_CONTROL (Section 4.3.13)

4.2.14. THREAT.MFST.EXPOSURE: Confidential Manifest Element Exposure

Classification: Information Disclosure

A third party may be able to extract sensitive information from the manifest.

Mitigated by: REQ.SEC.MFST.CONFIDENTIALITY (Section 4.3.14)

4.2.15. THREAT.IMG.EXTRA: Extra data after image

Classification: All Types

If a third party modifies the image so that it contains extra code after a valid, authentic image, that third party can then use their own code in order to make better use of an existing vulnerability.

Mitigated by: REQ.SEC.IMG.COMPLETE_DIGEST (Section 4.3.15)

4.2.16. THREAT.KEY.EXPOSURE: Exposure of signing keys

Classification: All Types

If a third party obtains a key or even indirect access to a key, for example in an hardware security module (HSM), then they can perform the same actions as the legitimate owner of the key. If the key is trusted for firmware update, then the third party can perform firmware updates as though they were the legitimate owner of the key.

For example, if manifest signing is performed on a server connected to the internet, an attacker may compromise the server and then be

able to sign manifests, even if the keys for manifest signing are held in an HSM that is accessed by the server.

Mitigated by: REQ.SEC.KEY.PROTECTION (Section 4.3.17),
REQ.SEC.KEY.ROTATION (Section 4.3.18)

4.2.17. THREAT.MFST.MODIFICATION: Modification of manifest or payload prior to signing

Classification: All Types

If an attacker can alter a manifest or payload before it is signed, they can perform all the same actions as the manifest author. This allows the attacker to deploy firmware updates to any devices that trust the manifest author. If an attacker can modify the code of a payload before the corresponding manifest is created, they can insert their own code. If an attacker can modify the manifest before it is signed, they can redirect the manifest to their own payload.

For example, the attacker deploys malware to the developer's computer or signing service that watches manifest creation activities and inserts code into any binary that is referenced by a manifest.

For example, the attacker deploys malware to the developer's computer or signing service that replaces the referenced binary (digest) and URI with the attacker's binary (digest) and URI.

Mitigated by: REQ.SEC.MFST.CHECK (Section 4.3.19),
REQ.SEC.MFST.TRUSTED (Section 4.3.20)

4.2.18. THREAT.MFST.TOCTOU: Modification of manifest between authentication and use

Classification: All Types

If an attacker can modify a manifest after it is authenticated (Time Of Check) but before it is used (Time Of Use), then the attacker can place any content whatsoever in the manifest.

Mitigated by: REQ.SEC.MFST.CONST (Section 4.3.21)

4.3. Security Requirements

The security requirements here are a set of policies that mitigate the threats described in Section 4.1.

4.3.1. REQ.SEC.SEQUENCE: Monotonic Sequence Numbers

Only an actor with firmware installation authority is permitted to decide when device firmware can be installed. To enforce this rule, manifests MUST contain monotonically increasing sequence numbers. Manifests may use UTC epoch timestamps to coordinate monotonically increasing sequence numbers across many actors in many locations. If UTC epoch timestamps are used, they must not be treated as times, they must be treated only as sequence numbers. Devices must reject manifests with sequence numbers smaller than any onboard sequence number, i.e. there is no sequence number roll over.

Note: This is not a firmware version field. It is a manifest sequence number. A firmware version may be rolled back by creating a new manifest for the old firmware version with a later sequence number.

Mitigates: THREAT.IMG.EXPIRED (Section 4.2.1)

Implemented by: Monotonic Sequence Number (Section 3.2)

4.3.2. REQ.SEC.COMPATIBLE: Vendor, Device-type Identifiers

Devices MUST only apply firmware that is intended for them. Devices must know that a given update applies to their vendor, model, hardware revision, and software revision. Human-readable identifiers are often error-prone in this regard, so unique identifiers should be used instead.

Mitigates: THREAT.IMG.INCOMPATIBLE (Section 4.2.3)

Implemented by: Vendor ID Condition (Section 3.3), Class ID Condition (Section 3.4)

4.3.3. REQ.SEC.EXP: Expiration Time

A firmware manifest MAY expire after a given time and devices may have a secure clock (local or remote). If a secure clock is provided and the Firmware manifest has an expiration timestamp, the device must reject the manifest if current time is later than the expiration time.

Special consideration is required for end-of-life in case device will not be updated again, for example if a business stops issuing updates for a device. The last valid firmware should not have an expiration time.

If a device has a flawed time source (either local or remote), an old update can be deployed as new.

Mitigates: THREAT.IMG.EXPIRED.OFFLINE (Section 4.2.2)

Implemented by: Expiration Time (Section 3.7)

4.3.4. REQ.SEC.AUTHENTIC: Cryptographic Authenticity

The authenticity of an update MUST be demonstrable. Typically, this means that updates must be digitally signed. Because the manifest contains information about how to install the update, the manifest's authenticity must also be demonstrable. To reduce the overhead required for validation, the manifest contains the cryptographic digest of the firmware image, rather than a second digital signature. The authenticity of the manifest can be verified with a digital signature or Message Authentication Code. The authenticity of the firmware image is tied to the manifest by the use of a cryptographic digest of the firmware image.

Mitigates: THREAT.IMG.NON_AUTH (Section 4.2.9), THREAT.NET.ONPATH (Section 4.2.7)

Implemented by: Signature (Section 3.15), Payload Digest (Section 3.13)

4.3.5. REQ.SEC.AUTH.IMG_TYPE: Authenticated Payload Type

The type of payload MUST be authenticated. For example, the target must know whether the payload is XIP firmware, a loadable module, or configuration data.

Mitigates: THREAT.IMG.FORMAT (Section 4.2.4)

Implemented by: Payload Format (Section 3.8), Signature (Section 3.15)

4.3.6. Security Requirement REQ.SEC.AUTH.IMG_LOC: Authenticated Storage Location

The location on the target where the payload is to be stored MUST be authenticated.

Mitigates: THREAT.IMG.LOCATION (Section 4.2.5)

Implemented by: Storage Location (Section 3.10)

4.3.7. REQ.SEC.AUTH.REMOTE_LOC: Authenticated Remote Payload

The location where a target should find a payload MUST be authenticated. Remote resources need to receive an equal amount of cryptographic protection as the manifest itself, when dereferencing URIs. The security considerations of Uniform Resource Identifiers (URIs) are applicable [RFC3986].

Mitigates: THREAT.NET.REDIRECT (Section 4.2.6), THREAT.NET.ONPATH (Section 4.2.7)

Implemented by: Payload Indicator (Section 3.12)

4.3.8. REQ.SEC.AUTH.EXEC: Secure Execution

The target SHOULD verify firmware at time of boot. This requires authenticated payload size, and digest.

Mitigates: THREAT.IMG.REPLACE (Section 4.2.8)

Implemented by: Payload Digest (Section 3.13), Size (Section 3.14)

4.3.9. REQ.SEC.AUTH.PRECURSOR: Authenticated precursor images

If an update uses a differential compression method, it MUST specify the digest of the precursor image and that digest MUST be authenticated.

Mitigates: THREAT.UPD.WRONG_PRECURSOR (Section 4.2.10)

Implemented by: Precursor Image Digest (Section 3.5)

4.3.10. REQ.SEC.AUTH.COMPATIBILITY: Authenticated Vendor and Class IDs

The identifiers that specify firmware compatibility MUST be authenticated to ensure that only compatible firmware is installed on a target device.

Mitigates: THREAT.IMG.INCOMPATIBLE (Section 4.2.3)

Implemented By: Vendor ID Condition (Section 3.3), Class ID Condition (Section 3.4)

4.3.11. REQ.SEC.RIGHTS: Rights Require Authenticity

If a device grants different rights to different actors, exercising those rights MUST be accompanied by proof of those rights, in the form of proof of authenticity. Authenticity mechanisms, such as

those required in REQ.SEC.AUTHENTIC (Section 4.3.4), can be used to prove authenticity.

For example, if a device has a policy that requires that firmware have both an Authorship right and a Qualification right and if that device grants Authorship and Qualification rights to different parties, such as a Device Operator and a Network Operator, respectively, then the firmware cannot be installed without proof of rights from both the Device Operator and the Network Operator.

Mitigates: THREAT.UPD.UNAPPROVED (Section 4.2.11)

Implemented by: Signature (Section 3.15)

4.3.12. REQ.SEC.IMG.CONFIDENTIALITY: Payload Encryption

The manifest information model MUST enable encrypted payloads. Encryption helps to prevent third parties, including attackers, from reading the content of the firmware image. This can protect against confidential information disclosures and discovery of vulnerabilities through reverse engineering. Therefore the manifest must convey the information required to allow an intended recipient to decrypt an encrypted payload.

Mitigates: THREAT.IMG.DISCLOSURE (Section 4.2.12), THREAT.NET.ONPATH (Section 4.2.7)

Implemented by: Encryption Wrapper (Section 3.20)

4.3.13. REQ.SEC.ACCESS_CONTROL: Access Control

If a device grants different rights to different actors, then an exercise of those rights MUST be validated against a list of rights for the actor. This typically takes the form of an Access Control List (ACL). ACLs are applied to two scenarios:

1. An ACL decides which elements of the manifest may be overridden and by which actors.
2. An ACL decides which component identifier/storage identifier pairs can be written by which actors.

Mitigates: THREAT.MFST.OVERRIDE (Section 4.2.13), THREAT.UPD.UNAPPROVED (Section 4.2.11)

Implemented by: Client-side code, not specified in manifest.

4.3.14. REQ.SEC.MFST.CONFIDENTIALITY: Encrypted Manifests

A manifest format MUST allow encryption of selected parts of the manifest or encryption of the entire manifest to prevent sensitive content of the firmware metadata to be leaked.

Mitigates: THREAT.MFST.EXPOSURE (Section 4.2.14), THREAT.NET.ONPATH (Section 4.2.7)

Implemented by: Manifest Encryption Wrapper / Transport Security

4.3.15. REQ.SEC.IMG.COMPLETE_DIGEST: Whole Image Digest

The digest SHOULD cover all available space in a fixed-size storage location. Variable-size storage locations MUST be restricted to exactly the size of deployed payload. This prevents any data from being distributed without being covered by the digest. For example, XIP microcontrollers typically have fixed-size storage. These devices should deploy a digest that covers the deployed firmware image, concatenated with the default erased value of any remaining space.

Mitigates: THREAT.IMG.EXTRA (Section 4.2.15)

Implemented by: Payload Digests (Section 3.13)

4.3.16. REQ.SEC.REPORTING: Secure Reporting

Status reports from the device to any remote system MUST be performed over an authenticated, confidential channel in order to prevent modification or spoofing of the reports.

Mitigates: THREAT.NET.ONPATH (Section 4.2.7)

Implemented by: Transport Security / Manifest format triggering generation of reports

4.3.17. REQ.SEC.KEY.PROTECTION: Protected storage of signing keys

Cryptographic keys for signing/authenticating manifests SHOULD be stored in a manner that is inaccessible to networked devices, for example in an HSM, or an air-gapped computer. This protects against an attacker obtaining the keys.

Keys SHOULD be stored in a way that limits the risk of a legitimate, but compromised, entity (such as a server or developer computer) issuing signing requests.

Mitigates: THREAT.KEY.EXPOSURE (Section 4.2.16)

Implemented by: Hardware-assisted isolation technologies, which are outside the scope of the manifest format.

4.3.18. REQ.SEC.KEY.ROTATION: Protected storage of signing keys

Cryptographic keys for signing/authenticating manifests SHOULD be replaced from time to time. Because it is difficult and risky to replace a Trust Anchor, keys used for signing updates SHOULD be delegates of that Trust Anchor.

If key expiration is performed based on time, then a secure clock is needed. If the time source used by a recipient to check for expiration is flawed, an old signing key can be used as current, which compounds THREAT.KEY.EXPOSURE (Section 4.2.16).

Mitigates: THREAT.KEY.EXPOSURE (Section 4.2.16)

Implemented by: Secure storage technology, which is a system design/implementation aspect outside the scope of the manifest format.

4.3.19. REQ.SEC.MFST.CHECK: Validate manifests prior to deployment

Manifests SHOULD be verified prior to deployment. This reduces problems that may arise with devices installing firmware images that damage devices unintentionally.

Mitigates: THREAT.MFST.MODIFICATION (Section 4.2.17)

Implemented by: Testing infrastructure. While outside the scope of the manifest format, proper testing of low-level software is essential for avoiding unnecessary down-time or worse situations.

4.3.20. REQ.SEC.MFST.TRUSTED: Construct manifests in a trusted environment

For high risk deployments, such as large numbers of devices or critical function devices, manifests SHOULD be constructed in an environment that is protected from interference, such as an air-gapped computer. Note that a networked computer connected to an HSM does not fulfill this requirement (see THREAT.MFST.MODIFICATION (Section 4.2.17)).

Mitigates: THREAT.MFST.MODIFICATION (Section 4.2.17)

Implemented by: Physical and network security for protecting the environment where firmware updates are prepared to avoid unauthorized access to this infrastructure.

4.3.21. REQ.SEC.MFST.CONST: Manifest kept immutable between check and use

Both the manifest and any data extracted from it MUST be held immutable between its authenticity verification (time of check) and its use (time of use). To make this guarantee, the manifest MUST fit within an internal memory or a secure memory, such as encrypted memory. The recipient SHOULD defend the manifest from tampering by code or hardware resident in the recipient, for example other processes or debuggers.

If an application requires that the manifest is verified before storing it, then this means the manifest MUST fit in RAM.

Mitigates: THREAT.MFST.TOCTOU (Section 4.2.18)

Implemented by: Proper system design with sufficient resources and implementation avoiding TOCTOU attacks.

4.4. User Stories

User stories provide expected use cases. These are used to feed into usability requirements.

4.4.1. USER_STORY.INSTALL.INSTRUCTIONS: Installation Instructions

As a Device Operator, I want to provide my devices with additional installation instructions so that I can keep process details out of my payload data.

Some installation instructions might be:

- o Use a table of hashes to ensure that each block of the payload is validated before writing.
- o Do not report progress.
- o Pre-cache the update, but do not install.
- o Install the pre-cached update matching this manifest.
- o Install this update immediately, overriding any long-running tasks.

Satisfied by: REQ.USE.MFST.PRE_CHECK (Section 4.5.1)

4.4.2. USER_STORY.MFST.FAIL_EARLY: Fail Early

As a designer of a resource-constrained IoT device, I want bad updates to fail as early as possible to preserve battery life and limit consumed bandwidth.

Satisfied by: REQ.USE.MFST.PRE_CHECK (Section 4.5.1)

4.4.3. USER_STORY.OVERRIDE: Override Non-Critical Manifest Elements

As a Device Operator, I would like to be able to override the non-critical information in the manifest so that I can control my devices more precisely. The authority to override this information is provided via the installation of a limited trust anchor by another authority.

Some examples of potentially overridable information:

- o URIs (Section 3.12): this allows the Device Operator to direct devices to their own infrastructure in order to reduce network load.
- o Conditions: this allows the Device Operator to pose additional constraints on the installation of the manifest.
- o Directives (Section 3.16): this allows the Device Operator to add more instructions such as time of installation.
- o Processing Steps (Section 3.9): If an intermediary performs an action on behalf of a device, it may need to override the processing steps. It is still possible for a device to verify the final content and the result of any processing step that specifies a digest. Some processing steps should be non-overridable.

Satisfied by: REQ.USE.MFST.COMPONENT (Section 4.5.4)

4.4.4. USER_STORY.COMPONENT: Component Update

As a Device Operator, I want to divide my firmware into components, so that I can reduce the size of updates, make different parties responsible for different components, and divide my firmware into frequently updated and infrequently updated components.

Satisfied by: REQ.USE.MFST.COMPONENT (Section 4.5.4)

4.4.5. USER_STORY.MULTI_AUTH: Multiple Authorizations

As a Device Operator, I want to ensure the quality of a firmware update before installing it, so that I can ensure interoperability of all devices in my product family. I want to restrict the ability to make changes to my devices to require my express approval.

Satisfied by: REQ.USE.MFST.MULTI_AUTH (Section 4.5.5),
REQ.SEC.ACCESS_CONTROL (Section 4.3.13)

4.4.6. USER_STORY.IMG.FORMAT: Multiple Payload Formats

As a Device Operator, I want to be able to send multiple payload formats to suit the needs of my update, so that I can optimise the bandwidth used by my devices.

Satisfied by: REQ.USE.IMG.FORMAT (Section 4.5.6)

4.4.7. USER_STORY.IMG.CONFIDENTIALITY: Prevent Confidential Information Disclosures

As a firmware author, I want to prevent confidential information in the manifest from being disclosed when distributing manifests and firmware images. Confidential information may include information about the device these updates are being applied to as well as information in the firmware image itself.

Satisfied by: REQ.SEC.IMG.CONFIDENTIALITY (Section 4.3.12)

4.4.8. USER_STORY.IMG.UNKNOWN_FORMAT: Prevent Devices from Unpacking Unknown Formats

As a Device Operator, I want devices to determine whether they can process a payload prior to downloading it.

In some cases, it may be desirable for a third party to perform some processing on behalf of a target. For this to occur, the third party MUST indicate what processing occurred and how to verify it against the Trust Provisioning Authority's intent.

This amounts to overriding Processing Steps (Section 3.9) and Payload Indicator (Section 3.12).

Satisfied by: REQ.USE.IMG.FORMAT (Section 4.5.6), REQ.USE.IMG.NESTED (Section 4.5.7), REQ.USE.MFST.OVERRIDE_REMOTE (Section 4.5.3)

4.4.9. USER_STORY.IMG.CURRENT_VERSION: Specify Version Numbers of Target Firmware

As a Device Operator, I want to be able to target devices for updates based on their current firmware version, so that I can control which versions are replaced with a single manifest.

Satisfied by: REQ.USE.IMG.VERSIONS (Section 4.5.8)

4.4.10. USER_STORY.IMG.SELECT: Enable Devices to Choose Between Images

As a developer, I want to be able to sign two or more versions of my firmware in a single manifest so that I can use a very simple bootloader that chooses between two or more images that are executed in-place.

Satisfied by: REQ.USE.IMG.SELECT (Section 4.5.9)

4.4.11. USER_STORY.EXEC.MFST: Secure Execution Using Manifests

As a signer for both secure execution/boot and firmware deployment, I would like to use the same signed document for both tasks so that my data size is smaller, I can share common code, and I can reduce signature verifications.

Satisfied by: REQ.USE.EXEC (Section 4.5.10)

4.4.12. USER_STORY.EXEC.DECOMPRESS: Decompress on Load

As a developer of firmware for a run-from-RAM device, I would like to use compressed images and to indicate to the bootloader that I am using a compressed image in the manifest so that it can be used with secure execution/boot.

Satisfied by: REQ.USE.LOAD (Section 4.5.11)

4.4.13. USER_STORY.MFST.IMG: Payload in Manifest

As an operator of devices on a constrained network, I would like the manifest to be able to include a small payload in the same packet so that I can reduce network traffic.

Small payloads may include, for example, wrapped content encryption keys, configuration information, public keys, authorization tokens, or X.509 certificates.

Satisfied by: REQ.USE.PAYLOAD (Section 4.5.12)

4.4.14. USER_STORY.MFST.PARSE: Simple Parsing

As a developer for constrained devices, I want a low complexity library for processing updates so that I can fit more application code on my device.

Satisfied by: REQ.USE.PARSE (Section 4.5.13)

4.4.15. USER_STORY.MFST.DELEGATION: Delegated Authority in Manifest

As a Device Operator that rotates delegated authority more often than delivering firmware updates, I would like to delegate a new authority when I deliver a firmware update so that I can accomplish both tasks in a single transmission.

Satisfied by: REQ.USE.DELEGATION (Section 4.5.14)

4.4.16. USER_STORY.MFST.PRE_CHECK: Update Evaluation

As an operator of a constrained network, I would like devices on my network to be able to evaluate the suitability of an update prior to initiating any large download so that I can prevent unnecessary consumption of bandwidth.

Satisfied by: REQ.USE.MFST.PRE_CHECK (Section 4.5.1)

4.4.17. USER_STORY.MFST.ADMINISTRATION: Administration of manifests

As a Device Operator, I want to understand what an update will do and to which devices it applies so that I can make informed choices about which updates to apply, when to apply them, and which devices should be updated.

Satisfied by REQ.USE.MFST.TEXT (Section 4.5.2)

4.5. Usability Requirements

The following usability requirements satisfy the user stories listed above.

4.5.1. REQ.USE.MFST.PRE_CHECK: Pre-Installation Checks

A manifest format MUST be able to carry all information required to process an update.

For example: Information about which precursor image is required for a differential update must be placed in the manifest.

Satisfies: [USER_STORY.MFST.PRE_CHECK(#user-story-mfst-pre-check),
USER_STORY.INSTALL.INSTRUCTIONS (Section 4.4.1)]

Implemented by: Additional installation instructions (Section 3.16)

4.5.2. REQ.USE.MFST.TEXT: Descriptive Manifest Information

It MUST be possible for a Device Operator to determine what a manifest will do and which devices will accept it prior to distribution.

Satisfies: USER_STORY.MFST.ADMINISTRATION (Section 4.4.17)

Implemented by: Manifest text information (Section 3.17)

4.5.3. REQ.USE.MFST.OVERRIDE_REMOTE: Override Remote Resource Location

A manifest format MUST be able to redirect payload fetches. This applies where two manifests are used in conjunction. For example, a Device Operator creates a manifest specifying a payload and signs it, and provides a URI for that payload. A Network Operator creates a second manifest, with a dependency on the first. They use this second manifest to override the URIs provided by the Device Operator, directing them into their own infrastructure instead. Some devices may provide this capability, while others may only look at canonical sources of firmware. For this to be possible, the device must fetch the payload, whereas a device that accepts payload pushes will ignore this feature.

Satisfies: USER_STORY.OVERRIDE (Section 4.4.3)

Implemented by: Aliases (Section 3.18)

4.5.4. REQ.USE.MFST.COMPONENT: Component Updates

A manifest format MUST be able to express the requirement to install one or more payloads from one or more authorities so that a multi-payload update can be described. This allows multiple parties with different permissions to collaborate in creating a single update for the IoT device, across multiple components.

This requirement implies that it must be possible to construct a tree of manifests on a multi-image target.

In order to enable devices with a heterogeneous storage architecture, the manifest must enable specification of both storage system and the storage location within that storage system.

Satisfies: USER_STORY.OVERRIDE (Section 4.4.3), USER_STORY.COMPONENT (Section 4.4.4)

Implemented by: Dependencies, StorageIdentifier, ComponentIdentifier

4.5.4.1. Example 1: Multiple Microcontrollers

An IoT device with multiple microcontrollers in the same physical device will likely require multiple payloads with different component identifiers.

4.5.4.2. Example 2: Code and Configuration

A firmware image can be divided into two payloads: code and configuration. These payloads may require authorizations from different actors in order to install (see REQ.SEC.RIGHTS (Section 4.3.11) and REQ.SEC.ACCESS_CONTROL (Section 4.3.13)). This structure means that multiple manifests may be required, with a dependency structure between them.

4.5.4.3. Example 3: Multiple Software Modules

A firmware image can be divided into multiple functional blocks for separate testing and distribution. This means that code would need to be distributed in multiple payloads. For example, this might be desirable in order to ensure that common code between devices is identical in order to reduce distribution bandwidth.

4.5.5. REQ.USE.MFST.MULTI_AUTH: Multiple authentications

A manifest format MUST be able to carry multiple signatures so that authorizations from multiple parties with different permissions can be required in order to authorize installation of a manifest.

Satisfies: USER_STORY.MULTI_AUTH (Section 4.4.5)

Implemented by: Signature (Section 3.15)

4.5.6. REQ.USE.IMG.FORMAT: Format Usability

The manifest format MUST accommodate any payload format that an Operator wishes to use. This enables the recipient to detect which format the Operator has chosen. Some examples of payload format are:

- o Binary
- o Executable and Linkable Format (ELF)

- o Differential
- o Compressed
- o Packed configuration
- o Intel HEX
- o Motorola S-Record

Satisfies: USER_STORY.IMG.FORMAT (Section 4.4.6)
USER_STORY.IMG.UNKNOWN_FORMAT (Section 4.4.8)

Implemented by: Payload Format (Section 3.8)

4.5.7. REQ.USE.IMG.NESTED: Nested Formats

The manifest format MUST accommodate nested formats, announcing to the target device all the nesting steps and any parameters used by those steps.

Satisfies: USER_STORY.IMG.CONFIDENTIALITY (Section 4.4.7)

Implemented by: Processing Steps (Section 3.9)

4.5.8. REQ.USE.IMG.VERSIONS: Target Version Matching

The manifest format MUST provide a method to specify multiple version numbers of firmware to which the manifest applies, either with a list or with range matching.

Satisfies: USER_STORY.IMG.CURRENT_VERSION (Section 4.4.9)

Implemented by: Required Image Version List (Section 3.6)

4.5.9. REQ.USE.IMG.SELECT: Select Image by Destination

The manifest format MUST provide a mechanism to list multiple equivalent payloads by Execute-In-Place Installation Address, including the payload digest and, optionally, payload URIs.

Satisfies: USER_STORY.IMG.SELECT (Section 4.4.10)

Implemented by: XIP Address (Section 3.21)

4.5.10. REQ.USE.EXEC: Executable Manifest

The manifest format MUST allow to describe an executable system with a manifest on both Execute-In-Place microcontrollers and on complex operating systems. In addition, the manifest format MUST be able to express metadata, such as a kernel command-line, used by any loader or bootloader.

Satisfies: USER_STORY.EXEC.MFST (Section 4.4.11)

Implemented by: Run-time metadata (Section 3.23)

4.5.11. REQ.USE.LOAD: Load-Time Information

The manifest format MUST enable carrying additional metadata for load time processing of a payload, such as cryptographic information, load-address, and compression algorithm. Note that load comes before execution/boot.

Satisfies: USER_STORY.EXEC.DECOMPRESS (Section 4.4.12)

Implemented by: Load-time metadata (Section 3.22)

4.5.12. REQ.USE.PAYLOAD: Payload in Manifest Envelope

The manifest format MUST allow placing a payload in the same structure as the manifest. This may place the payload in the same packet as the manifest.

Integrated payloads may include, for example, binaries as well as configuration information, and keying material.

When an integrated payload is provided, this increases the size of the manifest. Manifest size can cause several processing and storage concerns that require careful consideration. The payload can prevent the whole manifest from being contained in a single network packet, which can cause fragmentation and the loss of portions of the manifest in lossy networks. This causes the need for reassembly and retransmission logic. The manifest MUST be held immutable between verification and processing (see REQ.SEC.MFST.CONST (Section 4.3.21)), so a larger manifest will consume more memory with immutability guarantees, for example internal RAM or NVRAM, or external secure memory. If the manifest exceeds the available immutable memory, then it MUST be processed modularly, evaluating each of: delegation chains, the security container, and the actual manifest, which includes verifying the integrated payload. If the security model calls for downloading the manifest and validating it before storing to NVRAM in order to prevent wear to NVRAM and energy

expenditure in NVRAM, then either increasing memory allocated to manifest storage or modular processing of the received manifest may be required. While the manifest has been organised to enable this type of processing, it creates additional complexity in the parser. If the manifest is stored in NVRAM prior to processing, the integrated payload may cause the manifest to exceed the available storage. Because the manifest is received prior to validation of applicability, authority, or correctness, integrated payloads cause the recipient to expend network bandwidth and energy that may not be required if the manifest is discarded and these costs vary with the size of the integrated payload.

See also: REQ.SEC.MFST.CONST (Section 4.3.21).

Satisfies: USER_STORY.MFST.IMG (Section 4.4.13)

Implemented by: Payload (Section 3.24)

4.5.13. REQ.USE.PARSE: Simple Parsing

The structure of the manifest **MUST** be simple to parse to reduce the attack vectors against manifest parsers.

Satisfies: USER_STORY.MFST.PARSE (Section 4.4.14)

Implemented by: N/A

4.5.14. REQ.USE.DELEGATION: Delegation of Authority in Manifest

A manifest format **MUST** enable the delivery of delegation information. This information delivers a new key with which the recipient can verify the manifest.

Satisfies: USER_STORY.MFST.DELEGATION (Section 4.4.15)

Implemented by: Delegation Chain (Section 3.25)

5. IANA Considerations

This document does not require any actions by IANA.

6. Acknowledgements

We would like to thank our working group chairs, Dave Thaler, Russ Housley and David Waltermire, for their review comments and their support.

We would like to thank the participants of the 2018 Berlin SUIT Hackathon and the June 2018 virtual design team meetings for their discussion input.

In particular, we would like to thank Koen Zandberg, Emmanuel Baccelli, Carsten Bormann, David Brown, Markus Gueller, Frank Audun Kvamtro, Oyvind Ronningstad, Michael Richardson, Jan-Frederik Rieckers, Francisco Acosta, Anton Gerasimov, Matthias Waehlich, Max Groening, Daniel Petry, Gaetan Harter, Ralph Hamm, Steve Patrick, Fabio Utzig, Paul Lambert, Said Gharout, and Milen Stoychev.

We would like to thank those who contributed to the development of this information model. In particular, we would like to thank Milosch Meriac, Jean-Luc Giraud, Dan Ros, Amyas Philips, and Gary Thomson.

Finally, we would like to thank the following IESG members for their review feedback: Erik Kline, Murray Kucherawy, Barry Leiba, Alissa Cooper, Stephen Farrell and Benjamin Kaduk.

7. References

7.1. Normative References

- [I-D.ietf-suit-architecture]
Moran, B., Tschofenig, H., Brown, D., and M. Meriac, "A Firmware Update Architecture for Internet of Things", draft-ietf-suit-architecture-16 (work in progress), January 2021.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <<https://www.rfc-editor.org/info/rfc4122>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8392] Jones, M., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "CBOR Web Token (CWT)", RFC 8392, DOI 10.17487/RFC8392, May 2018, <<https://www.rfc-editor.org/info/rfc8392>>.

- [RFC8747] Jones, M., Seitz, L., Selander, G., Erdtman, S., and H. Tschofenig, "Proof-of-Possession Key Semantics for CBOR Web Tokens (CWTs)", RFC 8747, DOI 10.17487/RFC8747, March 2020, <<https://www.rfc-editor.org/info/rfc8747>>.

7.2. Informative References

- [RFC3444] Pras, A. and J. Schoenwaelder, "On the Difference between Information Models and Data Models", RFC 3444, DOI 10.17487/RFC3444, January 2003, <<https://www.rfc-editor.org/info/rfc3444>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.
- [STRIDE] Microsoft, "The STRIDE Threat Model", May 2018, <[https://msdn.microsoft.com/en-us/library/ee823878\(v=cs.20\).aspx](https://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx)>.

Authors' Addresses

Brendan Moran
Arm Limited

EMail: Brendan.Moran@arm.com

Hannes Tschofenig
Arm Limited

EMail: hannes.tschofenig@gmx.net

Henk Birkholz
Fraunhofer SIT

EMail: henk.birkholz@sit.fraunhofer.de

SUIT
Internet-Draft
Intended status: Standards Track
Expires: 30 October 2022

B. Moran
H. Tschofenig
Arm Limited
H. Birkholz
Fraunhofer SIT
K. Zandberg
Inria
28 April 2022

A Concise Binary Object Representation (CBOR)-based Serialization Format
for the Software Updates for Internet of Things (SUIT) Manifest
draft-ietf-suit-manifest-17

Abstract

This specification describes the format of a manifest. A manifest is a bundle of metadata about code/data obtained by a recipient (chiefly the firmware for an IoT device), where to find the that code/data, the devices to which it applies, and cryptographic information protecting the manifest. Software updates and Trusted Invocation both tend to use sequences of common operations, so the manifest encodes those sequences of operations, rather than declaring the metadata.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 30 October 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

| | |
|---|----|
| 1. Introduction | 4 |
| 2. Conventions and Terminology | 6 |
| 3. How to use this Document | 8 |
| 4. Background | 9 |
| 4.1. IoT Firmware Update Constraints | 9 |
| 4.2. SUIT Workflow Model | 10 |
| 5. Metadata Structure Overview | 11 |
| 5.1. Envelope | 12 |
| 5.2. Authentication Block | 13 |
| 5.3. Manifest | 13 |
| 5.3.1. Critical Metadata | 13 |
| 5.3.2. Common | 13 |
| 5.3.3. Command Sequences | 14 |
| 5.3.4. Integrity Check Values | 14 |
| 5.3.5. Human-Readable Text | 14 |
| 5.4. Severable Elements | 15 |
| 5.5. Integrated Payloads | 15 |
| 6. Manifest Processor Behavior | 15 |
| 6.1. Manifest Processor Setup | 16 |
| 6.2. Required Checks | 17 |
| 6.2.1. Minimizing Signature Verifications | 18 |
| 6.3. Interpreter Fundamental Properties | 18 |
| 6.4. Abstract Machine Description | 19 |
| 6.5. Special Cases of Component Index | 21 |
| 6.6. Serialized Processing Interpreter | 22 |
| 6.7. Parallel Processing Interpreter | 22 |
| 7. Creating Manifests | 23 |
| 7.1. Compatibility Check Template | 23 |
| 7.2. Trusted Invocation Template | 24 |
| 7.3. Component Download Template | 24 |
| 7.4. Install Template | 25 |
| 7.5. Integrated Payload Template | 25 |
| 7.6. Load from Nonvolatile Storage Template | 26 |
| 7.7. A/B Image Template | 26 |
| 8. Metadata Structure | 28 |
| 8.1. Encoding Considerations | 28 |
| 8.2. Envelope | 28 |

| | | |
|--------------------|--|----|
| 8.3. | Authenticated Manifests | 29 |
| 8.4. | Manifest | 29 |
| 8.4.1. | suit-manifest-version | 30 |
| 8.4.2. | suit-manifest-sequence-number | 30 |
| 8.4.3. | suit-reference-uri | 30 |
| 8.4.4. | suit-text | 31 |
| 8.4.5. | suit-common | 32 |
| 8.4.6. | SUIT_Command_Sequence | 33 |
| 8.4.7. | Reporting Policy | 35 |
| 8.4.8. | SUIT_Parameters | 36 |
| 8.4.9. | SUIT_Condition | 42 |
| 8.4.10. | SUIT_Directive | 45 |
| 8.4.11. | Integrity Check Values | 50 |
| 8.5. | Severable Elements | 50 |
| 9. | Access Control Lists | 50 |
| 10. | SUIT Digest Container | 51 |
| 11. | IANA Considerations | 51 |
| 11.1. | SUIT Commands | 52 |
| 11.2. | SUIT Parameters | 53 |
| 11.3. | SUIT Text Values | 54 |
| 11.4. | SUIT Component Text Values | 55 |
| 12. | Security Considerations | 55 |
| 13. | Acknowledgements | 55 |
| 14. | References | 56 |
| 14.1. | Normative References | 56 |
| 14.2. | Informative References | 57 |
| Appendix A. | A. Full CDDL | 58 |
| Appendix B. | B. Examples | 64 |
| B.1. | Example 0: Secure Boot | 65 |
| B.2. | Example 1: Simultaneous Download and Installation of Payload | 67 |
| B.3. | Example 2: Simultaneous Download, Installation, Secure Boot, Severed Fields | 69 |
| B.4. | Example 3: A/B images | 73 |
| B.5. | Example 4: Load from External Storage | 76 |
| B.6. | Example 5: Two Images | 79 |
| Appendix C. | C. Design Rational | 82 |
| C.1. | C.1 Design Rationale: Envelope | 83 |
| C.2. | C.2 Byte String Wrappers | 84 |
| Appendix D. | D. Implementation Conformance Matrix | 85 |
| Authors' Addresses | | 87 |

1. Introduction

A firmware update mechanism is an essential security feature for IoT devices to deal with vulnerabilities. While the transport of firmware images to the devices themselves is important there are already various techniques available. Equally important is the inclusion of metadata about the conveyed firmware image (in the form of a manifest) and the use of a security wrapper to provide end-to-end security protection to detect modifications and (optionally) to make reverse engineering more difficult. End-to-end security allows the author, who builds the firmware image, to be sure that no other party (including potential adversaries) can install firmware updates on IoT devices without adequate privileges. For confidentiality protected firmware images it is additionally required to encrypt the firmware image. Starting security protection at the author is a risk mitigation technique so firmware images and manifests can be stored on untrusted repositories; it also reduces the scope of a compromise of any repository or intermediate system to be no worse than a denial of service.

A manifest is a bundle of metadata describing one or more code or data payloads and how to:

- * Obtain any dependencies
- * Obtain the payload(s)
- * Install them
- * Verify them
- * Load them into memory
- * Invoke them

This specification defines the SUIT manifest format and it is intended to meet several goals:

- * Meet the requirements defined in [RFC9124].
- * Simple to parse on a constrained node
- * Simple to process on a constrained node
- * Compact encoding
- * Comprehensible by an intermediate system

- * Expressive enough to enable advanced use cases on advanced nodes
- * Extensible

The SUIT manifest can be used for a variety of purposes throughout its lifecycle, such as:

- * a Firmware Author to reason about releasing a firmware.
- * a Network Operator to reason about compatibility of a firmware.
- * a Device Operator to reason about the impact of a firmware.
- * the Device Operator to manage distribution of firmware to devices.
- * a Plant Manager to reason about timing and acceptance of firmware updates.
- * a device to reason about the authority & authenticity of a firmware prior to installation.
- * a device to reason about the applicability of a firmware.
- * a device to reason about the installation of a firmware.
- * a device to reason about the authenticity & encoding of a firmware at boot.

Each of these uses happens at a different stage of the manifest lifecycle, so each has different requirements.

It is assumed that the reader is familiar with the high-level firmware update architecture [RFC9019] and the threats, requirements, and user stories in [RFC9124].

The design of this specification is based on an observation that the vast majority of operations that a device can perform during an update or Trusted Invocation are composed of a small group of operations:

- * Copy some data from one place to another
- * Transform some data
- * Digest some data and compare to an expected value
- * Compare some system parameters to an expected value

* Run some code

In this document, these operations are called commands. Commands are classed as either conditions or directives. Conditions have no side-effects, while directives do have side-effects. Conceptually, a sequence of commands is like a script but the language is tailored to software updates and Trusted Invocation.

The available commands support simple steps, such as copying a firmware image from one place to another, checking that a firmware image is correct, verifying that the specified firmware is the correct firmware for the device, or unpacking a firmware. By using these steps in different orders and changing the parameters they use, a broad range of use cases can be supported. The SUIT manifest uses this observation to optimize metadata for consumption by constrained devices.

While the SUIT manifest is informed by and optimized for firmware update and Trusted Invocation use cases, there is nothing in the SUIT Information Model ([RFC9124]) that restricts its use to only those use cases. Other use cases include the management of trusted applications (TAs) in a Trusted Execution Environment (TEE), as discussed in [I-D.ietf-teep-architecture].

2. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Additionally, the following terminology is used throughout this document:

- * SUIT: Software Update for the Internet of Things, also the IETF working group for this standard.
- * Payload: A piece of information to be delivered. Typically Firmware for the purposes of SUIT.
- * Resource: A piece of information that is used to construct a payload.
- * Manifest: A manifest is a bundle of metadata about the firmware for an IoT device, where to find the firmware, and the devices to which it applies.

- * **Envelope:** A container with the manifest, an authentication wrapper with cryptographic information protecting the manifest, authorization information, and severable elements.
- * **Update:** One or more manifests that describe one or more payloads.
- * **Update Authority:** The owner of a cryptographic key used to sign updates, trusted by Recipients.
- * **Recipient:** The system, typically an IoT device, that receives and processes a manifest.
- * **Manifest Processor:** A component of the Recipient that consumes Manifests and executes the commands in the Manifest.
- * **Component:** An updatable logical block of the Firmware, Software, configuration, or data of the Recipient.
- * **Component Set:** A group of interdependent Components that must be updated simultaneously.
- * **Command:** A Condition or a Directive.
- * **Condition:** A test for a property of the Recipient or its Components.
- * **Directive:** An action for the Recipient to perform.
- * **Trusted Invocation:** A process by which a system ensures that only trusted code is executed, for example secure boot or launching a Trusted Application.
- * **A/B images:** Dividing a Recipient's storage into two or more bootable images, at different offsets, such that the active image can write to the inactive image(s).
- * **Record:** The result of a Command and any metadata about it.
- * **Report:** A list of Records.
- * **Procedure:** The process of invoking one or more sequences of commands.
- * **Update Procedure:** A procedure that updates a Recipient by fetching dependencies and images, and installing them.

- * **Invocation Procedure:** A procedure in which a Recipient verifies dependencies and images, loading images, and invokes one or more image.
- * **Software:** Instructions and data that allow a Recipient to perform a useful function.
- * **Firmware:** Software that is typically changed infrequently, stored in nonvolatile memory, and small enough to apply to [RFC7228] Class 0-2 devices.
- * **Image:** Information that a Recipient uses to perform its function, typically firmware/software, configuration, or resource data such as text or images. Also, a Payload, once installed is an Image.
- * **Slot:** One of several possible storage locations for a given Component, typically used in A/B image systems
- * **Abort:** An event in which the Manifest Processor immediately halts execution of the current Procedure. It creates a Record of an error condition.

3. How to use this Document

This specification covers five aspects of firmware update:

- * Section 4 describes the device constraints, use cases, and design principles that informed the structure of the manifest.
- * Section 5 gives a general overview of the metadata structure to inform the following sections
- * Section 6 describes what actions a Manifest processor should take.
- * Section 7 describes the process of creating a Manifest.
- * Section 8 specifies the content of the Envelope and the Manifest.

To implement an updatable device, see Section 6 and Section 8. To implement a tool that generates updates, see Section 7 and Section 8.

The IANA consideration section, see Section 11, provides instructions to IANA to create several registries. This section also provides the CBOR labels for the structures defined in this document.

The complete CDDL description is provided in Appendix A, examples are given in Appendix B and a design rational is offered in Appendix C. Finally, Appendix D gives a summarize of the mandatory-to-implement features of this specification.

This specification covers the core features of SUIF. Additional specifications describe functionality of advanced use cases, such as:

- * Firmware Encryption is covered in [I-D.ietf-suit-firmware-encryption]
- * Update Management is covered in [I-D.ietf-suit-update-management]
- * Features, such as dependencies, key delegation, multiple processors, required by the use of multiple trust domains are covered in [I-D.ietf-suit-trust-domains]
- * Secure reporting of the update status is covered in [I-D.ietf-suit-report]
- * Compression of firmware images

4. Background

Distributing software updates to diverse devices with diverse trust anchors in a coordinated system presents unique challenges. Devices have a broad set of constraints, requiring different metadata to make appropriate decisions. There may be many actors in production IoT systems, each of whom has some authority. Distributing firmware in such a multi-party environment presents additional challenges. Each party requires a different subset of data. Some data may not be accessible to all parties. Multiple signatures may be required from parties with different authorities. This topic is covered in more depth in [RFC9019]. The security aspects are described in [RFC9124].

4.1. IoT Firmware Update Constraints

The various constraints of IoT devices and the range of use cases that need to be supported create a broad set of requirements. For example, devices with:

- * limited processing power and storage may require a simple representation of metadata.
- * bandwidth constraints may require firmware compression or partial update support.

- * bootloader complexity constraints may require simple selection between two bootable images.
- * small internal storage may require external storage support.
- * multiple microcontrollers may require coordinated update of all applications.
- * large storage and complex functionality may require parallel update of many software components.
- * extra information may need to be conveyed in the manifest in the earlier stages of the device lifecycle before those data items are stripped when the manifest is delivered to a constrained device.

Supporting the requirements introduced by the constraints on IoT devices requires the flexibility to represent a diverse set of possible metadata, but also requires that the encoding is kept simple.

4.2. SUIIT Workflow Model

There are several fundamental assumptions that inform the model of Update Procedure workflow:

- * Compatibility must be checked before any other operation is performed.
- * In some applications, payloads must be fetched and validated prior to installation.

There are several fundamental assumptions that inform the model of the Invocation Procedure workflow:

- * Compatibility must be checked before any other operation is performed.
- * All payloads must be validated prior to loading.
- * All loaded images must be validated prior to execution.

Based on these assumptions, the manifest is structured to work with a pull parser, where each section of the manifest is used in sequence. The expected workflow for a Recipient installing an update can be broken down into five steps:

1. Verify the signature of the manifest.

2. Verify the applicability of the manifest.
3. Fetch payload(s).
4. Install payload(s).

When installation is complete, similar information can be used for validating and running images in a further three steps:

1. Verify image(s).
2. Load image(s).
3. Run image(s).

If verification and running is implemented in a bootloader, then the bootloader MUST also verify the signature of the manifest and the applicability of the manifest in order to implement secure boot workflows. The bootloader may add its own authentication, e.g. a Message Authentication Code (MAC), to the manifest in order to prevent further verifications.

5. Metadata Structure Overview

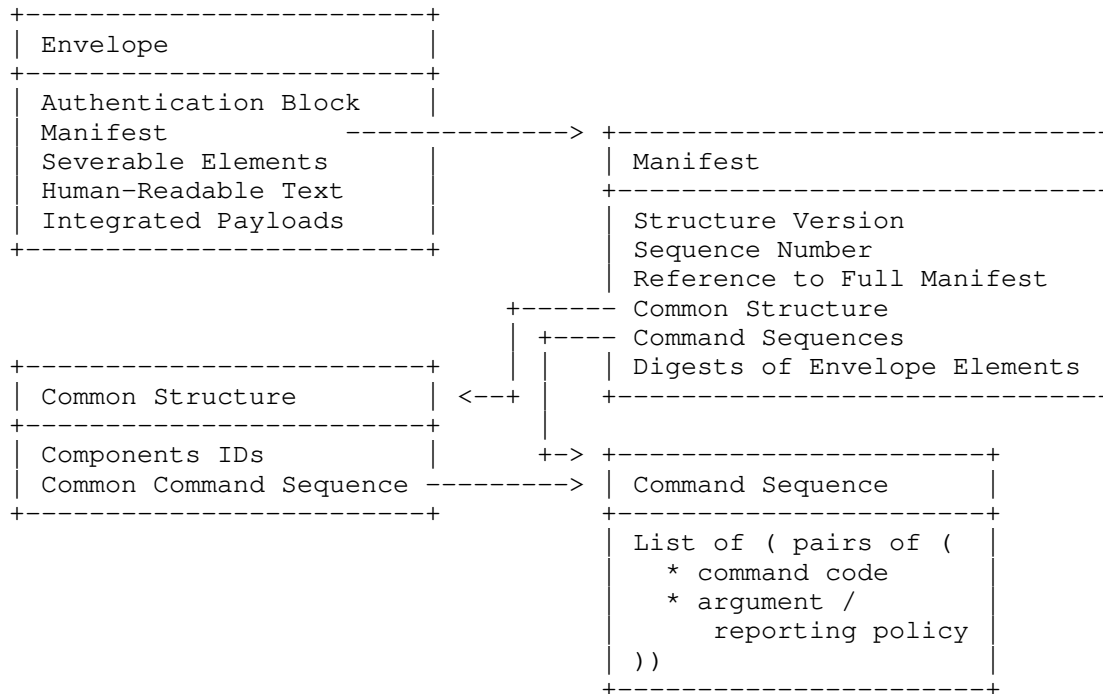
This section provides a high level overview of the manifest structure. The full description of the manifest structure is in Section 8.4

The manifest is structured from several key components:

1. The Envelope (see Section 5.1) contains the Authentication Block, the Manifest, any Severable Elements, and any Integrated Payloads.
2. The Authentication Block (see Section 5.2) contains a list of signatures or MACs of the manifest..
3. The Manifest (see Section 5.3) contains all critical, non-severable metadata that the Recipient requires. It is further broken down into:
 1. Critical metadata, such as sequence number.
 2. Common metadata, such as affected components.
 3. Command sequences, directing the Recipient how to install and use the payload(s).

4. Integrity check values for severable elements.
4. Severable elements (see Section 5.4).
5. Integrated payloads (see Section 5.5).

The diagram below illustrates the hierarchy of the Envelope.



5.1. Envelope

The SUIT Envelope is a container that encloses the Authentication Block, the Manifest, any Severable Elements, and any integrated payloads. The Envelope is used instead of conventional cryptographic envelopes, such as COSE_Envelope because it allows modular processing, severing of elements, and integrated payloads in a way that would add substantial complexity with existing solutions. See Appendix C.1 for a description of the reasoning for this.

See Section 8.2 for more detail.

5.2. Authentication Block

The Authentication Block contains a bstr-wrapped SUIT Digest Container, see Section 10, and one or more [RFC8152] CBOR Object Signing and Encryption (COSE) authentication blocks. These blocks are one of:

- * COSE_Sign_Tagged
- * COSE_Sign1_Tagged
- * COSE_Mac_Tagged
- * COSE_Mac0_Tagged

Each of these objects is used in detached payload mode. The payload is the bstr-wrapped SUIT_Digest.

See Section 8.3 for more detail.

5.3. Manifest

The Manifest contains most metadata about one or more images. The Manifest is divided into Critical Metadata, Common Metadata, Command Sequences, and Integrity Check Values.

See Section 8.4 for more detail.

5.3.1. Critical Metadata

Some metadata needs to be accessed before the manifest is processed. This metadata can be used to determine which manifest is newest and whether the structure version is supported. It also MAY provide a URI for obtaining a canonical copy of the manifest and Envelope.

See Section 8.4.1, Section 8.4.2, and Section 8.4.3 for more detail.

5.3.2. Common

Some metadata is used repeatedly and in more than one command sequence. In order to reduce the size of the manifest, this metadata is collected into the Common section. Common is composed of two parts: a list of components referenced by the manifest, and a command sequence to execute prior to each other command sequence. The common command sequence is typically used to set commonly used values and perform compatibility checks. The common command sequence MUST NOT have any side-effects outside of setting parameter values.

See Section 8.4.5 for more detail.

5.3.3. Command Sequences

Command sequences provide the instructions that a Recipient requires in order to install or use an image. These sequences tell a device to set parameter values, test system parameters, copy data from one place to another, transform data, digest data, and run code.

Command sequences are broken up into three groups: Common Command Sequence (see Section 5.3.2), update commands, and secure boot commands.

Update Command Sequences are: Payload Fetch, and Payload Installation. An Update Procedure is the complete set of each Update Command Sequence, each preceded by the Common Command Sequence.

Invocation Command Sequences are: System Validation, Image Loading, and Image Invocation. An Invocation Procedure is the complete set of each Invocation Command Sequence, each preceded by the Common Command Sequence.

Command Sequences are grouped into these sets to ensure that there is common coordination between dependencies and dependents on when to execute each command (dependencies are not defined in this specification).

See Section 8.4.6 for more detail.

5.3.4. Integrity Check Values

To enable Section 5.4, there needs to be a mechanism to verify integrity of any metadata outside the manifest. Integrity Check Values are used to verify the integrity of metadata that is not contained in the manifest. This MAY include Severable Command Sequences, or Text data. Integrated Payloads are integrity-checked using Command Sequences, so they do not have Integrity Check Values present in the Manifest.

See Section 8.4.11 for more detail.

5.3.5. Human-Readable Text

Text is typically a Severable Element (Section 5.4). It contains all the text that describes the update. Because text is explicitly for human consumption, it is all grouped together so that it can be Severed easily. The text section has space both for describing the manifest as a whole and for describing each individual component.

See Section 8.4.4 for more detail.

5.4. Severable Elements

Severable Elements are elements of the Envelope (Section 5.1) that have Integrity Check Values (Section 5.3.4) in the Manifest (Section 5.3).

Because of this organisation, these elements can be discarded or "Severed" from the Envelope without changing the signature of the Manifest. This allows savings based on the size of the Envelope in several scenarios, for example:

- * A management system severs the Text sections before sending an Envelope to a constrained Recipient, which saves Recipient bandwidth.
- * A Recipient severs the Installation section after installing the Update, which saves storage space.

See Section 8.5 for more detail.

5.5. Integrated Payloads

In some cases, it is beneficial to include a payload in the Envelope of a manifest. For example:

- * When an update is delivered via a comparatively unconstrained medium, such as a removable mass storage device, it may be beneficial to bundle updates into single files.
- * When a manifest transports a small payload, such as an encrypted key, that payload may be placed in the manifest's envelope.

See Section 7.5 for more detail.

6. Manifest Processor Behavior

This section describes the behavior of the manifest processor and focuses primarily on interpreting commands in the manifest. However, there are several other important behaviors of the manifest processor: encoding version detection, rollback protection, and authenticity verification are chief among these.

6.1. Manifest Processor Setup

Prior to executing any command sequence, the manifest processor or its host application **MUST** inspect the manifest version field and fail when it encounters an unsupported encoding version. Next, the manifest processor or its host application **MUST** extract the manifest sequence number and perform a rollback check using this sequence number. The exact logic of rollback protection may vary by application, but it has the following properties:

- * Whenever the manifest processor can choose between several manifests, it **MUST** select the latest valid, authentic manifest.
- * If the latest valid, authentic manifest fails, it **MAY** select the next latest valid, authentic manifest, according to application-specific policy.

Here, valid means that a manifest has a supported encoding version and it has not been excluded for other reasons. Reasons for excluding typically involve first executing the manifest and may include:

- * Test failed (e.g. Vendor ID/Class ID).
- * Unsupported command encountered.
- * Unsupported parameter encountered.
- * Unsupported Component Identifier encountered.
- * Payload not available.
- * Application crashed when executed.
- * Watchdog timeout occurred.
- * Payload verification failed.
- * Missing required component from a Component Set.
- * Required parameter not supplied.

These failure reasons **MAY** be combined with retry mechanisms prior to marking a manifest as invalid.

Selecting an older manifest in the event of failure of the latest valid manifest is a robustness mechanism that is necessary for supporting the requirements in [RFC9019], section 3.5. It may not be

appropriate for all applications. In particular Trusted Execution Environments MAY require a failure to invoke a new installation, rather than a rollback approach. See [RFC9124], Section 4.2.1 for more discussion on the security considerations that apply to rollback.

Following these initial tests, the manifest processor clears all parameter storage. This ensures that the manifest processor begins without any leaked data.

6.2. Required Checks

The RECOMMENDED process is to verify the signature of the manifest prior to parsing/executing any section of the manifest. This guards the parser against arbitrary input by unauthenticated third parties, but it costs extra energy when a Recipient receives an incompatible manifest.

When validating authenticity of manifests, the manifest processor MAY use an ACL (see Section 9) to determine the extent of the rights conferred by that authenticity.

Once a valid, authentic manifest has been selected, the manifest processor MUST examine the component list and verify that its maximum number of components is not exceeded and that each listed component is supported.

For each listed component, the manifest processor MUST provide storage for the supported parameters. If the manifest processor does not have sufficient temporary storage to process the parameters for all components, it MAY process components serially for each command sequence. See Section 6.6 for more details.

The manifest processor SHOULD check that the common sequence contains at least Check Vendor Identifier command and at least one Check Class Identifier command.

Because the common sequence contains Check Vendor Identifier and Check Class Identifier command(s), no custom commands are permitted in the common sequence. This ensures that any custom commands are only executed by devices that understand them.

If the manifest contains more than one component, each command sequence MUST begin with a Set Component Index.

If a Recipient supports groups of interdependent components (a Component Set), then it SHOULD verify that all Components in the Component Set are specified by one update, that is the manifest:

1. has sufficient permissions imparted by its signatures
2. specifies a digest and a payload for every Component in the Component Set.

6.2.1. Minimizing Signature Verifications

Signature verification can be energy and time expensive on a constrained device. MAC verification is typically unaffected by these concerns. A Recipient MAY choose to parse and execute only the SUIF_Common section of the manifest prior to signature verification, if all of the below apply:

- * The Authentication Block contains a COSE_Sign_Tagged or COSE_Sign1_Tagged
- * The Recipient receives manifests over an unauthenticated channel, exposing it to more inauthentic or incompatible manifests, and
- * The Recipient has a power budget that makes signature verification undesirable

When executing Common prior to authenticity validation, the Manifest Processor MUST first evaluate the integrity of the manifest using the SUIF_Digest present in the authentication block.

The guidelines in Creating Manifests (Section 7) require that the common section contains the applicability checks, so this section is sufficient for applicability verification. The parser MUST restrict acceptable commands to conditions and the following directives: Override Parameters, Set Parameters, Try Each, and Run Sequence ONLY. The manifest parser MUST NOT execute any command with side-effects outside the parser (for example, Run, Copy, Swap, or Fetch commands) prior to authentication and any such command MUST Abort. The Common Sequence MUST be executed again, in its entirety, after authenticity validation.

A Recipient MAY rely on network infrastructure to filter inapplicable manifests.

6.3. Interpreter Fundamental Properties

The interpreter has a small set of design goals:

1. Executing an update MUST either result in an error, or a verifiably correct system state.

2. Executing a Trusted Invocation MUST either result in an error, or an invoked image.
3. Executing the same manifest on multiple Recipients MUST result in the same system state.

NOTE: when using A/B images, the manifest functions as two (or more) logical manifests, each of which applies to a system in a particular starting state. With that provision, design goal 3 holds.

6.4. Abstract Machine Description

The heart of the manifest is the list of commands, which are processed by a Manifest Processor--a form of interpreter. This Manifest Processor can be modeled as a simple abstract machine. This machine consists of several data storage locations that are modified by commands.

There are two types of commands, namely those that modify state (directives) and those that perform tests (conditions). Parameters are used as the inputs to commands. Some directives offer control flow operations. Directives target a specific component. A component is a unit of code or data that can be targeted by an update. Components are identified by Component Identifiers, but referenced in commands by Component Index; Component Identifiers are arrays of binary strings and a Component Index is an index into the array of Component Identifiers.

Conditions MUST NOT have any side-effects other than informing the interpreter of success or failure. The Interpreter does not Abort if the Soft Failure flag (Section 8.4.8.14) is set when a Condition reports failure.

Directives MAY have side-effects in the parameter table, the interpreter state, or the current component. The Interpreter MUST Abort if a Directive reports failure regardless of the Soft Failure flag.

To simplify the logic describing the command semantics, the object "current" is used. It represents the component identified by the Component Index:

```
current := components\[component-index\]
```

As a result, Set Component Index is described as `current := components[arg]`.

The following table describes the behavior of each command. "params" represents the parameters for the current component. Most commands operate on a component.

| Command Name | Semantic of the Operation |
|-------------------------|--|
| Check Vendor Identifier | <code>assert(binary-match(current, current.params[vendor-id]))</code> |
| Check Class Identifier | <code>assert(binary-match(current, current.params[class-id]))</code> |
| Verify Image | <code>assert(binary-match(digest(current), current.params[digest]))</code> |
| Set Component Index | <code>current := components[arg]</code> |
| Override Parameters | <code>current.params[k] := v for-each k,v in arg</code> |
| Set Parameters | <code>current.params[k] := v if not k in params for-each k,v in arg</code> |
| Run | <code>run(current)</code> |
| Fetch | <code>store(current, fetch(current.params[uri]))</code> |
| Use Before | <code>assert(now() < arg)</code> |
| Check Component Slot | <code>assert(current.slot-index == arg)</code> |
| Check Device Identifier | <code>assert(binary-match(current, current.params[device-id]))</code> |
| Abort | <code>assert(0)</code> |
| Try Each | <code>try-each-done if exec(seq) is not error for-each seq in arg</code> |
| Copy | <code>store(current, current.params[src-component])</code> |
| Swap | <code>swap(current, current.params[src-component])</code> |
| Run Sequence | <code>exec(arg)</code> |

| | |
|--------------------|-------------------|
| Run with Arguments | run(current, arg) |
|--------------------|-------------------|

Table 1

6.5. Special Cases of Component Index

Component Index can take on one of three types:

1. Integer
2. Array of integers
3. True

Integers MUST always be supported by Set Component Index. Arrays of integers MUST be supported by Set Component Index if the Recipient supports 3 or more components. True MUST be supported by Set Component Index if the Recipient supports 2 or more components. Each of these operates on the list of components declared in the manifest.

Integer indices are the default case as described in the previous section. An array of integers represents a list of the components (Set Component Index) to which each subsequent command applies. The value True replaces the list of component indices with the full list of components, as defined in the manifest.

When a command is executed, it either 1. operates on the component identified by the component index if that index is an integer, or 2. it operates on each component identified by an array of indices, or 3. it operates on every component if the index is the boolean True. This is described by the following pseudocode:

```

if component-index is true:
    current-list = components
else if component-index is array:
    current-list = [ components[idx] for idx in component-index ]
else:
    current-list = [ components[component-index] ]
for current in current-list:
    cmd(current)

```

Try Each and Run Sequence are affected in the same way as other commands: they are invoked once for each possible Component. This means that the sequences that are arguments to Try Each and Run Sequence are NOT invoked with Component Index = True, nor are they invoked with array indices. They are only invoked with integer indices. The interpreter loops over the whole sequence, setting the Component Index to each index in turn.

6.6. Serialized Processing Interpreter

In highly constrained devices, where storage for parameters is limited, the manifest processor MAY handle one component at a time, traversing the manifest tree once for each listed component. In this mode, the interpreter ignores any commands executed while the component index is not the current component. This reduces the overall volatile storage required to process the update so that the only limit on number of components is the size of the manifest. However, this approach requires additional processing power.

In order to operate in this mode, the manifest processor loops on each section for every supported component, simply ignoring commands when the current component is not selected.

When a serialized Manifest Processor encounters a component index of True, it does not ignore any commands. It applies them to the current component on each iteration.

6.7. Parallel Processing Interpreter

Advanced Recipients MAY make use of the Strict Order parameter and enable parallel processing of some Command Sequences, or it may reorder some Command Sequences. To perform parallel processing, once the Strict Order parameter is set to False, the Recipient may issue each or every command concurrently until the Strict Order parameter is returned to True or the Command Sequence ends. Then, it waits for all issued commands to complete before continuing processing of commands. To perform out-of-order processing, a similar approach is used, except the Recipient consumes all commands after the Strict Order parameter is set to False, then it sorts these commands into its preferred order, invokes them all, then continues processing.

When the manifest processor encounters any of these scenarios the parallel processing MUST halt until all issued commands have completed:

- * Set Parameters.
- * Override Parameters.

- * Set Strict Order = True.

- * Set Component Index.

To perform more useful parallel operations, a manifest author may collect sequences of commands in a Run Sequence command. Then, each of these sequences MAY be run in parallel. Each sequence defaults to Strict Order = True. To isolate each sequence from each other sequence, each sequence MUST begin with a Set Component Index directive with the following exception: when the index is either True or an array of indices, the Set Component Index is implied. Any further Set Component Index directives MUST cause an Abort. This allows the interpreter that issues Run Sequence commands to check that the first element is correct, then issue the sequence to a parallel execution context to handle the remainder of the sequence.

7. Creating Manifests

Manifests are created using tools for constructing COSE structures, calculating cryptographic values and compiling desired system state into a sequence of operations required to achieve that state. The process of constructing COSE structures and the calculation of cryptographic values is covered in [RFC8152].

Compiling desired system state into a sequence of operations can be accomplished in many ways. Several templates are provided below to cover common use-cases. These templates can be combined to produce more complex behavior.

The author MUST ensure that all parameters consumed by a command are set prior to invoking that command. Where Component Index = True, this means that the parameters consumed by each command MUST have been set for each Component.

This section details a set of templates for creating manifests. These templates explain which parameters, commands, and orders of commands are necessary to achieve a stated goal.

NOTE: On systems that support only a single component, Set Component Index has no effect and can be omitted.

NOTE: *A digest MUST always be set using Override Parameters.*

7.1. Compatibility Check Template

The goal of the compatibility check template ensure that Recipients only install compatible images.

In this template all information is contained in the common sequence and the following sequence of commands is used:

- * Set Component Index directive (see Section 8.4.10.1)
- * Override Parameters directive (see Section 8.4.10.3) for Vendor ID and Class ID (see Section 8.4.8)
- * Check Vendor Identifier condition (see Section 8.4.8.2)
- * Check Class Identifier condition (see Section 8.4.8.2)

7.2. Trusted Invocation Template

The goal of the Trusted Invocation template is to ensure that only authorized code is invoked; such as in Secure Boot or when a Trusted Application is loaded into a TEE.

The following commands are placed into the common sequence:

- * Set Component Index directive (see Section 8.4.10.1)
- * Override Parameters directive (see Section 8.4.10.3) for Image Digest and Image Size (see Section 8.4.8)

The system validation sequence contains the following commands:

- * Set Component Index directive (see Section 8.4.10.1)
- * Check Image Match condition (see Section 8.4.9.2)

Then, the run sequence contains the following commands:

- * Set Component Index directive (see Section 8.4.10.1)
- * Run directive (see Section 8.4.10.7)

7.3. Component Download Template

The goal of the Component Download template is to acquire and store an image.

The following commands are placed into the common sequence:

- * Set Component Index directive (see Section 8.4.10.1)
- * Override Parameters directive (see Section 8.4.10.3) for Image Digest and Image Size (see Section 8.4.8)

Then, the install sequence contains the following commands:

- * Set Component Index directive (see Section 8.4.10.1)
- * Override Parameters directive (see Section 8.4.10.3) for URI (see Section 8.4.8.9)
- * Fetch directive (see Section 8.4.10.4)
- * Check Image Match condition (see Section 8.4.9.2)

The Fetch directive needs the URI parameter to be set to determine where the image is retrieved from. Additionally, the destination of where the component shall be stored has to be configured. The URI is configured via the Set Parameters directive while the destination is configured via the Set Component Index directive.

7.4. Install Template

The goal of the Install template is to use an image already stored in an identified component to copy into a second component.

This template is typically used with the Component Download template, however a modification to that template is required: the Component Download operations are moved from the Payload Install sequence to the Payload Fetch sequence.

Then, the install sequence contains the following commands:

- * Set Component Index directive (see Section 8.4.10.1)
- * Override Parameters directive (see Section 8.4.10.3) for Source Component (see Section 8.4.8.10)
- * Copy directive (see Section 8.4.10.5)
- * Check Image Match condition (see Section 8.4.9.2)

7.5. Integrated Payload Template

The goal of the Integrated Payload template is to install a payload that is included in the manifest envelope. It is identical to the Component Download template (Section 7.3).

An implementer MAY choose to place a payload in the envelope of a manifest. The payload envelope key MUST be a string. The payload MUST be serialized in a bstr element.

The URI for a payload enclosed in this way MAY be expressed as a fragment-only reference, as defined in [RFC3986], Section 4.4.

A distributor MAY choose to pre-fetch a payload and add it to the manifest envelope, using the URI as the key.

7.6. Load from Nonvolatile Storage Template

The goal of the Load from Nonvolatile Storage template is to load an image from a non-volatile component into a volatile component, for example loading a firmware image from external Flash into RAM.

The following commands are placed into the load sequence:

- * Set Component Index directive (see Section 8.4.10.1)
- * Override Parameters directive (see Section 8.4.10.3) for Source Component (see Section 8.4.8)
- * Copy directive (see Section 8.4.10.5)

As outlined in Section 6.4, the Copy directive needs a source and a destination to be configured. The source is configured via Component Index (with the Set Parameters directive) and the destination is configured via the Set Component Index directive.

7.7. A/B Image Template

The goal of the A/B Image Template is to acquire, validate, and invoke one of two images, based on a test.

The following commands are placed in the common block:

- * Set Component Index directive (see Section 8.4.10.1)
- * Try Each
 - First Sequence:
 - o Override Parameters directive (see Section 8.4.10.3, Section 8.4.8) for Slot A
 - o Check Slot Condition (see Section 8.4.9.3)
 - o Override Parameters directive (see Section 8.4.10.3) for Image Digest A and Image Size A (see Section 8.4.8)
 - Second Sequence:

- o Override Parameters directive (see Section 8.4.10.3, Section 8.4.8) for Slot B
- o Check Slot Condition (see Section 8.4.9.3)
- o Override Parameters directive (see Section 8.4.10.3) for Image Digest B and Image Size B (see Section 8.4.8)

The following commands are placed in the fetch block or install block

- * Set Component Index directive (see Section 8.4.10.1)
- * Try Each
 - First Sequence:
 - o Override Parameters directive (see Section 8.4.10.3, Section 8.4.8) for Slot A
 - o Check Slot Condition (see Section 8.4.9.3)
 - o Set Parameters directive (see Section 8.4.10.3) for URI A (see Section 8.4.8)
 - Second Sequence:
 - o Override Parameters directive (see Section 8.4.10.3, Section 8.4.8) for Slot B
 - o Check Slot Condition (see Section 8.4.9.3)
 - o Set Parameters directive (see Section 8.4.10.3) for URI B (see Section 8.4.8)
- * Fetch

If Trusted Invocation (Section 7.2) is used, only the run sequence is added to this template, since the common sequence is populated by this template:

- * Set Component Index directive (see Section 8.4.10.1)
- * Try Each
 - First Sequence:
 - o Override Parameters directive (see Section 8.4.10.3, Section 8.4.8) for Slot A

- o Check Slot Condition (see Section 8.4.9.3)
- Second Sequence:
 - o Override Parameters directive (see Section 8.4.10.3, Section 8.4.8) for Slot B
 - o Check Slot Condition (see Section 8.4.9.3)
- * Run

NOTE: Any test can be used to select between images, Check Slot Condition is used in this template because it is a typical test for execute-in-place devices.

8. Metadata Structure

The metadata for SUIIT updates is composed of several primary constituent parts: the Envelope, Authentication Information, Manifest, and Severable Elements.

For a diagram of the metadata structure, see Section 5.

8.1. Encoding Considerations

The map indices in the envelope encoding are reset to 1 for each map within the structure. This is to keep the indices as small as possible. The goal is to keep the index objects to single bytes (CBOR positive integers 1-23).

Wherever enumerations are used, they are started at 1. This allows detection of several common software errors that are caused by uninitialized variables. Positive numbers in enumerations are reserved for IANA registration. Negative numbers are used to identify application-specific values, as described in Section 11.

All elements of the envelope must be wrapped in a bstr to minimize the complexity of the code that evaluates the cryptographic integrity of the element and to ensure correct serialization for integrity and authenticity checks.

8.2. Envelope

The Envelope contains each of the other primary constituent parts of the SUIIT metadata. It allows for modular processing of the manifest by ordering components in the expected order of processing.

The Envelope is encoded as a CBOR Map. Each element of the Envelope is enclosed in a bstr, which allows computation of a message digest against known bounds.

8.3. Authenticated Manifests

The suit-authentication-wrapper contains a SUIIT Digest Container (see Section 10) and one or more SUIIT Authentication Blocks. The SUIIT_Digest carries the result of computing the indicated hash algorithm over the suit-manifest element. A signing application MUST verify the suit-manifest element against the SUIIT_Digest prior to signing. A SUIIT Authentication Block is implemented as COSE_Mac_Tagged, COSE_Mac0_Tagged, COSE_Sign_Tagged or COSE_Sign1_Tagged structures with detached payloads, as described in RFC 8152 [RFC8152].

For COSE_Sign and COSE_Sign1 a special signature structure (called Sig_structure) has to be created onto which the selected digital signature algorithm is applied to, see Section 4.4 of [RFC8152] for details. This specification requires Sig_structure to be populated as follows: * The external_aad field MUST be set to a zero-length binary string (i.e. there is no external additional authenticated data). * The payload field contains the SUIIT_Digest wrapped in a bstr, as per the requirements in Section 4.4 of RFC 8152. All other fields in the Sig_structure are populated as described in Section 4.4 of [RFC8152].

Likewise, Section 6.3 of [RFC8152] describes the details for computing a MAC and the fields of the MAC_structure need to be populated. The rules for external_aad and the payload fields described in the paragraph above also apply to this structure.

The suit-authentication-wrapper MUST come before the suit-manifest element, regardless of canonical encoding of CBOR.

A SUIIT_Envelope that has not had authentication information added MUST still contain the suit-authentication-wrapper element, but the content MUST be a list containing only the SUIIT_Digest.

8.4. Manifest

The manifest contains:

- * a version number (see Section 8.4.1)
- * a sequence number (see Section 8.4.2)
- * a reference URI (see Section 8.4.3)

- * a common structure with information that is shared between command sequences (see Section 8.4.5)
- * one or more lists of commands that the Recipient should perform (see Section 8.4.6)
- * a reference to the full manifest (see Section 8.4.3)
- * human-readable text describing the manifest found in the SUIF_Envelope (see Section 8.4.4)

The Text section, or any Command Sequence of the Update Procedure (Image Fetch, Image Installation) can be either a CBOR structure or a SUIF_Digest. In each of these cases, the SUIF_Digest provides for a severable element. Severable elements are RECOMMENDED to implement. In particular, the human-readable text SHOULD be severable, since most useful text elements occupy more space than a SUIF_Digest, but are not needed by the Recipient. Because SUIF_Digest is a CBOR Array and each severable element is a CBOR bstr, it is straight-forward for a Recipient to determine whether an element has been severed. The key used for a severable element is the same in the SUIF_Manifest and in the SUIF_Envelope so that a Recipient can easily identify the correct data in the envelope. See Section 8.4.11 for more detail.

8.4.1. suit-manifest-version

The suit-manifest-version indicates the version of serialization used to encode the manifest. Version 1 is the version described in this document. suit-manifest-version is REQUIRED to implement.

8.4.2. suit-manifest-sequence-number

The suit-manifest-sequence-number is a monotonically increasing anti-rollback counter. Each Recipient MUST reject any manifest that has a sequence number lower than its current sequence number. For convenience, an implementer MAY use a UTC timestamp in seconds as the sequence number. suit-manifest-sequence-number is REQUIRED to implement.

8.4.3. suit-reference-uri

suit-reference-uri is a text string that encodes a URI where a full version of this manifest can be found. This is convenient for allowing management systems to show the severed elements of a manifest when this URI is reported by a Recipient after installation.

8.4.4. suit-text

suit-text SHOULD be a severable element. suit-text is a map containing two different types of pair:

- * integer => text
- * SUIT_Component_Identifier => map

Each SUIT_Component_Identifier => map entry contains a map of integer => text values. All SUIT_Component_Identifiers present in suit-text MUST also be present in suit-common (Section 8.4.5).

suit-text contains all the human-readable information that describes any and all parts of the manifest, its payload(s) and its resource(s). The text section is typically severable, allowing manifests to be distributed without the text, since end-nodes do not require text. The meaning of each field is described below.

Each section MAY be present. If present, each section MUST be as described. Negative integer IDs are reserved for application-specific text values.

The following table describes the text fields available in suit-text:

| CDDL Structure | Description |
|--------------------------------|---|
| suit-text-manifest-description | Free text description of the manifest |
| suit-text-update-description | Free text description of the update |
| suit-text-manifest-json-source | The JSON-formatted document that was used to create the manifest |
| suit-text-manifest-yaml-source | The YAML ([YAML])-formatted document that was used to create the manifest |

Table 2

The following table describes the text fields available in each map identified by a SUIT_Component_Identifier.

| CDDL Structure | Description |
|---------------------------------|---|
| suit-text-vendor-name | Free text vendor name |
| suit-text-model-name | Free text model name |
| suit-text-vendor-domain | The domain used to create the vendor-id condition |
| suit-text-model-info | The information used to create the class-id condition |
| suit-text-component-description | Free text description of each component in the manifest |
| suit-text-component-version | A free text representation of the component version |

Table 3

suit-text is OPTIONAL to implement.

8.4.5. suit-common

suit-common encodes all the information that is shared between each of the command sequences, including: suit-components, and suit-common-sequence. suit-common is REQUIRED to implement.

suit-components is a list of SUIF_Component_Identifier (Section 8.4.5.1) blocks that specify the component identifiers that will be affected by the content of the current manifest. suit-components is REQUIRED to implement.

suit-common-sequence is a SUIF_Command_Sequence to execute prior to executing any other command sequence. Typical actions in suit-common-sequence include setting expected Recipient identity and image digests when they are conditional (see Section 8.4.10.2 and Section 7.7 for more information on conditional sequences). suit-common-sequence is RECOMMENDED to implement. It is REQUIRED if the optimizations described in Section 6.2.1 will be used. Whenever a parameter or Try Each command is required by more than one Command Sequence, placing that parameter or command in suit-common-sequence results in a smaller encoding.

8.4.5.1. SUIT_Component_Identifier

A component is a unit of code or data that can be targeted by an update. To facilitate composite devices, components are identified by a list of CBOR byte strings, which allows construction of hierarchical component structures. Components are identified by Component Identifiers, but referenced in commands by Component Index; Component Identifiers are arrays of binary strings and a Component Index is an index into the array of Component Identifiers.

A Component Identifier can be trivial, such as the simple array [h'00']. It can also represent a filesystem path by encoding each segment of the path as an element in the list. For example, the path "/usr/bin/env" would encode to ['usr','bin','env'].

This hierarchical construction allows a component identifier to identify any part of a complex, multi-component system.

8.4.6. SUIT_Command_Sequence

A SUIT_Command_Sequence defines a series of actions that the Recipient MUST take to accomplish a particular goal. These goals are defined in the manifest and include:

1. Payload Fetch: suit-payload-fetch is a SUIT_Command_Sequence to execute in order to obtain a payload. Some manifests may include these actions in the suit-install section instead if they operate in a streaming installation mode. This is particularly relevant for constrained devices without any temporary storage for staging the update. suit-payload-fetch is OPTIONAL to implement.
2. Payload Installation: suit-install is a SUIT_Command_Sequence to execute in order to install a payload. Typical actions include verifying a payload stored in temporary storage, copying a staged payload from temporary storage, and unpacking a payload. suit-install is OPTIONAL to implement.
3. Image Validation: suit-validate is a SUIT_Command_Sequence to execute in order to validate that the result of applying the update is correct. Typical actions involve image validation. suit-validate is REQUIRED to implement.
4. Image Loading: suit-load is a SUIT_Command_Sequence to execute in order to prepare a payload for execution. Typical actions include copying an image from permanent storage into RAM, optionally including actions such as decryption or decompression. suit-load is OPTIONAL to implement.

5. Run or Boot: `suit-run` is a `SUIIT_Command_Sequence` to execute in order to run an image. `suit-run` typically contains a single instruction: the "run" directive. `suit-run` is OPTIONAL to implement.

Goals 1,2 form the Update Procedure. Goals 4,5,6 form the Invocation Procedure.

Each Command Sequence follows exactly the same structure to ensure that the parser is as simple as possible.

Lists of commands are constructed from two kinds of element:

1. Conditions that MUST be true and any failure is treated as a failure of the update/load/invocation
2. Directives that MUST be executed.

Each condition is composed of:

1. A command code identifier
2. A `SUIIT_Reporting_Policy` (Section 8.4.7)

Each directive is composed of:

1. A command code identifier
2. An argument block or a `SUIIT_Reporting_Policy` (Section 8.4.7)

Argument blocks are consumed only by flow-control directives:

- * Set Component Index
- * Set/Override Parameters
- * Try Each
- * Run Sequence

Reporting policies provide a hint to the manifest processor of whether to add the success or failure of a command to any report that it generates.

Many conditions and directives apply to a given component, and these generally grouped together. Therefore, a special command to set the current component index is provided. This index is a numeric index into the Component Identifier table defined at the beginning of the manifest.

To facilitate optional conditions, a special directive, `suit-directive-try-each` (Section 8.4.10.2), is provided. It runs several new lists of conditions/directives, one after another, that are contained as an argument to the directive. By default, it assumes that a failure of a condition should not indicate a failure of the update/invocation, but a parameter is provided to override this behavior. See `suit-parameter-soft-failure` (Section 8.4.8.14).

8.4.7. Reporting Policy

To facilitate construction of Reports that describe the success or failure of a given Procedure, each command is given a Reporting Policy. This is an integer bitfield that follows the command and indicates what the Recipient should do with the Record of executing the command. The options are summarized in the table below.

| Policy | Description |
|--|--|
| <code>suit-send-record-on-success</code> | Record when the command succeeds |
| <code>suit-send-record-on-failure</code> | Record when the command fails |
| <code>suit-send-sysinfo-success</code> | Add system information when the command succeeds |
| <code>suit-send-sysinfo-failure</code> | Add system information when the command fails |

Table 4

Any or all of these policies may be enabled at once.

At the completion of each command, a Manifest Processor MAY forward information about the command to a Reporting Engine, which is responsible for reporting boot or update status to a third party. The Reporting Engine is entirely implementation-defined, the reporting policy simply facilitates the Reporting Engine's interface to the SUIT Manifest Processor.

The information elements provided to the Reporting Engine are:

- * The reporting policy
- * The result of the command
- * The values of parameters consumed by the command
- * The system information consumed by the command

Together, these elements are called a Record. A group of Records is a Report.

If the component index is set to True or an array when a command is executed with a non-zero reporting policy, then the Reporting Engine MUST receive one Record for each Component, in the order expressed in the Components list or the component index array.

This specification does not define a particular format of Records or Reports. This specification only defines hints to the Reporting Engine for which Records it should aggregate into the Report. The Reporting Engine MAY choose to ignore these hints and apply its own policy instead.

When used in a Invocation Procedure, the report MAY form the basis of an attestation report. When used in an Update Process, the report MAY form the basis for one or more log entries.

8.4.8. SUIIT_Parameters

Many conditions and directives require additional information. That information is contained within parameters that can be set in a consistent way. This allows reuse of parameters between commands, thus reducing manifest size.

Most parameters are scoped to a specific component. This means that setting a parameter for one component has no effect on the parameters of any other component. The only exceptions to this are two Manifest Processor parameters: Strict Order and Soft Failure.

The defined manifest parameters are described below.

| Name | CDDL Structure | Reference |
|------------------|----------------------------------|------------------|
| Vendor ID | suit-parameter-vendor-identifier | Section 8.4.8.3 |
| Class ID | suit-parameter-class-identifier | Section 8.4.8.4 |
| Device ID | suit-parameter-device-identifier | Section 8.4.8.5 |
| Image Digest | suit-parameter-image-digest | Section 8.4.8.6 |
| Image Size | suit-parameter-image-size | Section 8.4.8.7 |
| Component Slot | suit-parameter-component-slot | Section 8.4.8.8 |
| URI | suit-parameter-uri | Section 8.4.8.9 |
| Source Component | suit-parameter-source-component | Section 8.4.8.10 |
| Run Args | suit-parameter-run-args | Section 8.4.8.11 |
| Fetch Arguments | suit-parameter-fetch-arguments | Section 8.4.8.12 |
| Strict Order | suit-parameter-strict-order | Section 8.4.8.13 |
| Soft Failure | suit-parameter-soft-failure | Section 8.4.8.14 |
| Custom | suit-parameter-custom | Section 8.4.8.15 |

Table 5

CBOR-encoded object parameters are still wrapped in a bstr. This is because it allows a parser that is aggregating parameters to reference the object with a single pointer and traverse it without understanding the contents. This is important for modularization and division of responsibility within a pull parser. The same consideration does not apply to Directives because those elements are invoked with their arguments immediately.

8.4.8.1. CBOR PEN UUID Namespace Identifier

The CBOR PEN UUID Namespace Identifier is constructed as follows:

It uses the OID Namespace as a starting point, then uses the CBOR absolute OID encoding for the IANA PEN OID (1.3.6.1.4.1):

```
D8 6F          # tag(111)
 45           # bytes(5)
# Absolute OID encoding of IANA Private Enterprise Number:
#   1.3. 6. 1. 4. 1
    2B 06 01 04 01 # X.690 Clause 8.19
```

Computing a type 5 UUID from these produces:

```
NAMESPACE_CBOR_PEN = UUID5(NAMESPACE_OID, h'D86F452B06010401')
NAMESPACE_CBOR_PEN = 47fbdabb-f2e4-55f0-bb39-3620c2f6df4e
```

8.4.8.2. Constructing UUIDs

Several conditions use identifiers to determine whether a manifest matches a given Recipient or not. These identifiers are defined to be RFC 4122 [RFC4122] UUIDs. These UUIDs are not human-readable and are therefore used for machine-based processing only.

A Recipient MAY match any number of UUIDs for vendor or class identifier. This may be relevant to physical or software modules. For example, a Recipient that has an OS and one or more applications might list one Vendor ID for the OS and one or more additional Vendor IDs for the applications. This Recipient might also have a Class ID that must be matched for the OS and one or more Class IDs for the applications.

Identifiers are used for compatibility checks. They MUST NOT be used as assertions of identity. They are evaluated by identifier conditions (Section 8.4.9.1).

A more complete example: Imagine a device has the following physical components: 1. A host MCU 2. A WiFi module

This same device has three software modules: 1. An operating system 2. A WiFi module interface driver 3. An application

Suppose that the WiFi module's firmware has a proprietary update mechanism and doesn't support manifest processing. This device can report four class IDs:

1. Hardware model/revision

2. OS

3. WiFi module model/revision

4. Application

This allows the OS, WiFi module, and application to be updated independently. To combat possible incompatibilities, the OS class ID can be changed each time the OS has a change to its API.

This approach allows a vendor to target, for example, all devices with a particular WiFi module with an update, which is a very powerful mechanism, particularly when used for security updates.

UUIDs MUST be created according to RFC 4122 [RFC4122]. UUIDs SHOULD use versions 3, 4, or 5, as described in RFC4122. Versions 1 and 2 do not provide a tangible benefit over version 4 for this application.

The RECOMMENDED method to create a vendor ID is:

Vendor ID = UUID5(DNS_PREFIX, vendor domain name)

If the Vendor ID is a UUID, the RECOMMENDED method to create a Class ID is:

Class ID = UUID5(Vendor ID, Class-Specific-Information)

If the Vendor ID is a CBOR PEN (see Section 8.4.8.3), the RECOMMENDED method to create a Class ID is:

```
Class ID = UUID5(  
    UUID5(NAMESPACE_CBOR_PEN, CBOR_PEN),  
    Class-Specific-Information)
```

Class-specific-information is composed of a variety of data, for example:

- * Model number.
- * Hardware revision.
- * Bootloader version (for immutable bootloaders).

8.4.8.3. suit-parameter-vendor-identifier

suit-parameter-vendor-identifier may be presented in one of two ways:

- * A Private Enterprise Number

- * A byte string containing a UUID ([RFC4122])

Private Enterprise Numbers are encoded as a relative OID, according to the definition in [I-D.ietf-cbor-tags-oid]. All PENs are relative to the IANA PEN: 1.3.6.1.4.1.

8.4.8.4. suit-parameter-class-identifier

A RFC 4122 UUID representing the class of the device or component. The UUID is encoded as a 16 byte bstr, containing the raw bytes of the UUID. It MUST be constructed as described in Section 8.4.8.2

8.4.8.5. suit-parameter-device-identifier

A RFC 4122 UUID representing the specific device or component. The UUID is encoded as a 16 byte bstr, containing the raw bytes of the UUID. It MUST be constructed as described in Section 8.4.8.2

8.4.8.6. suit-parameter-image-digest

A fingerprint computed over the component itself, encoded in the SUIF_Digest Section 10 structure. The SUIF_Digest is wrapped in a bstr, as required in Section 8.4.8.

8.4.8.7. suit-parameter-image-size

The size of the firmware image in bytes. This size is encoded as a positive integer.

8.4.8.8. suit-parameter-component-slot

This parameter sets the slot index of a component. Some components support multiple possible Slots (offsets into a storage area). This parameter describes the intended Slot to use, identified by its index into the component's storage area. This slot MUST be encoded as a positive integer.

8.4.8.9. suit-parameter-uri

A URI Reference ([RFC3986]) from which to fetch a resource, encoded as a text string. CBOR Tag 32 is not used because the meaning of the text string is unambiguous in this context.

8.4.8.10. `suit-parameter-source-component`

This parameter sets the source component to be used with either `suit-directive-copy` (Section 8.4.10.5) or with `suit-directive-swap` (Section 8.4.10.8). The current Component, as set by `suit-directive-set-component-index` defines the destination, and `suit-parameter-source-component` defines the source.

8.4.8.11. `suit-parameter-run-args`

This parameter contains an encoded set of arguments for `suit-directive-run` (Section 8.4.10.6). The arguments **MUST** be provided as an implementation-defined bstr.

8.4.8.12. `suit-parameter-fetch-arguments`

An implementation-defined set of arguments to `suit-directive-fetch` (Section 8.4.10.4). Arguments are encoded in a bstr.

8.4.8.13. `suit-parameter-strict-order`

The Strict Order Parameter allows a manifest to govern when directives can be executed out-of-order. This allows for systems that have a sensitivity to order of updates to choose the order in which they are executed. It also allows for more advanced systems to parallelize their handling of updates. Strict Order defaults to True. It **MAY** be set to False when the order of operations does not matter. When arriving at the end of a command sequence, **ALL** commands **MUST** have completed, regardless of the state of `SUIT_Parameter_Strict_Order`. If `SUIT_Parameter_Strict_Order` is returned to True, **ALL** preceding commands **MUST** complete before the next command is executed.

See Section 6.7 for behavioral description of Strict Order.

8.4.8.14. `suit-parameter-soft-failure`

When executing a command sequence inside `suit-directive-try-each` (Section 8.4.10.2) or `suit-directive-run-sequence` (Section 8.4.10.7) and a condition failure occurs, the manifest processor aborts the sequence. For `suit-directive-try-each`, if Soft Failure is True, the next sequence in Try Each is invoked, otherwise `suit-directive-try-each` fails with the condition failure code. In `suit-directive-run-sequence`, if Soft Failure is True the `suit-directive-run-sequence` simply halts with no side-effects and the Manifest Processor continues with the following command, otherwise, the `suit-directive-run-sequence` fails with the condition failure code.

suit-parameter-soft-failure is scoped to the enclosing SUIIT_Command_Sequence. Its value is discarded when SUIIT_Command_Sequence terminates. It MUST NOT be set outside of suit-directive-try-each or suit-directive-run-sequence.

When suit-directive-try-each is invoked, Soft Failure defaults to True. An Update Author may choose to set Soft Failure to False if they require a failed condition in a sequence to force an Abort.

When suit-directive-run-sequence is invoked, Soft Failure defaults to False. An Update Author may choose to make failures soft within a suit-directive-run-sequence.

8.4.8.15. suit-parameter-custom

This parameter is an extension point for any proprietary, application specific conditions and directives. It MUST NOT be used in the common sequence. This effectively scopes each custom command to a particular Vendor Identifier/Class Identifier pair.

8.4.9. SUIIT_Condition

Conditions are used to define mandatory properties of a system in order for an update to be applied. They can be pre-conditions or post-conditions of any directive or series of directives, depending on where they are placed in the list. All Conditions specify a Reporting Policy as described Section 8.4.7. Conditions include:

| Name | CDDL Structure | Reference |
|-------------------|----------------------------------|-----------------|
| Vendor Identifier | suit-condition-vendor-identifier | Section 8.4.9.1 |
| Class Identifier | suit-condition-class-identifier | Section 8.4.9.1 |
| Device Identifier | suit-condition-device-identifier | Section 8.4.9.1 |
| Image Match | suit-condition-image-match | Section 8.4.9.2 |
| Component Slot | suit-condition-component-slot | Section 8.4.9.3 |
| Abort | suit-condition-abort | Section 8.4.9.4 |
| Custom Condition | suit-condition-custom | Section 8.4.9.5 |

Table 6

The abstract description of these conditions is defined in Section 6.4.

Conditions compare parameters against properties of the system. These properties may be asserted in many different ways, including: calculation on-demand, volatile definition in memory, static definition within the manifest processor, storage in known location within an image, storage within a key storage system, storage in One-Time-Programmable memory, inclusion in mask ROM, or inclusion as a register in hardware. Some of these assertion methods are global in scope, such as a hardware register, some are scoped to an individual component, such as storage at a known location in an image, and some assertion methods can be either global or component-scope, based on implementation.

Each condition MUST report a result code on completion. If a condition reports failure, then the current sequence of commands MUST terminate. A subsequent command or command sequence MAY continue executing if `suit-parameter-soft-failure` (Section 8.4.8.14) is set. If a condition requires additional information, this MUST be specified in one or more parameters before the condition is executed.

If a Recipient attempts to process a condition that expects additional information and that information has not been set, it MUST report a failure. If a Recipient encounters an unknown condition, it MUST report a failure.

Condition labels in the positive number range are reserved for IANA registration while those in the negative range are custom conditions reserved for proprietary definition by the author of a manifest processor. See Section 11 for more details.

8.4.9.1. `suit-condition-vendor-identifier`, `suit-condition-class-identifier`, and `suit-condition-device-identifier`

There are three identifier-based conditions: `suit-condition-vendor-identifier`, `suit-condition-class-identifier`, and `suit-condition-device-identifier`. Each of these conditions match a RFC 4122 [RFC4122] UUID that MUST have already been set as a parameter. The installing Recipient MUST match the specified UUID in order to consider the manifest valid. These identifiers are scoped by component in the manifest. Each component MAY match more than one identifier. Care is needed to ensure that manifests correctly identify their targets using these conditions. Using only a generic class ID for a device-specific firmware could result in matching devices that are not compatible.

The Recipient uses the ID parameter that has already been set using the Set Parameters directive. If no ID has been set, this condition fails. `suit-condition-class-identifier` and `suit-condition-vendor-identifier` are REQUIRED to implement. `suit-condition-device-identifier` is OPTIONAL to implement.

Each identifier condition compares the corresponding identifier parameter to a parameter asserted to the Manifest Processor by the Recipient. Identifiers MUST be known to the Manifest Processor in order to evaluate compatibility.

8.4.9.2. `suit-condition-image-match`

Verify that the current component matches the `suit-parameter-image-digest` (Section 8.4.8.6) for the current component. The digest is verified against the digest specified in the Component's parameters list. If no digest is specified, the condition fails. `suit-condition-image-match` is REQUIRED to implement.

8.4.9.3. suit-condition-component-slot

Verify that the slot index of the current component matches the slot index set in suit-parameter-component-slot (Section 8.4.8.8). This condition allows a manifest to select between several images to match a target slot.

8.4.9.4. suit-condition-abort

Unconditionally fail. This operation is typically used in conjunction with suit-directive-try-each (Section 8.4.10.2).

8.4.9.5. suit-condition-custom

suit-condition-custom describes any proprietary, application specific condition. This is encoded as a negative integer, chosen by the firmware developer. If additional information must be provided to the condition, it should be encoded in a custom parameter (a nint) as described in Section 8.4.8. SUIIT_Condition_Custom is OPTIONAL to implement.

8.4.10. SUIIT_Directive

Directives are used to define the behavior of the recipient. Directives include:

| Name | CDDL Structure | Reference |
|---------------------|------------------------------------|------------------|
| Set Component Index | suit-directive-set-component-index | Section 8.4.10.1 |
| Try Each | suit-directive-try-each | Section 8.4.10.2 |
| Override Parameters | suit-directive-override-parameters | Section 8.4.10.3 |
| Fetch | suit-directive-fetch | Section 8.4.10.4 |
| Copy | suit-directive-copy | Section 8.4.10.5 |
| Run | suit-directive-run | Section 8.4.10.6 |
| Run Sequence | suit-directive-run-sequence | Section 8.4.10.7 |
| Swap | suit-directive-swap | Section 8.4.10.8 |

Table 7

The abstract description of these commands is defined in Section 6.4.

When a Recipient executes a Directive, it MUST report a result code. If the Directive reports failure, then the current Command Sequence MUST be terminated.

8.4.10.1. suit-directive-set-component-index

Set Component Index defines the component to which successive directives and conditions will apply. The supplied argument MUST be one of three types:

1. An unsigned integer (REQUIRED to implement in parser)
2. A boolean (REQUIRED to implement in parser ONLY IF 2 or more components supported)

3. An array of unsigned integers (REQUIRED to implement in parser ONLY IF 3 or more components supported)

If the following commands apply to ONE component, an unsigned integer index into the component list is used. If the following commands apply to ALL components, then the boolean value "True" is used instead of an index. If the following commands apply to more than one, but not all components, then an array of unsigned integer indices into the component list is used. See Section 6.5 for more details.

If component index is set to True when a command is invoked, then the command applies to all components, in the order they appear in suit-common-components. When the Manifest Processor invokes a command while the component index is set to True, it must execute the command once for each possible component index, ensuring that the command receives the parameters corresponding to that component index.

8.4.10.2. suit-directive-try-each

This command runs several SUIF_Command_Sequence instances, one after another, in a strict order. Use this command to implement a "try/catch-try/catch" sequence. Manifest processors MAY implement this command.

suit-parameter-soft-failure (Section 8.4.8.14) is initialized to True at the beginning of each sequence. If one sequence aborts due to a condition failure, the next is started. If no sequence completes without condition failure, then suit-directive-try-each returns an error. If a particular application calls for all sequences to fail and still continue, then an empty sequence (nil) can be added to the Try Each Argument.

The argument to suit-directive-try-each is a list of SUIF_Command_Sequence. suit-directive-try-each does not specify a reporting policy.

8.4.10.3. suit-directive-override-parameters

suit-directive-override-parameters replaces any listed parameters that are already set with the values that are provided in its argument. This allows a manifest to prevent replacement of critical parameters.

Available parameters are defined in Section 8.4.8.

suit-directive-override-parameters does not specify a reporting policy.

8.4.10.4. `suit-directive-fetch`

`suit-directive-fetch` instructs the manifest processor to obtain one or more manifests or payloads, as specified by the manifest index and component index, respectively.

`suit-directive-fetch` can target one or more payloads. `suit-directive-fetch` retrieves each component listed in `component-index`. If `component-index` is `True`, instead of an integer, then all current manifest components are fetched. If `component-index` is an array, then all listed components are fetched.

`suit-directive-fetch` typically takes no arguments unless one is needed to modify fetch behavior. If an argument is needed, it must be wrapped in a `bstr` and set in `suit-parameter-fetch-arguments`.

`suit-directive-fetch` reads the `URI` parameter to find the source of the fetch it performs.

8.4.10.5. `suit-directive-copy`

`suit-directive-copy` instructs the manifest processor to obtain one or more payloads, as specified by the component index. As described in Section 6.5 component index may be a single integer, a list of integers, or `True`. `suit-directive-copy` retrieves each component specified by the current `component-index`, respectively.

`suit-directive-copy` reads its source from `suit-parameter-source-component` (Section 8.4.8.10).

If either the source component parameter or the source component itself is absent, this command fails.

8.4.10.6. `suit-directive-run`

`suit-directive-run` directs the manifest processor to transfer execution to the current Component Index. When this is invoked, the manifest processor MAY be unloaded and execution continues in the Component Index. Arguments are provided to `suit-directive-run` through `suit-parameter-run-arguments` (Section 8.4.8.11) and are forwarded to the executable code located in Component Index in an application-specific way. For example, this could form the Linux Kernel Command Line if booting a Linux device.

If the executable code at Component Index is constructed in such a way that it does not unload the manifest processor, then the manifest processor may resume execution after the executable completes. This allows the manifest processor to invoke suitable helpers and to verify them with image conditions.

8.4.10.7. `suit-directive-run-sequence`

To enable conditional commands, and to allow several strictly ordered sequences to be executed out-of-order, `suit-directive-run-sequence` allows the manifest processor to execute its argument as a `SUIT_Command_Sequence`. The argument must be wrapped in a `bstr`.

When a sequence is executed, any failure of a condition causes immediate termination of the sequence.

When `suit-directive-run-sequence` completes, it forwards the last status code that occurred in the sequence. If the `Soft Failure` parameter is true, then `suit-directive-run-sequence` only fails when a directive in the argument sequence fails.

`suit-parameter-soft-failure` (Section 8.4.8.14) defaults to False when `suit-directive-run-sequence` begins. Its value is discarded when `suit-directive-run-sequence` terminates.

8.4.10.8. `suit-directive-swap`

`suit-directive-swap` instructs the manifest processor to move the source to the destination and the destination to the source simultaneously. Swap has nearly identical semantics to `suit-directive-copy` except that `suit-directive-swap` replaces the source with the current contents of the destination in an application-defined way. As with `suit-directive-copy`, if the source component is missing, this command fails.

If `SUIT_Parameter_Compression_Info` or `SUIT_Parameter_Encryption_Info` are present, they MUST be handled in a symmetric way, so that the source is decompressed into the destination and the destination is compressed into the source. The source is decrypted into the destination and the destination is encrypted into the source. `suit-directive-swap` is OPTIONAL to implement.

8.4.11. Integrity Check Values

When the Text section or any Command Sequence of the Update Procedure is made severable, it is moved to the Envelope and replaced with a SUIIT_Digest. The SUIIT_Digest is computed over the entire bstr enclosing the Manifest element that has been moved to the Envelope. Each element that is made severable from the Manifest is placed in the Envelope. The keys for the envelope elements have the same values as the keys for the manifest elements.

Each Integrity Check Value covers the corresponding Envelope Element as described in Section 8.5.

8.5. Severable Elements

Because the manifest can be used by different actors at different times, some parts of the manifest can be removed or "Severed" without affecting later stages of the lifecycle. Severing of information is achieved by separating that information from the signed container so that removing it does not affect the signature. This means that ensuring integrity of severable parts of the manifest is a requirement for the signed portion of the manifest. Severing some parts makes it possible to discard parts of the manifest that are no longer necessary. This is important because it allows the storage used by the manifest to be greatly reduced. For example, no text size limits are needed if text is removed from the manifest prior to delivery to a constrained device.

Elements are made severable by removing them from the manifest, encoding them in a bstr, and placing a SUIIT_Digest of the bstr in the manifest so that they can still be authenticated. The SUIIT_Digest typically consumes 4 bytes more than the size of the raw digest, therefore elements smaller than $(\text{Digest Bits})/8 + 4$ SHOULD NOT be severable. Elements larger than $(\text{Digest Bits})/8 + 4$ MAY be severable, while elements that are much larger than $(\text{Digest Bits})/8 + 4$ SHOULD be severable.

Because of this, all command sequences in the manifest are encoded in a bstr so that there is a single code path needed for all command sequences.

9. Access Control Lists

To manage permissions in the manifest, there are three models that can be used.

First, the simplest model requires that all manifests are authenticated by a single trusted key. This mode has the advantage that only a root manifest needs to be authenticated, since all of its dependencies have digests included in the root manifest.

This simplest model can be extended by adding key delegation without much increase in complexity.

A second model requires an ACL to be presented to the Recipient, authenticated by a trusted party or stored on the Recipient. This ACL grants access rights for specific component IDs or Component Identifier prefixes to the listed identities or identity groups. Any identity can verify an image digest, but fetching into or fetching from a Component Identifier requires approval from the ACL.

A third model allows a Recipient to provide even more fine-grained controls: The ACL lists the Component Identifier or Component Identifier prefix that an identity can use, and also lists the commands and parameters that the identity can use in combination with that Component Identifier.

10. SUIIT Digest Container

The SUIIT digest is a CBOR List containing two elements: an algorithm identifier and a bstr containing the bytes of the digest. Some forms of digest may require additional parameters. These can be added following the digest.

The values of the algorithm identifier are defined by [I-D.ietf-cose-hash-algs]. The following algorithms MUST be implemented by all Manifest Processors:

- * SHA-256 (-16)

The following algorithms MAY be implemented in a Manifest Processor:

- * SHAKE128 (-18)

- * SHA-384 (-43)

- * SHA-512 (-44)

- * SHAKE256 (-45)

11. IANA Considerations

IANA is requested to:

- * allocate CBOR tag 107 in the CBOR Tags registry for the SUIIT Envelope.
- * allocate CBOR tag 1070 in the CBOR Tags registry for the SUIIT Manifest.
- * allocate media type application/suit-envelope in the Media Types registry.
- * setup several registries as described below.

IANA is requested to setup a registry for SUIIT manifests. Several registries defined in the subsections below need to be created.

For each registry, values 0-23 are Standards Action, 24-255 are IETF Review, 256-65535 are Expert Review, and 65536 or greater are First Come First Served.

Negative values -23 to 0 are Experimental Use, -24 and lower are Private Use.

11.1. SUIIT Commands

| Label | Name | Reference |
|-------|---------------------|------------------|
| 1 | Vendor Identifier | Section 8.4.9.1 |
| 2 | Class Identifier | Section 8.4.9.1 |
| 3 | Image Match | Section 8.4.9.2 |
| 4 | Reserved | |
| 5 | Component Slot | Section 8.4.9.3 |
| 12 | Set Component Index | Section 8.4.10.1 |
| 13 | Reserved | |
| 14 | Abort | |
| 15 | Try Each | Section 8.4.10.2 |
| 16 | Reserved | |
| 17 | Reserved | |

| | | |
|------|---------------------|------------------|
| 18 | Reserved | |
| 19 | Reserved | |
| 20 | Override Parameters | Section 8.4.10.3 |
| 21 | Fetch | Section 8.4.10.4 |
| 22 | Copy | Section 8.4.10.5 |
| 23 | Run | Section 8.4.10.6 |
| 24 | Device Identifier | Section 8.4.9.1 |
| 25 | Reserved | |
| 26 | Reserved | |
| 27 | Reserved | |
| 28 | Reserved | |
| 29 | Reserved | |
| 30 | Reserved | |
| 31 | Swap | Section 8.4.10.8 |
| 32 | Run Sequence | Section 8.4.10.7 |
| 33 | Reserved | |
| nint | Custom Condition | Section 8.4.9.5 |

Table 8

11.2. SUIIT Parameters

| Label | Name | Reference |
|-------|--------------|-----------------|
| 1 | Vendor ID | Section 8.4.8.3 |
| 2 | Class ID | Section 8.4.8.4 |
| 3 | Image Digest | Section 8.4.8.6 |

| | | |
|------|------------------|------------------|
| 4 | Reserved | |
| 5 | Component Slot | Section 8.4.8.8 |
| 12 | Strict Order | Section 8.4.8.13 |
| 13 | Soft Failure | Section 8.4.8.14 |
| 14 | Image Size | Section 8.4.8.7 |
| 18 | Reserved | |
| 19 | Reserved | |
| 20 | Reserved | |
| 21 | URI | Section 8.4.8.9 |
| 22 | Source Component | Section 8.4.8.10 |
| 23 | Run Args | Section 8.4.8.11 |
| 24 | Device ID | Section 8.4.8.5 |
| 26 | Reserved | |
| 27 | Reserved | |
| 28 | Reserved | |
| 29 | Reserved | |
| 30 | Reserved | |
| nint | Custom | Section 8.4.8.15 |

Table 9

11.3. SUIIT Text Values

| Label | Name | Reference |
|-------|----------------------|---------------|
| 1 | Manifest Description | Section 8.4.4 |
| 2 | Update Description | Section 8.4.4 |

| | | | |
|------|----------------------|---------------|--|
| 3 | Manifest JSON Source | Section 8.4.4 | |
| 4 | Manifest YAML Source | Section 8.4.4 | |
| nint | Custom | Section 8.4.4 | |

Table 10

11.4. SUIIT Component Text Values

| Label | Name | Reference | |
|-------|----------------------------|---------------|--|
| 1 | Vendor Name | Section 8.4.4 | |
| 2 | Model Name | Section 8.4.4 | |
| 3 | Vendor Domain | Section 8.4.4 | |
| 4 | Model Info | Section 8.4.4 | |
| 5 | Component Description | Section 8.4.4 | |
| 6 | Component Version | Section 8.4.4 | |
| 7 | Component Version Required | Section 8.4.4 | |
| nint | Custom | Section 8.4.4 | |

Table 11

12. Security Considerations

This document is about a manifest format protecting and describing how to retrieve, install, and invoke firmware images and as such it is part of a larger solution for delivering firmware updates to IoT devices. A detailed security treatment can be found in the architecture [RFC9019] and in the information model [RFC9124] documents.

13. Acknowledgements

We would like to thank the following persons for their support in designing this mechanism:

* Milosch Meriac

- * Geraint Luff
- * Dan Ros
- * John-Paul Stanford
- * Hugo Vincent
- * Carsten Bormann
- * Oeyvind Roenningstad
- * Frank Audun Kvamtroe
- * Krzysztof Chruściński
- * Andrzej Puzdrowski
- * Michael Richardson
- * David Brown
- * Emmanuel Baccelli

14. References

14.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.
- [RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique Identifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <<https://www.rfc-editor.org/info/rfc4122>>.
- [RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <<https://www.rfc-editor.org/info/rfc8152>>.

- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC9019] Moran, B., Tschofenig, H., Brown, D., and M. Meriac, "A Firmware Update Architecture for Internet of Things", RFC 9019, DOI 10.17487/RFC9019, April 2021, <<https://www.rfc-editor.org/info/rfc9019>>.
- [RFC9124] Moran, B., Tschofenig, H., and H. Birkholz, "A Manifest Information Model for Firmware Updates in Internet of Things (IoT) Devices", RFC 9124, DOI 10.17487/RFC9124, January 2022, <<https://www.rfc-editor.org/info/rfc9124>>.

14.2. Informative References

- [I-D.ietf-cbor-tags-oid]
Bormann, C., "Concise Binary Object Representation (CBOR) Tags for Object Identifiers", Work in Progress, Internet-Draft, draft-ietf-cbor-tags-oid-08, 21 May 2021, <<https://www.ietf.org/archive/id/draft-ietf-cbor-tags-oid-08.txt>>.
- [I-D.ietf-cose-hash-algs]
Schaad, J., "CBOR Object Signing and Encryption (COSE): Hash Algorithms", Work in Progress, Internet-Draft, draft-ietf-cose-hash-algs-09, 14 September 2020, <<https://www.ietf.org/archive/id/draft-ietf-cose-hash-algs-09.txt>>.
- [I-D.ietf-suit-firmware-encryption]
Tschofenig, H., Housley, R., and B. Moran, "Firmware Encryption with SUIF Manifests", Work in Progress, Internet-Draft, draft-ietf-suit-firmware-encryption-04, 20 April 2022, <<https://www.ietf.org/archive/id/draft-ietf-suit-firmware-encryption-04.txt>>.
- [I-D.ietf-suit-report]
Moran, B. and H. Birkholz, "Secure Reporting of Update Status", Work in Progress, Internet-Draft, draft-ietf-suit-report-01, 12 January 2022, <<https://www.ietf.org/archive/id/draft-ietf-suit-report-01.txt>>.

[I-D.ietf-suit-trust-domains]

Moran, B., "SUIT Manifest Extensions for Multiple Trust Domains", Work in Progress, Internet-Draft, draft-ietf-suit-trust-domains-00, 7 March 2022, <<https://www.ietf.org/archive/id/draft-ietf-suit-trust-domains-00.txt>>.

[I-D.ietf-suit-update-management]

Moran, B., "Update Management Extensions for Software Updates for Internet of Things (SUIT) Manifests", Work in Progress, Internet-Draft, draft-ietf-suit-update-management-00, 7 March 2022, <<https://www.ietf.org/archive/id/draft-ietf-suit-update-management-00.txt>>.

[I-D.ietf-teep-architecture]

Pei, M., Tschofenig, H., Thaler, D., and D. Wheeler, "Trusted Execution Environment Provisioning (TEEP) Architecture", Work in Progress, Internet-Draft, draft-ietf-teep-architecture-17, 19 April 2022, <<https://www.ietf.org/archive/id/draft-ietf-teep-architecture-17.txt>>.

[RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <<https://www.rfc-editor.org/info/rfc7228>>.

[YAML] "YAML Ain't Markup Language", 2020, <<https://yaml.org/>>.

Appendix A. A. Full CDDL

In order to create a valid SUIT Manifest document the structure of the corresponding CBOR message MUST adhere to the following CDDL data definition.

To be valid, the following CDDL MUST have the COSE CDDL appended to it. The COSE CDDL can be obtained by following the directions in [RFC8152], Section 1.3.

```
SUIT_Envelope_Tagged = #6.107(SUIT_Envelope)
SUIT_Envelope = {
    suit-authentication-wrapper => bstr .cbor SUIT_Authentication,
    suit-manifest => bstr .cbor SUIT_Manifest,
    SUIT_Severable_Manifest_Members,
    * SUIT_Integrated_Payload,
    * $$SUIT_Envelope_Extensions,
    * (int => bstr)
```

```

    }

    SUIF_Authentication = [
        bstr .cbor SUIF_Digest,
        * bstr .cbor SUIF_Authentication_Block
    ]

    SUIF_Digest = [
        suit-digest-algorithm-id : suit-cose-hash-algs,
        suit-digest-bytes : bstr,
        * $$SUIF_Digest-extensions
    ]

    SUIF_Authentication_Block /= COSE_Mac_Tagged
    SUIF_Authentication_Block /= COSE_Sign_Tagged
    SUIF_Authentication_Block /= COSE_Mac0_Tagged
    SUIF_Authentication_Block /= COSE_Sign1_Tagged

    SUIF_Severable_Manifest_Members = (
        ? suit-payload-fetch => bstr .cbor SUIF_Command_Sequence,
        ? suit-install => bstr .cbor SUIF_Command_Sequence,
        ? suit-text => bstr .cbor SUIF_Text_Map,
        * $$SUIF_severable-members-extensions,
    )

    SUIF_Integrated_Payload = (suit-integrated-payload-key => bstr)
    suit-integrated-payload-key = tstr

    SUIF_Manifest_Tagged = #6.1070(SUIF_Manifest)

    SUIF_Manifest = {
        suit-manifest-version          => 1,
        suit-manifest-sequence-number => uint,
        suit-common                    => bstr .cbor SUIF_Common,
        ? suit-reference-uri           => tstr,
        SUIF_Severable_Members_Choice,
        SUIF_Unseverable_Members,
        * $$SUIF_Manifest_Extensions,
    }

    SUIF_Unseverable_Members = (
        ? suit-validate => bstr .cbor SUIF_Command_Sequence,
        ? suit-load => bstr .cbor SUIF_Command_Sequence,
        ? suit-run => bstr .cbor SUIF_Command_Sequence,
        * $$unseverable-manifest-member-extensions,
    )

    SUIF_Severable_Members_Choice = (

```

```

    ? suit-payload-fetch =>
      bstr .cbor SUIF_Command_Sequence / SUIF_Digest,
    ? suit-install => bstr .cbor SUIF_Command_Sequence / SUIF_Digest,
    ? suit-text => bstr .cbor SUIF_Command_Sequence / SUIF_Digest,
    * $$severable-manifest-members-choice-extensions
  )

SUIF_Common = {
  ? suit-components          => SUIF_Components,
  ? suit-common-sequence     => bstr .cbor SUIF_Common_Sequence,
  * $$SUIF_Common-extensions,
}

SUIF_Components              = [ + SUIF_Component_Identifier ]

SUIF_Dependency = {
  suit-dependency-digest => SUIF_Digest,
  ? suit-dependency-prefix => SUIF_Component_Identifier,
  * $$SUIF_Dependency-extensions,
}

;REQUIRED to implement:
suit-cose-hash-algs /= cose-alg-sha-256

;OPTIONAL to implement:
suit-cose-hash-algs /= cose-alg-shake128
suit-cose-hash-algs /= cose-alg-sha-384
suit-cose-hash-algs /= cose-alg-sha-512
suit-cose-hash-algs /= cose-alg-shake256

SUIF_Component_Identifier = [* bstr]

SUIF_Common_Sequence = [
  + ( SUIF_Condition // SUIF_Common_Commands )
]

SUIF_Common_Commands // = (suit-directive-set-component-index,  IndexArg)
SUIF_Common_Commands // = (suit-directive-run-sequence,
  bstr .cbor SUIF_Command_Sequence)
SUIF_Common_Commands // = (suit-directive-try-each,
  SUIF_Directive_Try_Each_Argument)
SUIF_Common_Commands // = (suit-directive-override-parameters,
  {+ SUIF_Parameters})

IndexArg /= uint
IndexArg /= bool
IndexArg /= [+uint]

```

```

SUIF_Command_Sequence = [ + (
    SUIF_Condition // SUIF_Directive // SUIF_Command_Custom
) ]

SUIF_Command_Custom = (suit-command-custom, bstr/tstr/int/nil)
SUIF_Condition //= (suit-condition-vendor-identifier, SUIF_Rep_Policy)
SUIF_Condition //= (suit-condition-class-identifier, SUIF_Rep_Policy)
SUIF_Condition //= (suit-condition-device-identifier, SUIF_Rep_Policy)
SUIF_Condition //= (suit-condition-image-match, SUIF_Rep_Policy)
SUIF_Condition //= (suit-condition-component-slot, SUIF_Rep_Policy)
SUIF_Condition //= (suit-condition-abort, SUIF_Rep_Policy)

SUIF_Directive //= (suit-directive-set-component-index, IndexArg)
SUIF_Directive //= (suit-directive-run-sequence,
    bstr .cbor SUIF_Command_Sequence)
SUIF_Directive //= (suit-directive-try-each,
    SUIF_Directive_Try_Each_Argument)
SUIF_Directive //= (suit-directive-process-dependency, SUIF_Rep_Policy)
SUIF_Directive //= (suit-directive-override-parameters,
    {+ SUIF_Parameters})
SUIF_Directive //= (suit-directive-fetch, SUIF_Rep_Policy)
SUIF_Directive //= (suit-directive-copy, SUIF_Rep_Policy)
SUIF_Directive //= (suit-directive-swap, SUIF_Rep_Policy)
SUIF_Directive //= (suit-directive-run, SUIF_Rep_Policy)

SUIF_Directive_Try_Each_Argument = [
    2* bstr .cbor SUIF_Command_Sequence,
    ?nil
]

SUIF_Rep_Policy = uint .bits suit-reporting-bits

suit-reporting-bits = &(
    suit-send-record-success : 0,
    suit-send-record-failure : 1,
    suit-send-sysinfo-success : 2,
    suit-send-sysinfo-failure : 3
)

SUIF_Parameters //= (suit-parameter-vendor-identifier =>
    (RFC4122_UUID / cbor-pen))
cbor-pen = #6.112(bstr)

SUIF_Parameters //= (suit-parameter-class-identifier => RFC4122_UUID)
SUIF_Parameters //= (suit-parameter-image-digest
    => bstr .cbor SUIF_Digest)
SUIF_Parameters //= (suit-parameter-image-size => uint)
SUIF_Parameters //= (suit-parameter-component-slot => uint)

```

```
SUIT_Parameters //= (suit-parameter-uri => tstr)
SUIT_Parameters //= (suit-parameter-source-component => uint)
SUIT_Parameters //= (suit-parameter-run-args => bstr)

SUIT_Parameters //= (suit-parameter-device-identifier => RFC4122_UUID)

SUIT_Parameters //= (suit-parameter-custom => int/bool/tstr/bstr)

SUIT_Parameters //= (suit-parameter-strict-order => bool)
SUIT_Parameters //= (suit-parameter-soft-failure => bool)

RFC4122_UUID = bstr .size 16

SUIT_Text_Map = {
  SUIT_Text_Keys,
  * SUIT_Component_Identifier => {
    SUIT_Text_Component_Keys
  }
}

SUIT_Text_Component_Keys = (
  ? suit-text-vendor-name           => tstr,
  ? suit-text-model-name           => tstr,
  ? suit-text-vendor-domain        => tstr,
  ? suit-text-model-info           => tstr,
  ? suit-text-component-description => tstr,
  ? suit-text-component-version    => tstr,
  * $$suit-text-component-key-extensions
)

SUIT_Text_Keys = (
  ? suit-text-manifest-description => tstr,
  ? suit-text-update-description  => tstr,
  ? suit-text-manifest-json-source => tstr,
  ? suit-text-manifest-yaml-source => tstr,
  * $$suit-text-key-extensions
)

suit-authentication-wrapper = 2
suit-manifest = 3

;REQUIRED to implement:
cose-alg-sha-256 = -16

;OPTIONAL to implement:
cose-alg-shake128 = -18
cose-alg-sha-384 = -43
cose-alg-sha-512 = -44
```

```
cose-alg-shake256 = -45

suit-manifest-version = 1
suit-manifest-sequence-number = 2
suit-common = 3
suit-reference-uri = 4
suit-payload-fetch = 8
suit-install = 9
suit-validate = 10
suit-load = 11
suit-run = 12
suit-text = 13

suit-components = 2
suit-common-sequence = 4

suit-command-custom = nint

suit-condition-vendor-identifier = 1
suit-condition-class-identifier = 2
suit-condition-image-match = 3
suit-condition-component-slot = 5

suit-condition-abort = 14
suit-condition-device-identifier = 24

suit-directive-set-component-index = 12
suit-directive-try-each = 15
suit-directive-override-parameters = 20
suit-directive-fetch = 21
suit-directive-copy = 22
suit-directive-run = 23

suit-directive-swap = 31
suit-directive-run-sequence = 32

suit-parameter-vendor-identifier = 1
suit-parameter-class-identifier = 2
suit-parameter-image-digest = 3
suit-parameter-component-slot = 5

suit-parameter-strict-order = 12
suit-parameter-soft-failure = 13
suit-parameter-image-size = 14

suit-parameter-uri = 21
suit-parameter-source-component = 22
suit-parameter-run-args = 23
```

```

suit-parameter-device-identifier = 24

suit-parameter-custom = nint

suit-text-manifest-description = 1
suit-text-update-description   = 2
suit-text-manifest-json-source = 3
suit-text-manifest-yaml-source = 4

suit-text-vendor-name          = 1
suit-text-model-name           = 2
suit-text-vendor-domain        = 3
suit-text-model-info            = 4
suit-text-component-description = 5
suit-text-component-version     = 6

```

Appendix B. B. Examples

The following examples demonstrate a small subset of the functionality of the manifest. Even a simple manifest processor can execute most of these manifests.

The examples are signed using the following ECDSA secp256r1 key:

```

-----BEGIN PRIVATE KEY-----
MIGHAgEAMBMGBYqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgApZYjZCUGLM50VBC
CjYStX+09jGmnyJPrpDLTz/hiXOhRANCAASEloEarguqq9JhVxie7NomvqqL8Rtv
P+bitWWchdvArTsfKktsCYExwKNtrNHXi9OB3N+wnAUtszmR23M4tKiW
-----END PRIVATE KEY-----

```

The corresponding public key can be used to verify these examples:

```

-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEhJaBGq4LqqvSYVcYnuzaJr6qi/Eb
bz/m4rVlnIXbwK07HypLbAmBMcCjbazRl4vTgdzfsJwFLbM5kdtzOLSolg==
-----END PUBLIC KEY-----

```

Each example uses SHA256 as the digest function.

Note that reporting policies are declared for each non-flow-control command in these examples. The reporting policies used in the examples are described in the following tables.

| Policy | Label |
|-----------------------------|----------|
| suit-send-record-on-success | Rec-Pass |
| suit-send-record-on-failure | Rec-Fail |
| suit-send-sysinfo-success | Sys-Pass |
| suit-send-sysinfo-failure | Sys-Fail |

Table 12

| Command | Sys-Fail | Sys-Pass | Rec-Fail | Rec-Pass |
|----------------------------------|----------|----------|----------|----------|
| suit-condition-vendor-identifier | 1 | 1 | 1 | 1 |
| suit-condition-class-identifier | 1 | 1 | 1 | 1 |
| suit-condition-image-match | 1 | 1 | 1 | 1 |
| suit-condition-component-slot | 0 | 1 | 0 | 1 |
| suit-directive-fetch | 0 | 0 | 1 | 0 |
| suit-directive-copy | 0 | 0 | 1 | 0 |
| suit-directive-run | 0 | 0 | 1 | 0 |

Table 13

B.1. Example 0: Secure Boot

This example covers the following templates:

- * Compatibility Check (Section 7.1)
- * Secure Boot (Section 7.2)

It also serves as the minimum example.

```

107({
  / authentication-wrapper / 2:<<[
    digest: <<[
      / algorithm-id / -16 / "sha256" /,
      / digest-bytes /
h'a6c4590ac53043a98e8c4106e1e31b305516d7cf0a655eddfac6d45c810e036a'
    ]>>,
    signature: <<18([
      / protected / <<{
        / alg / 1:-7 / "ES256" /,
      }>>,
      / unprotected / {
      },
      / payload / F6 / nil /,
      / signature / h'd11a2dd9610fb62a707335f58407922570
9f96e8117e7eeed98a2f207d05c8ecfba1755208f6abea977b8a6efe3bc2ca3215e119
3be201467d052b42db6b7287'
    ]>>
  ]
}>>,
/ manifest / 3:<<{
  / manifest-version / 1:1,
  / manifest-sequence-number / 2:0,
  / common / 3:<<{
    / components / 2:[
      [h'00']
    ],
    / common-sequence / 4:<<[
      / directive-override-parameters / 20,{
        / vendor-id /
1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
        / class-id /
2:h'1492af1425695e48bf429b2d51f2ab45' /
1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
        / image-digest / 3:<<[
          / algorithm-id / -16 / "sha256" /,
          / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
        ]>>,
        / image-size / 14:34768,
      } ,
      / condition-vendor-identifier / 1,15 ,
      / condition-class-identifier / 2,15
    ]>>,
  }>>,
}>>,

```

```

        / validate / 10:<<[
          / condition-image-match / 3,15
        ]>>,
        / run / 12:<<[
          / directive-run / 23,2
        ]>>,
      ]>>,
    })

```

Total size of Envelope without COSE authentication object: 161

Envelope:

```

d86ba2025827815824822f5820a6c4590ac53043a98e8c4106e1e31b3055
16d7cf0a655eddfac6d45c810e036a035871a50101020003585fa2028181
41000458568614a40150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492
af1425695e48bf429b2d51f2ab45035824822f5820001122334455667788
99aabbccddeeff0123456789abcdeffedcba98765432100e1987d0010f02
0f0a4382030f0c43821702

```

Total size of Envelope with COSE authentication object: 237

Envelope with COSE authentication object:

```

d86ba2025873825824822f5820a6c4590ac53043a98e8c4106e1e31b3055
16d7cf0a655eddfac6d45c810e036a584ad28443a10126a0f65840d11a2d
d9610fb62a707335f584079225709f96e8117e7eed98a2f207d05c8ecfb
a1755208f6abea977b8a6efe3bc2ca3215e1193be201467d052b42db6b72
87035871a50101020003585fa202818141000458568614a40150fa6b4a53
d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab45
035824822f582000112233445566778899aabbccddeeff0123456789abcd
effedcba98765432100e1987d0010f020f0a4382030f0c43821702

```

B.2. Example 1: Simultaneous Download and Installation of Payload

This example covers the following templates:

- * Compatibility Check (Section 7.1)
- * Firmware Download (Section 7.3)

Simultaneous download and installation of payload. No secure boot is present in this example to demonstrate a download-only manifest.

```

107({
  / authentication-wrapper / 2:<<[
    digest: <<[
      / algorithm-id / -16 / "sha256" /,
      / digest-bytes /
h'60c61d6eb7a1aaeddc49ce8157a55cff0821537eeee77a4ded44155b03045132'
    ]>>,
    signature: <<18([
      / protected / <<{
        / alg / 1:-7 / "ES256" /,
      }>>,
      / unprotected / {
        },
      / payload / F6 / nil /,
      / signature / h'5249dacaf0ffc8326931b09586eb7e3769
e71a0e6a40ad8153db4980db9b05bd1742ddb46085fall1e62b65a79895c12ac7abe266
8ccc5afdd74466aed7bca389'
    ]>>
  ]
]>>,
/ manifest / 3:<<{
  / manifest-version / 1:1,
  / manifest-sequence-number / 2:1,
  / common / 3:<<{
    / components / 2:[
      [h'00']
    ],
    / common-sequence / 4:<<[
      / directive-override-parameters / 20,{
        / vendor-id /
1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
        / class-id /
2:h'1492af1425695e48bf429b2d51f2ab45' /
1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
        / image-digest / 3:<<[
          / algorithm-id / -16 / "sha256" /,
          / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
        ]>>,
        / image-size / 14:34768,
      } ,
      / condition-vendor-identifier / 1,15 ,
      / condition-class-identifier / 2,15
    ]>>,
  }>>,
/ install / 9:<<[
  / directive-set-parameters / 19,{

```

```

        / uri / 21:'http://example.com/file.bin',
      } ,
      / directive-fetch / 21,2 ,
      / condition-image-match / 3,15
    ]>>,
    / validate / 10:<<[
      / condition-image-match / 3,15
    ]>>,
  ]>>,
})

```

Total size of Envelope without COSE authentication object: 196

Envelope:

```

d86ba2025827815824822f582060c61d6eb7a1aaeddc49ce8157a55cff08
21537eeee77a4ded44155b03045132035894a50101020103585fa2028181
41000458568614a40150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492
af1425695e48bf429b2d51f2ab45035824822f5820001122334455667788
99aabbccddeeff0123456789abcdeffedcba98765432100e1987d0010f02
0f0958258613a115781b687474703a2f2f6578616d706c652e636f6d2f66
696c652e62696e1502030f0a4382030f

```

Total size of Envelope with COSE authentication object: 272

Envelope with COSE authentication object:

```

d86ba2025873825824822f582060c61d6eb7a1aaeddc49ce8157a55cff08
21537eeee77a4ded44155b03045132584ad28443a10126a0f658405249da
caf0ffc8326931b09586eb7e3769e71a0e6a40ad8153db4980db9b05bd17
42ddb46085fa11e62b65a79895c12ac7abe2668ccc5afdd74466aed7bca3
89035894a50101020103585fa202818141000458568614a40150fa6b4a53
d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab45
035824822f582000112233445566778899aabbccddeeff0123456789abcd
effedcba98765432100e1987d0010f020f0958258613a115781b68747470
3a2f2f6578616d706c652e636f6d2f66696c652e62696e1502030f0a4382
030f

```

B.3. Example 2: Simultaneous Download, Installation, Secure Boot, Severed Fields

This example covers the following templates:

- * Compatibility Check (Section 7.1)
- * Secure Boot (Section 7.2)
- * Firmware Download (Section 7.3)

This example also demonstrates severable elements (Section 5.4), and text (Section 8.4.4).

```

107({
  / authentication-wrapper / 2:<<[
    digest: <<[
      / algorithm-id / -16 / "sha256" /,
      / digest-bytes /
h'e45dcdb2074b951f1c88b866469939c2a83ed433a31fc7dfcb3f63955bd943ec'
    ]>>,
    signature: <<18([
      / protected / <<{
        / alg / 1:-7 / "ES256" /,
      }>>,
      / unprotected / {
      },
      / payload / F6 / nil /,
      / signature / h'b4fd3a6a18fe1062573488cf24ac96ef9f
30ac746696e50be96533b356b8156e4332587fe6f4e8743ae525d72005fddd4c1213d5
5a8061b2ce67b83640f4777c'
    ]>>
  ]
]>>,
  / manifest / 3:<<{
    / manifest-version / 1:1,
    / manifest-sequence-number / 2:2,
    / common / 3:<<{
      / components / 2:[
        [h'00']
      ],
      / common-sequence / 4:<<[
        / directive-override-parameters / 20,{
          / vendor-id /
1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
          / class-id /
2:h'1492af1425695e48bf429b2d51f2ab45' /
1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
          / image-digest / 3:<<[
            / algorithm-id / -16 / "sha256" /,
            / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
          ]>>,
          / image-size / 14:34768,
        } ,
        / condition-vendor-identifier / 1,15 ,
        / condition-class-identifier / 2,15
      ]>>,
    }
  ]>>,

```

```

    }>>,
    / install / 9:[
      / algorithm-id / -16 / "sha256" /,
      / digest-bytes /
h'3ee96dc79641970ae46b929ccf0b72ba9536dd846020dbdc9f949d84ea0e18d2'
    ],
    / validate / 10:<<[
      / condition-image-match / 3,15
    ]>>,
    / run / 12:<<[
      / directive-run / 23,2
    ]>>,
    / text / 13:[
      / algorithm-id / -16 / "sha256" /,
      / digest-bytes /
h'2bfc4d0cc6680be7dd9f5ca30aa2bb5d1998145de33d54101b80e2ca49faf918'
    ],
  }>>,
  / install / 9:<<[
    / directive-set-parameters / 19,{
      / uri /
21:'http://example.com/very/long/path/to/file/file.bin',
    } ,
    / directive-fetch / 21,2 ,
    / condition-image-match / 3,15
  ]>>,
  / text / 13:<<{
    [h'00']:{
      / vendor-domain / 3:'arm.com',
      / component-description / 5:'This component is a
demonstration. The digest is a sample pattern, not a real one.',
    }
  }>>,
})

```

Total size of the Envelope without COSE authentication object or
Severable Elements: 235

Envelope:

```

d86ba2025827815824822f5820e45dcdb2074b951f1c88b866469939c2a8
3ed433a31fc7dfcb3f63955bd943ec0358bba70101020203585fa2028181
41000458568614a40150fa6b4a53d5ad5fdfe9de663e4d41ffe02501492
af1425695e48bf429b2d51f2ab45035824822f5820001122334455667788
99aabbccddeeff0123456789abcdeffedcba98765432100e1987d0010f02
0f09822f58203ee96dc79641970ae46b929ccf0b72ba9536dd846020dbdc
9f949d84ea0e18d20a4382030f0c438217020d822f58202bfc4d0cc6680b
e7dd9f5ca30aa2bb5d1998145de33d54101b80e2ca49faf918

```

Total size of the Envelope with COSE authentication object but without Severable Elements: 311

Envelope:

```
d86ba2025873825824822f5820e45dcdb2074b951f1c88b866469939c2a8
3ed433a31fc7dfcb3f63955bd943ec584ad28443a10126a0f65840b4fd3a
6a18fe1062573488cf24ac96ef9f30ac746696e50be96533b356b8156e43
32587fe6f4e8743ae525d72005fddd4c1213d55a8061b2ce67b83640f477
7c0358bba70101020203585fa202818141000458568614a40150fa6b4a53
d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab45
035824822f582000112233445566778899aabbccddeeff0123456789abcd
effedcba98765432100e1987d0010f020f09822f58203ee96dc79641970a
e46b929ccf0b72ba9536dd846020dbdc9f949d84ea0e18d20a4382030f0c
438217020d822f58202bfc4d0cc6680be7dd9f5ca30aa2bb5d1998145de3
3d54101b80e2ca49faf918
```

Total size of Envelope with COSE authentication object and Severable Elements: 894

Envelope with COSE authentication object:

d86ba4025873825824822f5820e45dcdb2074b951f1c88b866469939c2a8
3ed433a31fc7dfcb3f63955bd943ec584ad28443a10126a0f65840b4fd3a
6a18fe1062573488cf24ac96ef9f30ac746696e50be96533b356b8156e43
32587fe6f4e8743ae525d72005fddd4c1213d55a8061b2ce67b83640f477
7c0358bba70101020203585fa202818141000458568614a40150fa6b4a53
d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab45
035824822f582000112233445566778899aabbccddeeff0123456789abcd
effedcba98765432100e1987d0010f020f09822f58203ee96dc79641970a
e46b929ccf0b72ba9536dd846020dbdc9f949d84ea0e18d20a4382030f0c
438217020d822f58202bfc4d0cc6680be7dd9f5ca30aa2bb5d1998145de3
3d54101b80e2ca49faf91809583c8613a1157832687474703a2f2f657861
6d706c652e636f6d2f766572792f6c6f6e672f706174682f746f2f66696c
652f66696c652e62696e1502030f0d590204a20179019d2323204578616d
706c6520323a2053696d756c74616e656f757320446f776e6c6f61642c20
496e7374616c6c6174696f6e2c2053656375726520426f6f742c20536576
65726564204669656c64730a0a2020202054686973206578616d706c6520
636f766572732074686520666f6c6c6f77696e672074656d706c61746573
3a0a202020200a202020202a20436f6d7061746962696c69747920436865
636b20287b7b74656d706c6174652d636f6d7061746962696c6974792d63
6865636b7d7d290a202020202a2053656375726520426f6f7420287b7b74
656d706c6174652d7365637572652d626f6f747d7d290a202020202a2046
69726d7761726520446f776e6c6f616420287b7b6669726d776172652d64
6f776e6c6f61642d74656d706c6174657d7d290a202020200a2020202054
686973206578616d706c6520616c736f2064656d6f6e7374726174657320
736576657261626c6520656c656d656e747320287b7b6f76722d73657665
7261626c657d7d292c20616e64207465787420287b7b6d616e6966657374
2d6469676573742d746578747d7d292e814100a2036761726d2e636f6d05
78525468697320636f6d706f6e656e7420697320612064656d6f6e737472
6174696f6e2e205468652064696765737420697320612073616d706c6520
7061747465726e2c206e6f742061207265616c206f6e652e

B.4. Example 3: A/B images

This example covers the following templates:

- * Compatibility Check (Section 7.1)
- * Secure Boot (Section 7.2)
- * Firmware Download (Section 7.3)
- * A/B Image Template (Section 7.7)

```

107({
  / authentication-wrapper / 2:<<[
    digest: <<[
      / algorithm-id / -16 / "sha256" /,
      / digest-bytes /
      h'7c9b3cb72c262608a42f944d59d659ff2b801c78af44def51b8ff51e9f45721b'
    ]>>,
    signature: <<18([
      / protected / <<{
        / alg / 1:-7 / "ES256" /,
      }>>,
      / unprotected / {
      },
      / payload / F6 / nil /,
      / signature / h'e33d618df0ad21e609529ab1a876afb231
      faff1d6a3189b5360324c2794250b87cf00cf83be50ea17dc721ca85393cd8e839a066
      d5dec0ad87a903ab31ea9afa'
    ]>>
  ]
]>>,
/ manifest / 3:<<{
  / manifest-version / 1:1,
  / manifest-sequence-number / 2:3,
  / common / 3:<<{
    / components / 2:[
      [h'00']
    ],
    / common-sequence / 4:<<[
      / directive-override-parameters / 20,{
        / vendor-id /
        1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
        be9d-e663e4d41ffe /,
        / class-id /
        2:h'1492af1425695e48bf429b2d51f2ab45' /
        1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
      } ,
      / directive-try-each / 15,[
        <<[
          / directive-override-parameters / 20,{
            / offset / 5:33792,
          } ,
          / condition-component-offset / 5,5 ,
          / directive-override-parameters / 20,{
            / image-digest / 3:<<[
              / algorithm-id / -16 / "sha256" /,
              / digest-bytes /
              h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
            ]>>,
          ]>>,
        ]>>,
      ]>>,
    ]>>
  }
}

```

```

        / image-size / 14:34768,
    }
  ]>> ,
  <<[
    / directive-override-parameters / 20,{
      / offset / 5:541696,
    } ,
    / condition-component-offset / 5,5 ,
    / directive-override-parameters / 20,{
      / image-digest / 3:<<[
        / algorithm-id / -16 / "sha256" /,
        / digest-bytes /
h'0123456789abcdeffedcba987654321000112233445566778899aabbccddeeff'
      ]>>,
      / image-size / 14:76834,
    }
  ]>>
] ,
/ condition-vendor-identifier / 1,15 ,
/ condition-class-identifier / 2,15
]>>,
}>>,
/ install / 9:<<[
  / directive-try-each / 15,[
    <<[
      / directive-set-parameters / 19,{
        / offset / 5:33792,
      } ,
      / condition-component-offset / 5,5 ,
      / directive-set-parameters / 19,{
        / uri / 21:'http://example.com/file1.bin',
      }
    ]>> ,
    <<[
      / directive-set-parameters / 19,{
        / offset / 5:541696,
      } ,
      / condition-component-offset / 5,5 ,
      / directive-set-parameters / 19,{
        / uri / 21:'http://example.com/file2.bin',
      }
    ]>>
  ] ,
  / directive-fetch / 21,2 ,
  / condition-image-match / 3,15
]>>,
/ validate / 10:<<[
  / condition-image-match / 3,15

```

```

    ]>>,
  }>>,
})

```

Total size of Envelope without COSE authentication object: 332

Envelope:

```

d86ba2025827815824822f58207c9b3cb72c262608a42f944d59d659ff2b
801c78af44def51b8ff51e9f45721b0359011ba5010102030358aaa20281
8141000458a18814a20150fa6b4a53d5ad5fdfe9de663e4d41ffe025014
92af1425695e48bf429b2d51f2ab450f8258368614a105198400050514a2
035824822f582000112233445566778899aabbccddeeff0123456789abcd
effedcba98765432100e1987d0583a8614a1051a00084400050514a20358
24822f58200123456789abcdeffedcba9876543210001122334455667788
99aabbccddeeff0e1a00012c22010f020f095861860f82582a8613a10519
8400050513a115781c687474703a2f2f6578616d706c652e636f6d2f6669
6c65312e62696e582c8613a1051a00084400050513a115781c687474703a
2f2f6578616d706c652e636f6d2f66696c65322e62696e1502030f0a4382
030f

```

Total size of Envelope with COSE authentication object: 408

Envelope with COSE authentication object:

```

d86ba2025873825824822f58207c9b3cb72c262608a42f944d59d659ff2b
801c78af44def51b8ff51e9f45721b584ad28443a10126a0f65840e33d61
8df0ad21e609529ab1a876afb231faff1d6a3189b5360324c2794250b87c
f00cf83be50ea17dc721ca85393cd8e839a066d5dec0ad87a903ab31ea9a
fa0359011ba5010102030358aaa202818141000458a18814a20150fa6b4a
53d5ad5fdfe9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab
450f8258368614a105198400050514a2035824822f582000112233445566
778899aabbccddeeff0123456789abcdeffedcba98765432100e1987d058
3a8614a1051a00084400050514a2035824822f58200123456789abcdeffe
dcba987654321000112233445566778899aabbccddeeff0e1a00012c2201
0f020f095861860f82582a8613a105198400050513a115781c687474703a
2f2f6578616d706c652e636f6d2f66696c65312e62696e582c8613a1051a
00084400050513a115781c687474703a2f2f6578616d706c652e636f6d2f
66696c65322e62696e1502030f0a4382030f

```

B.5. Example 4: Load from External Storage

This example covers the following templates:

- * Compatibility Check (Section 7.1)
- * Secure Boot (Section 7.2)

* Firmware Download (Section 7.3)

* Install (Section 7.4)

* Load (Section 7.6)

```

107({
  / authentication-wrapper / 2:<<[
    digest: <<[
      / algorithm-id / -16 / "sha256" /,
      / digest-bytes /
h'15736702a00f510805dcf89d6913a2cfb417ed414faa760f974d6755c68ba70a'
    ]>>,
    signature: <<18([
      / protected / <<{
        / alg / 1:-7 / "ES256" /,
      }>>,
      / unprotected / {
      },
      / payload / F6 / nil /,
      / signature / h'3ada2532326d512132c388677798c24ffd
cc979bfae2a26b19c8c8bbf511fd7dd85f1501662c1a9e1976b759c4019bab44ba5434
efb45d3868aedbca593671f3'
    ]>>
  ]
]>>,
  / manifest / 3:<<{
    / manifest-version / 1:1,
    / manifest-sequence-number / 2:4,
    / common / 3:<<{
      / components / 2:[
        [h'00'] ,
        [h'02'] ,
        [h'01']
      ],
      / common-sequence / 4:<<[
        / directive-set-component-index / 12,0 ,
        / directive-override-parameters / 20,{
          / vendor-id /
1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
          / class-id /
2:h'1492af1425695e48bf429b2d51f2ab45' /
1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
          / image-digest / 3:<<[
            / algorithm-id / -16 / "sha256" /,
            / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'

```

```

        ]>>,
        / image-size / 14:34768,
    } ,
    / condition-vendor-identifier / 1,15 ,
    / condition-class-identifier / 2,15
  ]>>,
}>>,
/ payload-fetch / 8:<<[
  / directive-set-component-index / 12,1 ,
  / directive-set-parameters / 19,{
    / uri / 21:'http://example.com/file.bin' ,
  } ,
  / directive-fetch / 21,2 ,
  / condition-image-match / 3,15
]>>,
/ install / 9:<<[
  / directive-set-component-index / 12,0 ,
  / directive-set-parameters / 19,{
    / source-component / 22:1 / [h'02'] / ,
  } ,
  / directive-copy / 22,2 ,
  / condition-image-match / 3,15
]>>,
/ validate / 10:<<[
  / directive-set-component-index / 12,0 ,
  / condition-image-match / 3,15
]>>,
/ load / 11:<<[
  / directive-set-component-index / 12,2 ,
  / directive-set-parameters / 19,{
    / image-digest / 3:<<[
      / algorithm-id / -16 / "sha256" / ,
      / digest-bytes /
h'0123456789abcdeffedcba987654321000112233445566778899aabbccddeeff'
    ]>>,
    / image-size / 14:76834,
    / source-component / 22:0 / [h'00'] / ,
    / compression-info / 19:<<[
      / compression-algorithm / 1:1 / "gzip" / ,
    ]>>,
  } ,
  / directive-copy / 22,2 ,
  / condition-image-match / 3,15
]>>,
/ run / 12:<<[
  / directive-set-component-index / 12,2 ,
  / directive-run / 23,2
]>>,

```

```
    }>>,
  })
```

Total size of Envelope without COSE authentication object: 292

Envelope:

```
d86ba2025827815824822f582015736702a00f510805dcf89d6913a2cfb4
17ed414faa760f974d6755c68ba70a0358f4a801010204035867a2028381
4100814102814101045858880c0014a40150fa6b4a53d5ad5fdfbe9de663
e4d41ffe02501492af1425695e48bf429b2d51f2ab45035824822f582000
112233445566778899aabbccddeeff0123456789abcdeffedcba98765432
100e1987d0010f020f085827880c0113a115781b687474703a2f2f657861
6d706c652e636f6d2f666696c652e62696e1502030f094b880c0013a11601
1602030f0a45840c00030f0b583d880c0213a4035824822f582001234567
89abcdeffedcba987654321000112233445566778899aabbccddeeff0e1a
00012c221343a1010116001602030f0c45840c021702
```

Total size of Envelope with COSE authentication object: 368

Envelope with COSE authentication object:

```
d86ba2025873825824822f582015736702a00f510805dcf89d6913a2cfb4
17ed414faa760f974d6755c68ba70a584ad28443a10126a0f658403ada25
32326d512132c388677798c24ffdcc979bfae2a26b19c8c8bbf511fd7dd8
5f1501662c1a9e1976b759c4019bab44ba5434efb45d3868aedbca593671
f30358f4a801010204035867a20283814100814102814101045858880c00
14a40150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492af1425695e48
bf429b2d51f2ab45035824822f582000112233445566778899aabbccdde
ff0123456789abcdeffedcba98765432100e1987d0010f020f085827880c
0113a115781b687474703a2f2f6578616d706c652e636f6d2f666696c652e
62696e1502030f094b880c0013a116011602030f0a45840c00030f0b583d
880c0213a4035824822f58200123456789abcdeffedcba98765432100011
2233445566778899aabbccddeeff0e1a00012c221343a101011600160203
0f0c45840c021702
```

B.6. Example 5: Two Images

This example covers the following templates:

- * Compatibility Check (Section 7.1)
- * Secure Boot (Section 7.2)
- * Firmware Download (Section 7.3)

Furthermore, it shows using these templates with two images.

```

107({
  / authentication-wrapper / 2:<<[
    digest: <<[
      / algorithm-id / -16 / "sha256" /,
      / digest-bytes /
h'dle73f16e4126007bc4d804cd33b0209fbab34728e60ee8c00f3387126748dd2'
    ]>>,
    signature: <<18([
      / protected / <<{
        / alg / 1:-7 / "ES256" /,
      }>>,
      / unprotected / {
        },
      / payload / F6 / nil /,
      / signature / h'b7ae0a46a28f02e25cda6d9a255bbaf863
30141831fae5a78012d648bc6cee55102e0f1890bdeacc3adaa4fae0560f83a45eeca
65cabce642f56d84ab97ef8d'
    ]>>
  ]
]>>,
/ manifest / 3:<<{
  / manifest-version / 1:1,
  / manifest-sequence-number / 2:5,
  / common / 3:<<{
    / components / 2:[
      [h'00'] ,
      [h'01']
    ],
    / common-sequence / 4:<<[
      / directive-set-component-index / 12,0 ,
      / directive-override-parameters / 20,{
        / vendor-id /
1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
        / class-id /
2:h'1492af1425695e48bf429b2d51f2ab45' /
1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
        / image-digest / 3:<<[
          / algorithm-id / -16 / "sha256" /,
          / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
        ]>>,
        / image-size / 14:34768,
      } ,
      / condition-vendor-identifier / 1,15 ,
      / condition-class-identifier / 2,15 ,
      / directive-set-component-index / 12,1 ,
      / directive-override-parameters / 20,{

```



```

        / image-digest / 3:<<[
          / algorithm-id / -16 / "sha256" /,
          / digest-bytes /
h'0123456789abcdeffedcba987654321000112233445566778899aabbccddeeff'
        ]>>,
        / image-size / 14:76834,
      }
    ]>>,
  }>>,
/ install / 9:<<[
  / directive-set-component-index / 12,0 ,
  / directive-set-parameters / 19,{
    / uri / 21:'http://example.com/file1.bin',
  } ,
  / directive-fetch / 21,2 ,
  / condition-image-match / 3,15 ,
  / directive-set-component-index / 12,1 ,
  / directive-set-parameters / 19,{
    / uri / 21:'http://example.com/file2.bin',
  } ,
  / directive-fetch / 21,2 ,
  / condition-image-match / 3,15
]>>,
/ validate / 10:<<[
  / directive-set-component-index / 12,0 ,
  / condition-image-match / 3,15 ,
  / directive-set-component-index / 12,1 ,
  / condition-image-match / 3,15
]>>,
/ run / 12:<<[
  / directive-set-component-index / 12,0 ,
  / directive-run / 23,2
]>>,
} >>,
))

```

Total size of Envelope without COSE authentication object: 306

Envelope:

```
d86ba2025827815824822f5820dle73f16e4126007bc4d804cd33b0209fb
ab34728e60ee8c00f3387126748dd203590101a601010205035895a20282
8141008141010458898c0c0014a40150fa6b4a53d5ad5fdfbe9de663e4d4
1ffe02501492af1425695e48bf429b2d51f2ab45035824822f5820001122
33445566778899aabbccddeeff0123456789abcdeffedcba98765432100e
1987d0010f020f0c0114a2035824822f58200123456789abcdeffedcba98
7654321000112233445566778899aabbccddeeff0e1a00012c2209584f90
0c0013a115781c687474703a2f2f6578616d706c652e636f6d2f66696c65
312e62696e1502030f0c0113a115781c687474703a2f2f6578616d706c65
2e636f6d2f66696c65322e62696e1502030f0a49880c00030f0c01030f0c
45840c001702
```

Total size of Envelope with COSE authentication object: 382

Envelope with COSE authentication object:

```
d86ba2025873825824822f5820dle73f16e4126007bc4d804cd33b0209fb
ab34728e60ee8c00f3387126748dd2584ad28443a10126a0f65840b7ae0a
46a28f02e25cda6d9a255bbaf86330141831fae5a78012d648bc6cee5510
2e0f1890bdeacc3adaa4fae0560f83a45eecae65cabce642f56d84ab97ef
8d03590101a601010205035895a202828141008141010458898c0c0014a4
0150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf42
9b2d51f2ab45035824822f582000112233445566778899aabbccddeeff01
23456789abcdeffedcba98765432100e1987d0010f020f0c0114a2035824
822f58200123456789abcdeffedcba987654321000112233445566778899
aabbccddeeff0e1a00012c2209584f900c0013a115781c687474703a2f2f
6578616d706c652e636f6d2f66696c65312e62696e1502030f0c0113a115
781c687474703a2f2f6578616d706c652e636f6d2f66696c65322e62696e
1502030f0a49880c00030f0c01030f0c45840c001702
```

Appendix C. C. Design Rational

In order to provide flexible behavior to constrained devices, while still allowing more powerful devices to use their full capabilities, the SUIT manifest encodes the required behavior of a Recipient device. Behavior is encoded as a specialized byte code, contained in a CBOR list. This promotes a flat encoding, which simplifies the parser. The information encoded by this byte code closely matches the operations that a device will perform, which promotes ease of processing. The core operations used by most update and trusted invocation operations are represented in the byte code. The byte code can be extended by registering new operations.

The specialized byte code approach gives benefits equivalent to those provided by a scripting language or conventional byte code, with two substantial differences. First, the language is extremely high level, consisting of only the operations that a device may perform during update and trusted invocation of a firmware image. Second,

the language specifies linear behavior, without reverse branches. Conditional processing is supported, and parallel and out-of-order processing may be performed by sufficiently capable devices.

By structuring the data in this way, the manifest processor becomes a very simple engine that uses a pull parser to interpret the manifest. This pull parser invokes a series of command handlers that evaluate a Condition or execute a Directive. Most data is structured in a highly regular pattern, which simplifies the parser.

The results of this allow a Recipient to implement a very small parser for constrained applications. If needed, such a parser also allows the Recipient to perform complex updates with reduced overhead. Conditional execution of commands allows a simple device to perform important decisions at validation-time.

Dependency handling is vastly simplified as well. Dependencies function like subroutines of the language. When a manifest has a dependency, it can invoke that dependency's commands and modify their behavior by setting parameters. Because some parameters come with security implications, the dependencies also have a mechanism to reject modifications to parameters on a fine-grained level.

Developing a robust permissions system works in this model too. The Recipient can use a simple ACL that is a table of Identities and Component Identifier permissions to ensure that operations on components fail unless they are permitted by the ACL. This table can be further refined with individual parameters and commands.

Capability reporting is similarly simplified. A Recipient can report the Commands, Parameters, Algorithms, and Component Identifiers that it supports. This is sufficiently precise for a manifest author to create a manifest that the Recipient can accept.

The simplicity of design in the Recipient due to all of these benefits allows even a highly constrained platform to use advanced update capabilities.

C.1. C.1 Design Rationale: Envelope

The Envelope is used instead of a COSE structure for several reasons:

1. This enables the use of Severable Elements (Section 8.5)
2. This enables modular processing of manifests, particularly with large signatures.
3. This enables multiple authentication schemes.

4. This allows integrity verification by a dependent to be unaffected by adding or removing authentication structures.

Modular processing is important because it allows a Manifest Processor to iterate forward over an Envelope, processing Delegation Chains and Authentication Blocks, retaining only intermediate values, without any need to seek forward and backwards in a stream until it gets to the Manifest itself. This allows the use of large, Post-Quantum signatures without requiring retention of the signature itself, or seeking forward and back.

Four authentication objects are supported by the Envelope:

- * COSE_Sign_Tagged
- * COSE_Sign1_Tagged
- * COSE_Mac_Tagged
- * COSE_Mac0_Tagged

The SUIF Envelope allows an Update Authority or intermediary to mix and match any number of different authentication blocks it wants without any concern for modifying the integrity of another authentication block. This also allows the addition or removal of an authentication blocks without changing the integrity check of the Manifest, which is important for dependency handling. See Section 6.2

C.2. C.2 Byte String Wrappers

Byte string wrappers are used in several places in the suit manifest. The primary reason for wrappers is to limit the parser extent when invoked at different times, with a possible loss of context.

The elements of the suit envelope are wrapped both to set the extents used by the parser and to simplify integrity checks by clearly defining the length of each element.

The common block is re-parsed in order to find components identifiers from their indices, to find dependency prefixes and digests from their identifiers, and to find the common sequence. The common sequence is wrapped so that it matches other sequences, simplifying the code path.

A severed SUIF command sequence will appear in the envelope, so it must be wrapped as with all envelope elements. For consistency, command sequences are also wrapped in the manifest. This also allows the parser to discern the difference between a command sequence and a SUIF_Digest.

Parameters that are structured types (arrays and maps) are also wrapped in a bstr. This is so that parser extents can be set correctly using only a reference to the beginning of the parameter. This enables a parser to store a simple list of references to parameters that can be retrieved when needed.

Appendix D. D. Implementation Conformance Matrix

This section summarizes the functionality a minimal manifest processor implementation needs to offer to claim conformance to this specification, in the absence of an application profile standard specifying otherwise.

The subsequent table shows the conditions.

| Name | Reference | Implementation |
|-------------------|-----------------|----------------|
| Vendor Identifier | Section 8.4.8.2 | REQUIRED |
| Class Identifier | Section 8.4.8.2 | REQUIRED |
| Device Identifier | Section 8.4.8.2 | OPTIONAL |
| Image Match | Section 8.4.9.2 | REQUIRED |
| Component Slot | Section 8.4.9.3 | OPTIONAL |
| Abort | Section 8.4.9.4 | OPTIONAL |
| Custom Condition | Section 8.4.9.5 | OPTIONAL |

Table 14

The subsequent table shows the directives.

| Name | Reference | Implementation |
|---------------------|------------------|-------------------------------------|
| Set Component Index | Section 8.4.10.1 | REQUIRED if more than one component |
| Try Each | Section 8.4.10.2 | OPTIONAL |
| Override Parameters | Section 8.4.10.3 | REQUIRED |
| Fetch | Section 8.4.10.4 | REQUIRED for Updater |
| Copy | Section 8.4.10.5 | OPTIONAL |
| Run | Section 8.4.10.6 | REQUIRED for Bootloader |
| Run Sequence | Section 8.4.10.7 | OPTIONAL |
| Swap | Section 8.4.10.8 | OPTIONAL |

Table 15

The subsequent table shows the parameters.

| Name | Reference | Implementation |
|------------------|------------------|----------------------|
| Vendor ID | Section 8.4.8.3 | REQUIRED |
| Class ID | Section 8.4.8.4 | REQUIRED |
| Image Digest | Section 8.4.8.6 | REQUIRED |
| Image Size | Section 8.4.8.7 | REQUIRED |
| Component Slot | Section 8.4.8.8 | OPTIONAL |
| URI | Section 8.4.8.9 | REQUIRED for Updater |
| Source Component | Section 8.4.8.10 | OPTIONAL |
| Run Args | Section 8.4.8.11 | OPTIONAL |
| Device ID | Section 8.4.8.5 | OPTIONAL |
| Strict Order | Section 8.4.8.13 | OPTIONAL |
| Soft Failure | Section 8.4.8.14 | OPTIONAL |
| Custom | Section 8.4.8.15 | OPTIONAL |

Table 16

Authors' Addresses

Brendan Moran
 Arm Limited
 Email: Brendan.Moran@arm.com

Hannes Tschofenig
 Arm Limited
 Email: hannes.tschofenig@arm.com

Henk Birkholz
 Fraunhofer SIT
 Email: henk.birkholz@sit.fraunhofer.de

Koen Zandberg
Inria
Email: koen.zandberg@inria.fr