

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.

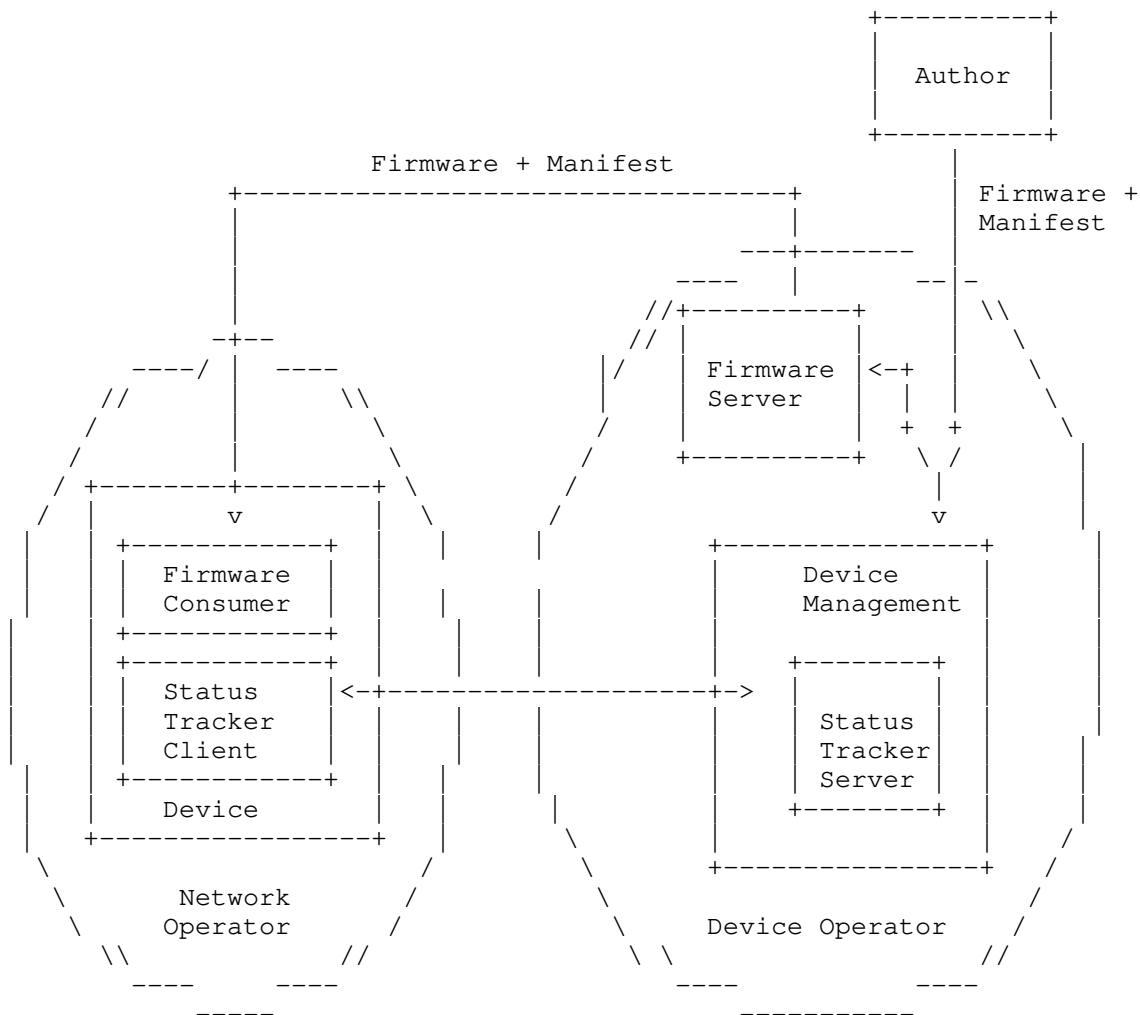


Figure 1: Architecture.

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].

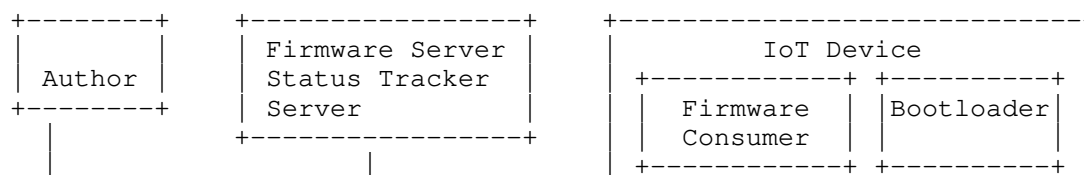
When a device obtains a monolithic firmware image from a single author without any additional approval steps, the authorization flow is relatively simple. There are, however, other cases where more complex policy decisions need to be made before updating a device.

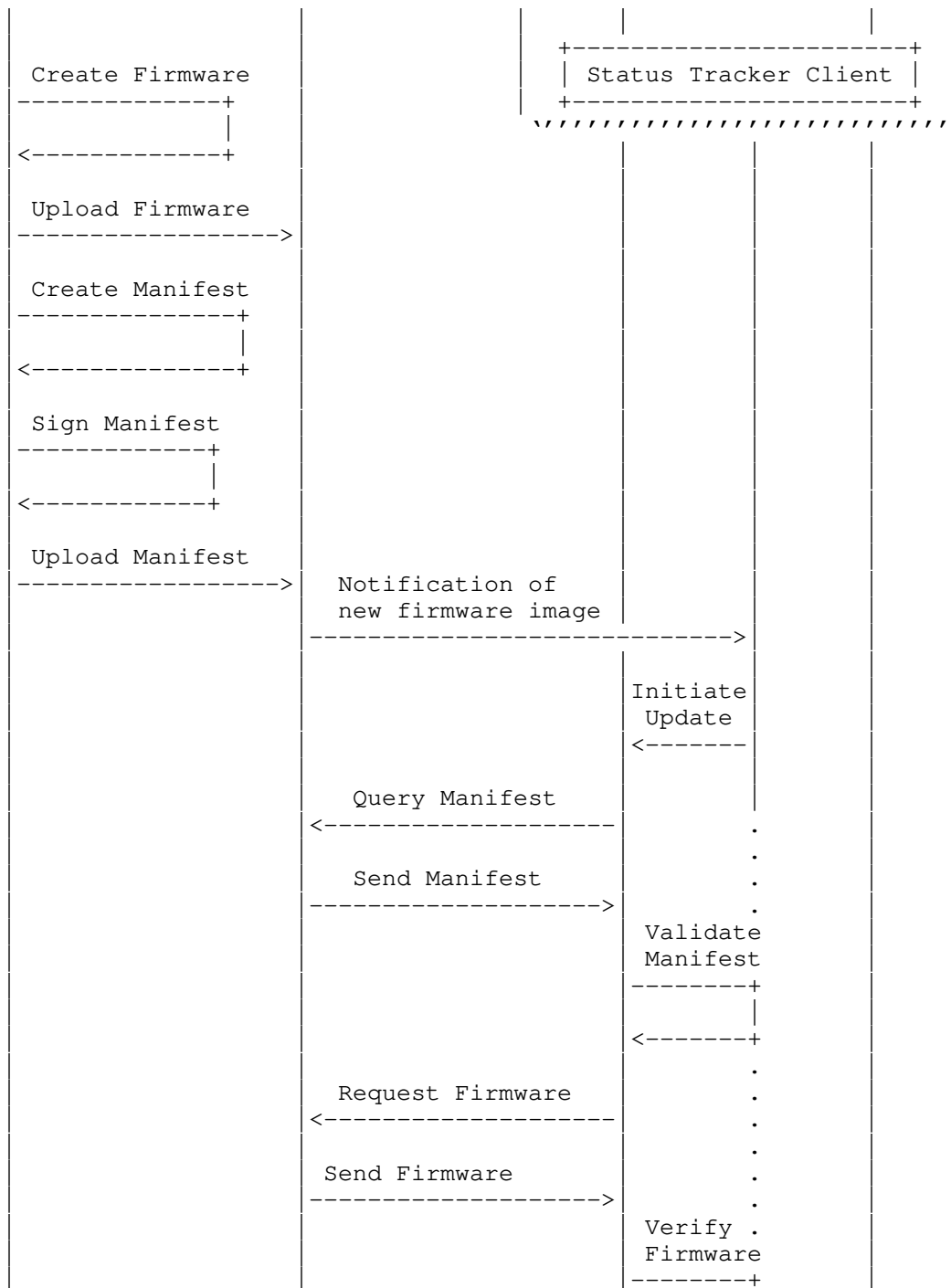
In this architecture the authorization policy is separated from the underlying communication architecture. This is accomplished by separating the entities from their permissions. For example, an author may not have the authority to install a firmware image on a device in critical infrastructure without 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.

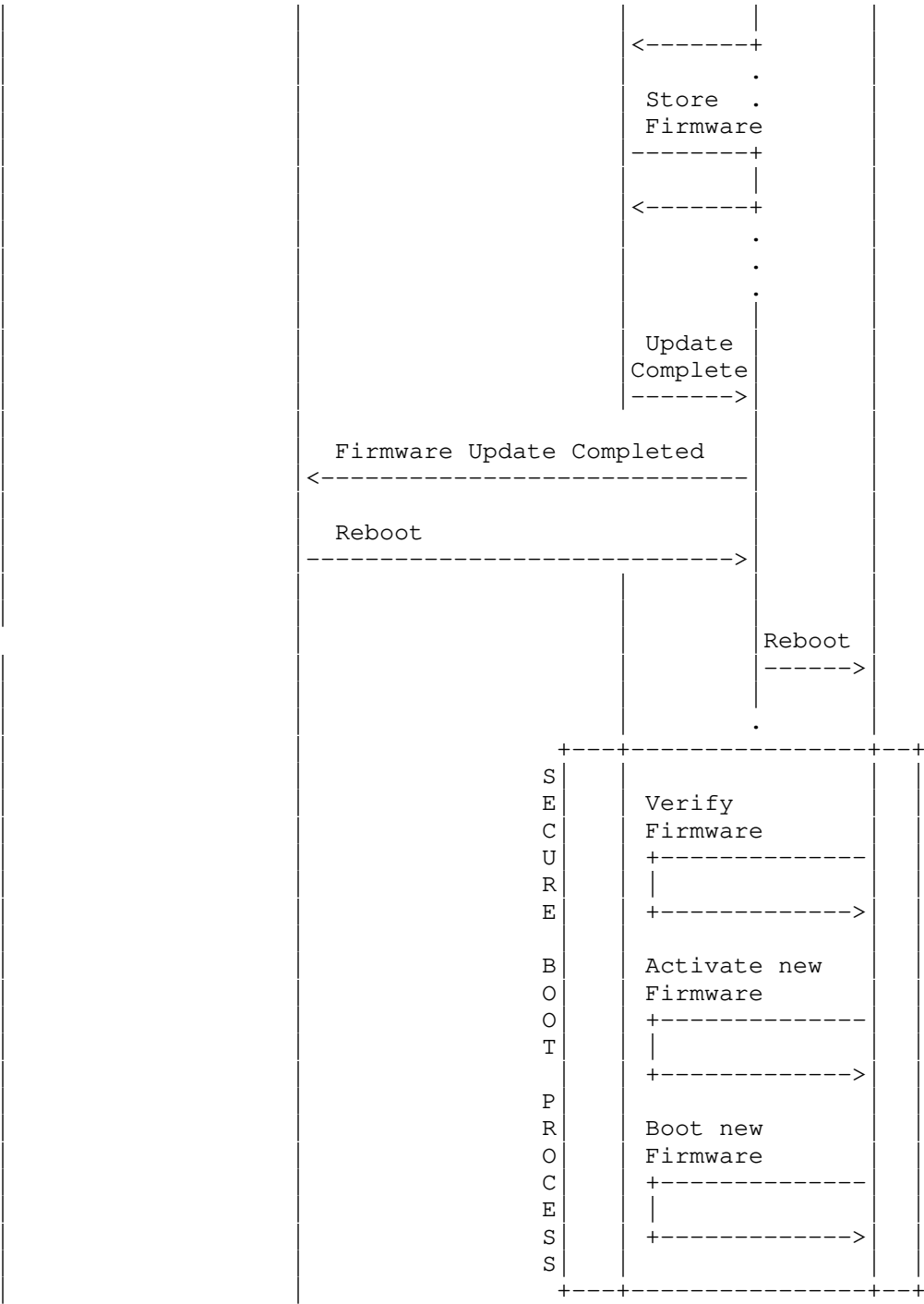
Alternatively, a device may trust precisely one entity, which does all permission management and coordination. This entity allows the device to offload complex permissions calculations for the device.

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.







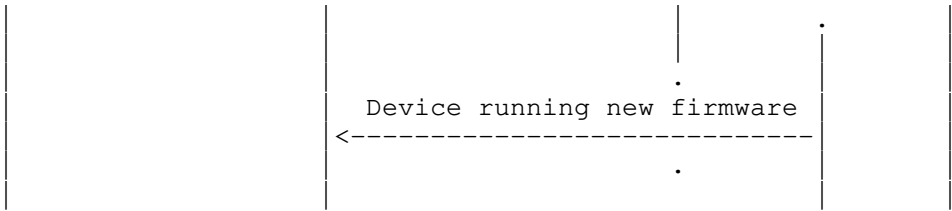
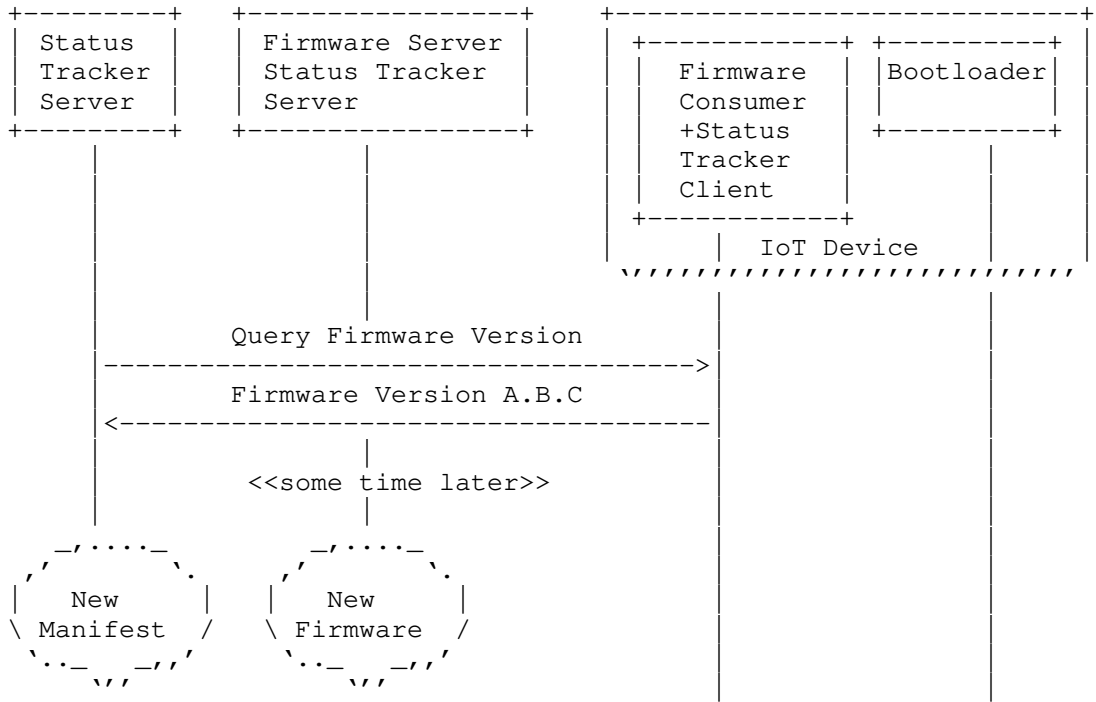


Figure 2: First Example Flow for a Firmware Update.

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.



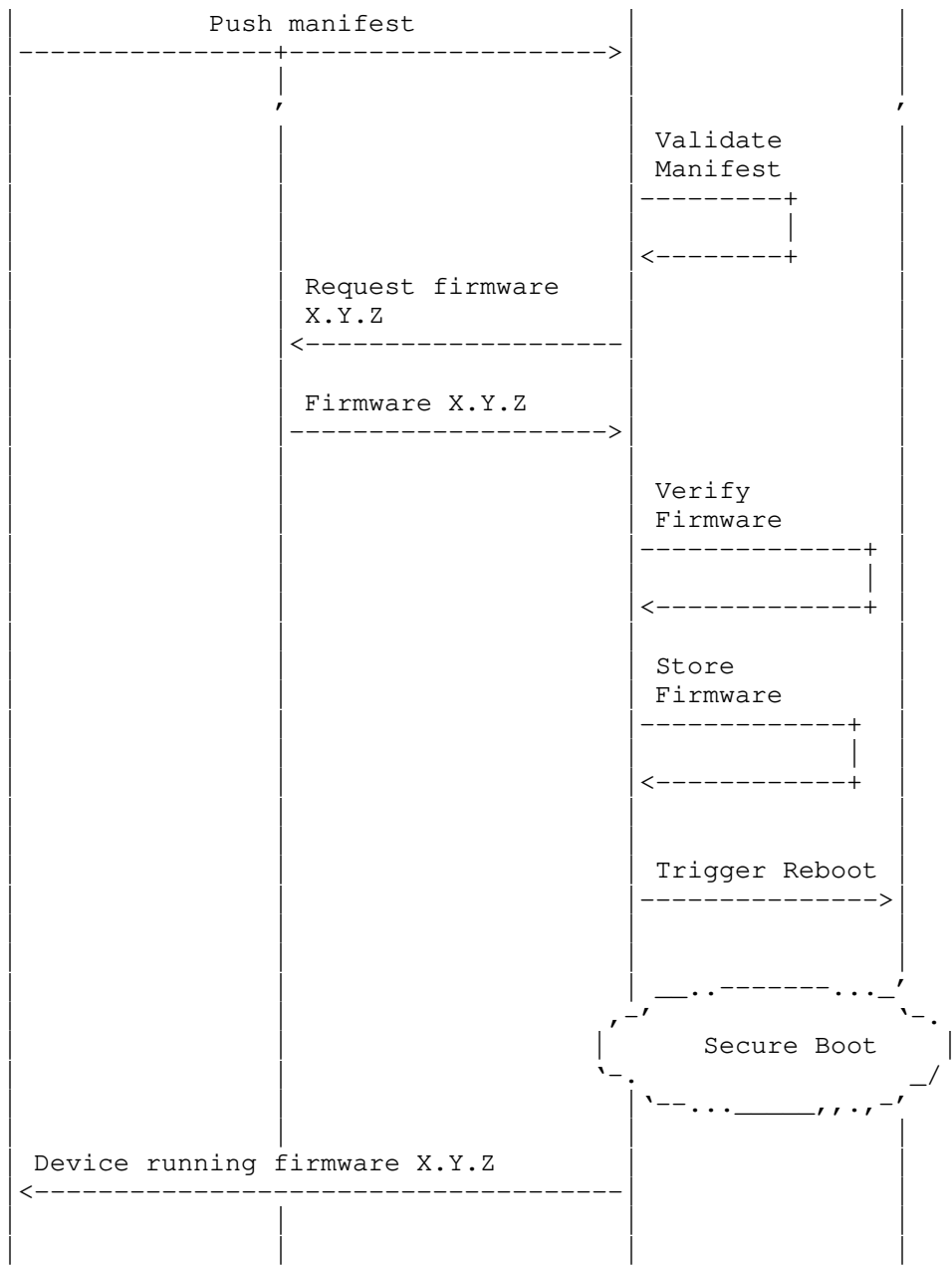


Figure 3: Second Example Flow for a Firmware Update.

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: Informational
Expires: January 9, 2020

B. Moran
H. Tschofenig
Arm Limited
H. Birkholz
Fraunhofer SIT
July 08, 2019

SUIT CBOR manifest serialisation format
draft-moran-suit-manifest-05

Abstract

This specification describes the format of a manifest. A manifest is a bundle of metadata about the firmware for an IoT device, where to find the firmware, the devices to which it applies, and cryptographic information protecting 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, 2020.

Copyright Notice

Copyright (c) 2019 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.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Introduction	3
2. Conventions and Terminology	4
3. Distributing firmware	5
4. Workflow of a device applying a firmware update	5
5. SUIT manifest goals	6
6. SUIT manifest design overview	7
6.1. Manifest Design Evaluation	8
6.2. Severable Elements	9
6.3. Conventions	9
6.4. Payloads	9
7. Manifest Structure	10
7.1. Outer wrapper	11
7.2. Manifest	13
7.3. SUIT_Dependency	16
7.4. SUIT_Component_Reference	17
7.5. Manifest Parameters	17
7.5.1. SUIT_Parameter_Strict_Order	19
7.5.2. SUIT_Parameter_Coerce_Condition_Failure	20
7.6. SUIT_Parameter_Encryption_Info	20
7.7. SUIT_Parameter_Compression_Info	20
7.8. SUIT_Parameter_Unpack_Info	20
7.9. SUIT_Parameters CDDL	21
7.10. SUIT_Command_Sequence	22
7.11. SUIT_Condition	24
7.11.1. Identifier Conditions	25
7.11.2. suit-condition-image-match	25
7.11.3. suit-condition-image-not-match	25
7.11.4. suit-condition-use-before	25
7.11.5. suit-condition-minimum-battery	25
7.11.6. suit-condition-update-authorised	26
7.11.7. suit-condition-version	26

7.11.8.	SUIT_Condition_Custom	27
7.11.9.	Identifiers	27
7.11.10.	SUIT_Condition CDDL	29
7.12.	SUIT_Directive	29
7.12.1.	suit-directive-set-component-index	30
7.12.2.	suit-directive-set-dependency-index	31
7.12.3.	suit-directive-abort	31
7.12.4.	suit-directive-run-sequence	31
7.12.5.	suit-directive-try-each	32
7.12.6.	suit-directive-process-dependency	32
7.12.7.	suit-directive-set-parameters	33
7.12.8.	suit-directive-override-parameters	33
7.12.9.	suit-directive-fetch	34
7.12.10.	suit-directive-copy	34
7.12.11.	suit-directive-swap	35
7.12.12.	suit-directive-run	35
7.12.13.	suit-directive-wait	36
7.12.14.	SUIT_Directive CDDL	37
8.	Dependency processing	39
9.	Access Control Lists	40
10.	SUIT digest container	40
11.	Creating conditional sequences	41
12.	Full CDDL	43
13.	Examples	48
13.1.	Example 0:	48
13.2.	Example 1:	49
13.3.	Example 2:	52
13.4.	Example 3:	54
13.5.	Example 4:	57
13.6.	Example 5:	61
13.7.	Example 6:	65
14.	IANA Considerations	68
15.	Security Considerations	68
16.	Mailing List Information	69
17.	Acknowledgements	69
18.	References	69
18.1.	Normative References	69
18.2.	Informative References	70
18.3.	URIs	70
	Authors' Addresses	71

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, such as the Lightweight Machine-to-Machine (LwM2M) protocol offering device management of IoT

devices. Equally important is the inclusion of meta-data about the conveyed firmware image (in the form of a manifest) and the use of 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. This authorization process is ensured by the use of dedicated symmetric or asymmetric keys installed on the IoT device: for use cases where only integrity protection is required it is sufficient to install a trust anchor on the IoT device. For confidentiality protected firmware images it is additionally required to install either one or multiple symmetric or asymmetric keys on the IoT device. 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.

It is assumed that the reader is familiar with the high-level firmware update architecture [Architecture].

The SUIT manifest is heavily optimised for consumption by constrained devices. This means that it is not constructed as a conventional descriptive document. Instead, of describing what an update IS, it describes what a recipient should DO.

While the SUIT manifest is informed by and optimised for firmware update use cases, there is nothing in the [Information] that restricts its use to only firmware use cases. Software update and delivery of arbitrary data can equally be managed by SUIT-based metadata.

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.

- SUIT: Software Update for the Internet of Things, 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 piece of information that describes one or more payloads, one or more resources, and the processors needed to transform resources into payloads.
- 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 recipient devices.
- Recipient: The system, typically an IoT device, that receives a manifest.
- Condition: A test for a property of the Recipient or its components.
- Directive: An action for the Recipient to perform.
- Command: A Condition or a Directive.
- Trusted Execution: A process by which a system ensures that only trusted code is executed, for example secure boot.

3. Distributing firmware

Distributing firmware in a multi-party environment is a difficult operation. 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 [Architecture].

4. Workflow of a device applying a firmware update

The manifest is designed to work with a pull parser, where each section of the manifest is used in sequence. The expected workflow for a device installing an update can be broken down into 5 steps:

1. Verify the signature of the manifest
2. Verify the applicability of the manifest
3. Resolve dependencies
4. Fetch payload(s)
5. 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)

When multiple manifests are used for an update, each manifest's steps occur in a lockstep fashion; all manifests have dependency resolution performed before any manifest performs a payload fetch, etc.

5. SUIT manifest goals

The manifest described in this document is intended to meet several goals, as described below.

1. Meet the requirements defined in [Information].
2. Simple to parse on a constrained node
3. Simple to process on a constrained node
4. Compact encoding
5. Comprehensible by an intermediate system
6. Expressive enough to enable advanced use cases on advanced nodes
7. Extensible

The SUIT manifest can be used for a variety of purposes throughout its lifecycle. The manifest allows:

1. the Firmware Author to reason about releasing a firmware.
2. the Network Operator to reason about compatibility of a firmware.
3. the Device Operator to reason about the impact of a firmware.
4. the Device Operator to manage distribution of firmware to devices.
5. the Plant Manager to reason about timing and acceptance of firmware updates.
6. the device to reason about the authority & authenticity of a firmware prior to installation.
7. the device to reason about the applicability of a firmware.

8. the device to reason about the installation of a firmware.
9. the 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.

6. SUIT manifest design overview

In order to provide flexible behaviour to constrained devices, while still allowing more powerful devices to use their full capabilities, the SUIT manifest encodes the required behaviour of a Recipient device. Behaviour is encoded as a specialised 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 execution operations are represented in the byte code. The byte code can be extended by registering new operations.

The specialised 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 execution of a firmware image. Second, the language specifies behaviours in a linearised form, 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 behaviour 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 only manifests authenticated by the appropriate identity have access to operate on a component.

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.

6.1. Manifest Design Evaluation

To evaluate this design, it is compared to the goals stated above.

Goal evaluation:

1. Each command and condition is anchored to a manifest information element in [Information]
2. The use of a byte code encourages flat encoding and reduces nesting depth. This promotes a simple encoding.
3. The encoded information closely matches the operations that a device will perform, making the format easy to process.
4. Encoding efficiency exceeds 50% when compared to raw data.
5. Tooling will be required to reason about the manifest.
6. The core operations used by most update and trusted execution operations are represented in the byte code. The use cases listed in [Information] are enabled.
7. Registration of new standard byte code identifiers enables extension in a comprehensible way.

The manifest described by this document meets the stated goals. Meeting goal 5-comprehensible by intermediate systems-will require additional tooling or a division of metadata.

6.2. Severable Elements

Because the manifest can be used by different actors at different times, some parts of the manifest can be removed without affecting later stages of the lifecycle. This is called "Severing." 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 authenticity 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 SUIT_Digest of the bstr in the manifest so that they can still be authenticated. The SUIT_Digest typically consumes 4 bytes more than the size of the raw digest, therefore elements smaller than $(\text{Digest Bits})/8 + 4$ SHOULD never 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.

6.3. Conventions

The map indices in this 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 uninitialised variables. Positive numbers in enumerations are reserved for IANA registration. Negative numbers are used to identify application-specific implementations.

CDDL names are hyphenated and CDDL structures follow the convention adopted in COSE [RFC8152]: SUIT_Structure_Name.

6.4. Payloads

Payloads can take many forms, for example, binary, hex, s-record, elf, binary diff, PEM certificate, CBOR Web Token, serialised configuration. These payloads fall into two broad categories: those that require installation-time unpacking and those that do not. Binary, PEM certificate, and CBOR Web Token do not require installation-time unpacking. Hex, s-record, and serialised

configuration require installation-time unpacking. Elf may or may not require unpacking depending on the target.

Some payloads cannot be directly converted to a writable binary stream. Hex, s-record, and elf may contain gaps and they have no guarantee of monotonic increase of address, which makes pre-processing them into a binary stream difficult on constrained platforms. Serialised configuration may be unpacked into a configuration database, which makes it impossible to preprocess into a binary stream, suitable for direct writing.

Where a specialised unpacking algorithm is needed, a digest is not always calculable over an installed payload. For example, an elf, s-record or hex file may contain gaps that can contain any data, while not changing whether or not an installed payload is valid. Serialised configuration may update only some device data rather than all of it. This means that the digest cannot always be calculated over an installed payload when a specialised installer is used.

This presents two problems for the manifest: first, it must indicate that a specialised installer is needed and, second, it cannot provide a hash of the payload that is checkable after installation. These two problems are resolved in two ways:

1. Payloads that need a specialised installer must indicate this in `suit-payload-info-unpack`.
2. Payloads that need specialised verification must indicate this in the `SUIT_Parameter_Image_Digest` by indicating a `SUIT_Digest` algorithm that correctly validates their information.

7. Manifest Structure

The manifest is divided into several sections in a hierarchy as follows:

1. The outer wrapper
 1. The authentication wrapper
 2. The manifest
 1. Critical Information
 2. Information shared by all command sequences
 1. List of dependencies

2. List of payloads
3. List of payloads in dependencies
4. Common list of conditions, directives
3. Dependency resolution Reference or list of conditions, directives
4. Payload fetch Reference or list of conditions, directives
5. Installation Reference or list of conditions, directives
6. Verification conditions/directives
7. Load conditions/directives
8. Run conditions/directives
9. Text / Reference
10. COSWID / Reference
3. Dependency resolution conditions/directives
4. Payload fetch conditions/directives
5. Installation conditions/directives
6. Text
7. COSWID / Reference
8. Intermediate Certificate(s) / CWTs
9. Inline Payload(s)

7.1. Outer wrapper

This object is a container for the other pieces of the manifest to provide a common mechanism to find each of the parts. All elements of the outer wrapper are contained in bstr objects. Wherever the manifest references an object in the outer wrapper, the bstr is included in the digest calculation.

The CDDL that describes the wrapper is below

```

SUIT_Outer_Wrapper = {
    suit-authentication-wrapper => bstr .cbor
                                SUIT_Authentication_Wrapper / nil,
    $SUIT_Manifest_Wrapped,
    ? suit-dependency-resolution => bstr .cbor SUIT_Command_Sequence,
    ? suit-payload-fetch         => bstr .cbor SUIT_Command_Sequence,
    ? suit-install               => bstr .cbor SUIT_Command_Sequence,
    ? suit-text-external         => bstr .cbor SUIT_Text_Info,
    ? suit-coswid-external       => bstr .cbor COSWID
}

SUIT_Authentication_Wrapper = [ + (COSE_Mac_Tagged / COSE_Sign_Tagged /
                                COSE_Mac0_Tagged / COSE_Sign1_Tagged)]
SUIT_Encryption_Wrapper = COSE_Encrypt_Tagged / COSE_Encrypt0_Tagged

SUIT_Manifest_Wrapped //= (suit-manifest => bstr .cbor SUIT_Manifest)
SUIT_Manifest_Wrapped //= (
    suit-manifest-encryption-info => bstr .cbor SUIT_Encryption_Wrapper,
    suit-manifest-encrypted       => bstr
)

```

All elements of the outer wrapper 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 serialisation for integrity and authenticity checks.

The suit-authentication-wrapper contains a list of 1 or more cryptographic authentication wrappers for the core part of the manifest. These are implemented as COSE_Mac_Tagged or COSE_Sign_Tagged blocks. The Manifest is authenticated by these blocks in "detached payload" mode. The COSE_Mac_Tagged and COSE_Sign_Tagged blocks are described in RFC 8152 [RFC8152] and are beyond the scope of this document. The suit-authentication-wrapper MUST come first in the SUIT_Outer_Wrapper, regardless of canonical encoding of CBOR. All validators MUST reject any SUIT_Outer_Wrapper that begins with any element other than a suit-authentication-wrapper.

A manifest that has not had authentication information added MUST still contain the suit-authentication-wrapper element, but the content MUST be nil.

The outer wrapper MUST contain only one of

- a plaintext manifest: SUIT_Manifest
- an encrypted manifest: both a SUIT_Encryption_Wrapper and the ciphertext of a manifest.

When the outer wrapper contains `SUIT_Encryption_Wrapper`, the `suit-authentication-wrapper` MUST authenticate the plaintext of `suit-manifest-encrypted`.

`suit-manifest` contains a `SUIT_Manifest` structure, which describes the payload(s) to be installed and any dependencies on other manifests.

`suit-manifest-encryption-info` contains a `SUIT_Encryption_Wrapper`, a COSE object that describes the information required to decrypt a ciphertext manifest.

`suit-manifest-encrypted` contains a ciphertext manifest.

Each of `suit-dependency-resolution`, `suit-payload-fetch`, and `suit-payload-installation` contain the severable contents of the identically named portions of the manifest, described in Section 7.2.

`suit-text` contains all the human-readable information that describes any and all parts of the manifest, its payload(s) and its resource(s).

`suit-coswid` contains a Concise Software Identifier. This may be discarded by the recipient if not needed.

7.2. Manifest

The manifest describes the critical metadata for the referenced payload(s). In addition, it contains:

1. a version number for the manifest structure itself
2. a sequence number
3. a list of dependencies
4. a list of components affected
5. a list of components affected by dependencies
6. a reference for each of the severable blocks.
7. a list of actions that the recipient should perform.

The following CDDL fragment defines the manifest.

```

SUIT_Manifest = {
    suit-manifest-version          => 1,
    suit-manifest-sequence-number => uint,
    suit-common                    => bstr .cbor SUIT_Common,
    ? suit-dependency-resolution  => Digest / bstr .cbor SUIT_Command_Sequence,
    ? suit-payload-fetch          => Digest / bstr .cbor SUIT_Command_Sequence,
    ? suit-install                => Digest / bstr .cbor SUIT_Command_Sequence
    ? suit-validate               => bstr .cbor SUIT_Command_Sequence
    ? suit-load                   => bstr .cbor SUIT_Command_Sequence
    ? suit-run                    => bstr .cbor SUIT_Command_Sequence
    ? suit-text-info              => Digest / bstr .cbor SUIT_Text_Map
    ? suit-coswid                 => Digest / bstr .cbor COSWID
}

SUIT_Common = {
    ? suit-dependencies            => bstr .cbor [ + SUIT_Dependency ],
    ? suit-components             => bstr .cbor [ + SUIT_Component_Identifier ],
    ? suit-dependency-components  => bstr .cbor [ + SUIT_Component_Reference ],
    ? suit-common-sequence        => bstr .cbor SUIT_Command_Sequence,
}

```

Several fields in the Manifest can be either a CBOR structure or a SUIT_Digest. In each of these cases, the SUIT_Digest provides for a severable field. Severable fields are RECOMMENDED to implement. In particular, text SHOULD be severable, since most useful text elements occupy more space than a SUIT_Digest, but are not needed by recipient devices. Because SUIT_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 is been severable. The key used for a severable element is the same in the SUIT_Manifest and in the SUIT_Outer_Wrapper so that a recipient can easily identify the correct data in the outer wrapper.

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

The suit-manifest-sequence-number is a monotonically increasing anti-rollback counter. It also helps devices to determine which in a set of manifests is the "root" manifest in a given update. Each manifest MUST have a sequence number higher than each of its dependencies. Each recipient MUST reject any manifest that has a sequence number lower than its current sequence number. It MAY be convenient to use a UTC timestamp in seconds as the sequence number. suit-manifest-sequence-number is REQUIRED.

suit-common encodes all the information that is shared between each of the command sequences, including: suit-dependencies, suit-

components, suit-dependency-components, and suit-common-sequence. suit-common is REQUIRED to implement.

suit-dependencies is a list of SUIT_Dependency blocks that specify manifests that must be present before the current manifest can be processed. suit-dependencies is OPTIONAL to implement.

In order to distinguish between components that are affected by the current manifest and components that are affected by a dependency, they are kept in separate lists. Components affected by the current manifest only list the component identifier. Components affected by a dependency include the component identifier and the index of the dependency that defines the component.

suit-components is a list of SUIT_Component blocks that specify the component identifiers that will be affected by the content of the current manifest. suit-components is OPTIONAL, but at least one manifest MUST contain a suit-components block.

suit-dependency-components is a list of SUIT_Component_Reference blocks that specify component identifiers that will be affected by the content of a dependency of the current manifest. suit-dependency-components is OPTIONAL.

suit-common-sequence is a SUIT_Command_Sequence to execute prior to executing any other command sequence. Typical actions in suit-common-sequence include setting expected device identity and image digests when they are conditional (see Section 11 for more information on conditional sequences). suit-common-sequence is RECOMMENDED.

suit-dependency-resolution is a SUIT_Command_Sequence to execute in order to perform dependency resolution. Typical actions include configuring URIs of dependency manifests, fetching dependency manifests, and validating dependency manifests' contents. suit-dependency-resolution is REQUIRED when suit-dependencies is present.

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.

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.

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 and manifest validation. suit-validate is REQUIRED. If the manifest contains dependencies, one process-dependency invocation per dependency or one process-dependency invocation targeting all dependencies SHOULD be present in validate.

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.

suit-run is a SUIT_Command_Sequence to execute in order to run an image. suit-run typically contains a single instruction: either the "run" directive for the bootable manifest or the "process dependencies" directive for any dependents of the bootable manifest. suit-run is OPTIONAL. Only one manifest in an update may contain the "run" directive.

suit-text-info is a digest that uniquely identifies the content of the Text that is packaged in the OuterWrapper. text is OPTIONAL.

suit-coswid is a digest that uniquely identifies the content of the concise-software-identifier that is packaged in the OuterWrapper. coswid is OPTIONAL.

7.3. SUIT_Dependency

SUIT_Dependency specifies a manifest that describes a dependency of the current manifest.

The following CDDL describes the SUIT_Dependency structure.

```
SUIT_Dependency = {  
    suit-dependency-digest => SUIT_Digest,  
    ? suit-dependency-prefix => SUIT_Component_Identifier,  
}
```

The suit-dependency-digest specifies the dependency manifest uniquely by identifying a particular Manifest structure. The digest is calculated over the Manifest structure instead of the COSE Sig_structure or Mac_structure. This means that a digest may need to be calculated more than once, however this is necessary to ensure that removing a signature from a manifest does not break dependencies due to missing signature elements. This is also necessary to support the trusted intermediary use case, where an intermediary re-signs the

Manifest, removing the original signature, potentially with a different algorithm, or trading COSE_Sign for COSE_Mac.

The suit-dependency-prefix element contains a `SUIT_Component_Identifier`. This specifies the scope at which the dependency operates. This allows the dependency to be forwarded on to a component that is capable of parsing its own manifests. It also allows one manifest to be deployed to multiple dependent devices without those devices needing consistent component hierarchy. This element is OPTIONAL.

7.4. `SUIT_Component_Reference`

The `SUIT_Component_Reference` describes an image that is defined by another manifest. This is useful for overriding the behaviour of another manifest, for example by directing the recipient to look at a different URI for the image or by changing the expected format, such as when a gateway performs decryption on behalf of a constrained device. The following CDDL describes the `SUIT_Component_Reference`.

```
SUIT_Component_Reference = {
    suit-component-identifier => SUIT_Component_Identifier,
    suit-component-dependency-index => uint
}
```

7.5. Manifest Parameters

Many conditions and directives require additional information. That information is contained within parameters that can be set in a consistent way. Parameters MUST only be:

1. Integers
2. Byte strings
3. Booleans

This allows reduction of manifest size and replacement of parameters from one manifest to the next. Byte strings MAY contain CBOR-encoded objects.

The defined manifest parameters are described below.

Parameter Code	CBOR Type	Default	Scope	Name	Description
1	boolean	True	Global	Strict Order	Requires that the manifest

						is processed in a strictly linear fashion. Set to 0 to enable parallel handling of manifest directives.
2	boolean	False	Command Segment	Coerce Condition Failure		Coerces the success code of a command segment to success even when aborted due to a condition failure.
3	bstr	nil	Component/Global	Vendor ID		A RFC4122 UUID representing the vendor of the device or component
4	bstr	nil	Component/Global	Class ID		A RFC4122 UUID representing the class of the device or component
5	bstr	nil	Component/Global	Device ID		A RFC4122 UUID representing the device or component
6	bstr	nil	Component/Dependency	URI		A URI from which to fetch a resource
7	bstr	nil	Component/Dependency	Encryption Info		A COSE object defining the encryption mode of a resource
8	bstr	nil	Component	Compress		A SUIT_Compress

					ion Info	sion_Info object
9	bstr	nil	Component		Unpack Info	A SUIT_Unpack_ Info object
10	uint	nil	Component		Source C omponent	A Component Index
11	bstr	nil	Component/Dep endency		Image Digest	A SUIT_Digest
12	bstr	nil	Component/Dep endency		Image Size	Integer size
24	bstr	nil	Component/Dep endency		URI List	A CBOR encoded list of ranked URIs
25	boole an	Fals e	Component/Dep endency		URI List Append	A CBOR encoded list of ranked URIs
nint	int/b str	nil	Custom		Custom P arameter	Application- defined parameter

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 modularisation and division of responsibility within a pull parser. The same consideration does not apply to Conditions and Directives because those elements are invoked with their arguments immediately

7.5.1. 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 parallelise 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.

7.5.2. SUIT_Parameter_Coerce_Condition_Failure

When executing a command sequence inside SUIT_Run_Sequence and a condition failure occurs, the manifest processor aborts the sequence. If Coerce Condition Failure is True, it returns Success. Otherwise, it returns the original condition failure.

SUIT_Parameter_Coerce_Condition_Failure is scoped to the enclosing SUIT_Directive_Run_Sequence. Its value is discarded when SUIT_Directive_Run_Sequence terminates.

7.6. SUIT_Parameter_Encryption_Info

Encryption Info defines the mechanism that Fetch or Copy should use to decrypt the data they transfer. SUIT_Parameter_Encryption_Info is encoded as a COSE_Encrypt_Tagged or a COSE_Encrypt0_Tagged, wrapped in a bstr

7.7. SUIT_Parameter_Compression_Info

Compression Info defines any information that is required for a device to perform decompression operations. Typically, this includes the algorithm identifier.

SUIT_Parameter_Compression_Info is defined by the following CDDL:

```
SUIT_Compression_Info = {  
    suit-compression-algorithm => SUIT_Compression_Algorithms  
    ? suit-compression-parameters => bstr  
}
```

```
SUIT_Compression_Algorithms /= SUIT_Compression_Algorithm_gzip  
SUIT_Compression_Algorithms /= SUIT_Compression_Algorithm_bzip2  
SUIT_Compression_Algorithms /= SUIT_Compression_Algorithm_deflate  
SUIT_Compression_Algorithms /= SUIT_Compression_Algorithm_LZ4  
SUIT_Compression_Algorithms /= SUIT_Compression_Algorithm_lzma
```

7.8. SUIT_Parameter_Unpack_Info

SUIT_Unpack_Info defines the information required for a device to interpret a packed format, such as elf, hex, or binary diff. SUIT_Unpack_Info is defined by the following CDDL:

```
SUIT_Unpack_Info = {  
    suit-unpack-algorithm => SUIT_Unpack_Algorithms  
    ? suit-unpack-parameters => bstr  
}  
  
SUIT_Unpack_Algorithms // = SUIT_Unpack_Algorithm_Delta  
SUIT_Unpack_Algorithms // = SUIT_Unpack_Algorithm_Hex  
SUIT_Unpack_Algorithms // = SUIT_Unpack_Algorithm_Elf
```

7.9. SUIT_Parameters CDDL

The following CDDL describes all SUIT_Parameters.

```

SUIT_Parameters //= (suit-parameter-strict-order => bool)
SUIT_Parameters //= (suit-parameter-coerce-condition-failure => bool)
SUIT_Parameters //= (suit-parameter-vendor-id => bstr)
SUIT_Parameters //= (suit-parameter-class-id => bstr)
SUIT_Parameters //= (suit-parameter-device-id => bstr)
SUIT_Parameters //= (suit-parameter-uri => bstr)
SUIT_Parameters //= (suit-parameter-encryption-info => bstr .cbor SUIT_Encryption_
n_Info)
SUIT_Parameters //= (suit-parameter-compression-info => bstr .cbor SUIT_Compress
ion_Info)
SUIT_Parameters //= (suit-parameter-unpack-info => bstr .cbor SUIT_Unpack_Info)
SUIT_Parameters //= (suit-parameter-source-component => bstr .cbor SUIT_Componen
t_Identifier)
SUIT_Parameters //= (suit-parameter-image-digest => bstr .cbor SUIT_Digest)
SUIT_Parameters //= (suit-parameter-image-size => uint)
SUIT_Parameters //= (suit-parameter-uri-list => bstr .cbor SUIT_URI_List)
SUIT_Parameters //= (suit-parameter_custom => int/bool/bstr)

SUIT_URI_List = [ + [priority: int, uri: tstr] ]

SUIT_Encryption_Info= COSE_Encrypt_Tagged/COSE_Encrypt0_Tagged
SUIT_Compression_Info = {
    suit-compression-algorithm => SUIT_Compression_Algorithms
    ? suit-compression-parameters => bstr
}

SUIT_Compression_Algorithms /= SUIT_Compression_Algorithm_gzip
SUIT_Compression_Algorithms /= SUIT_Compression_Algorithm_bzip2
SUIT_Compression_Algorithms /= SUIT_Compression_Algorithm_deflate
SUIT_Compression_Algorithms /= SUIT_Compression_Algorithm_LZ4
SUIT_Compression_Algorithms /= SUIT_Compression_Algorithm_lzma

SUIT_Unpack_Info = {
    suit-unpack-algorithm => SUIT_Unpack_Algorithms
    ? suit-unpack-parameters => bstr
}

SUIT_Unpack_Algorithms //= SUIT_Unpack_Algorithm_Delta
SUIT_Unpack_Algorithms //= SUIT_Unpack_Algorithm_Hex
SUIT_Unpack_Algorithms //= SUIT_Unpack_Algorithm_Elf

```

7.10. 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. Dependency Resolution
2. Payload Fetch

3. Payload Installation
4. Image Validation
5. Image Loading
6. Run or Boot

Each of these 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-any failure is treated as a failure of the update/load/boot
2. Directives that MUST be executed.

The lists of commands are logically structured into sequences of zero or more conditions followed by zero or more directives. The *logical* structure is described by the following CDDL:

```
Command_Sequence = {  
    conditions => [ * Condition],  
    directives => [ * Directive]  
}
```

This introduces significant complexity in the parser, however, so the structure is flattened to make parsing simpler:

```
SUIT_Command_Sequence = [ + (SUIT_Condition/SUIT_Directive) ]
```

Each condition and directive is composed of:

1. A command code identifier
2. An argument block

Argument blocks are defined for each type of command.

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 with a matching command to set the current dependency index. This index is a numeric index into the component ID tables defined at the beginning of the document. For the purpose of setting the index, the two component ID tables are considered to be concatenated together.

To facilitate optional conditions, a special directive 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/boot, but a parameter is provided to override this behaviour.

7.11. SUIT_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. Conditions include:

Condition Code	Condition Name	Argument Type
1	Vendor Identifier	nil
2	Class Identifier	nil
3	Image Match	nil
4	Use Before	Unsigned Integer timestamp
5	Component Offset	Unsigned Integer
24	Device Identifier	nil
25	Image Not Match	nil
26	Minimum Battery	Unsigned Integer
27	Update Authorised	Integer
28	Version	List of Integers
nint	Custom Condition	bstr

Each condition MUST report a success code on completion. If a condition reports failure, then the current sequence of commands MUST terminate. If a recipient encounters an unknown Condition Code, it MUST report a failure.

Positive Condition numbers are reserved for IANA registration. Negative numbers are reserved for proprietary, application-specific directives.

7.11.1. Identifier Conditions

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 device MUST match the specified UUID in order to consider the manifest valid. These identifiers MAY be scoped by component.

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.

7.11.2. `suit-condition-image-match`

Verify that the current component matches the digest parameter 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.

7.11.3. `suit-condition-image-not-match`

Verify that the current component does not match the supplied digest. If no digest is specified, then the digest is compared against the digest specified in the Components list. If no digest is specified and the component is not present in the Components list, the condition fails. `suit-condition-image-not-match` is OPTIONAL to implement.

7.11.4. `suit-condition-use-before`

Verify that the current time is BEFORE the specified time. `suit-condition-use-before` is used to specify the last time at which an update should be installed. One argument is required, encoded as a POSIX timestamp, that is seconds after 1970-01-01 00:00:00. Timestamp conditions MUST be evaluated in 64 bits, regardless of encoded CBOR size. `suit-condition-use-before` is OPTIONAL to implement.

7.11.5. `suit-condition-minimum-battery`

`suit-condition-minimum-battery` provides a mechanism to test a device's battery level before installing an update. This condition is for use in primary-cell applications, where the battery is only ever discharged. For batteries that are charged, `suit-directive-wait`

is more appropriate, since it defines a "wait" until the battery level is sufficient to install the update. `suit-condition-minimum-battery` is specified in mWh. `suit-condition-minimum-battery` is OPTIONAL to implement.

7.11.6. `suit-condition-update-authorized`

Request Authorisation from the application and fail if not authorised. This can allow a user to decline an update. Argument is an integer priority level. Priorities are application defined. `suit-condition-update-authorized` is OPTIONAL to implement.

7.11.7. `suit-condition-version`

`suit-condition-version` allows comparing versions of firmware. Verifying image digests is preferred to version checks because digests are more precise. The image can be compared as:

- Greater
- Greater or Equal
- Equal
- Lesser or Equal
- Lesser

Versions are encoded as a CBOR list of integers. Comparisons are done on each integer in sequence. Comparison stops after all integers in the list defined by the manifest have been consumed OR after a non-equal match has occurred. For example, if the manifest defines a comparison, "Equal [1]", then this will match all version sequences starting with 1. If a manifest defines both "Greater or Equal [1,0]" and "Lesser [1,10]", then it will match versions 1.0.x up to, but not including 1.10.

The following CDDL describes `SUIT_Condition_Version_Argument`

```

SUIT_Condition_Version_Argument = [
    suit-condition-version-comparison: SUIT_Condition_Version_Comparison_Types,
    suit-condition-version-comparison: SUIT_Condition_Version_Comparison_Value
]
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Greater
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Greater_Equal
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Equal
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Lesser_Equal
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Lesser
SUIT_Condition_Version_Comparison_Greater = 1
SUIT_Condition_Version_Comparison_Greater_Equal = 2
SUIT_Condition_Version_Comparison_Equal = 3
SUIT_Condition_Version_Comparison_Lesser_Equal = 4
SUIT_Condition_Version_Comparison_Lesser = 5

SUIT_Condition_Version_Comparison_Value = [+int]

```

While the exact encoding of versions is application-defined, semantic versions map conveniently. For example,

- 1.2.3 = [1,2,3]
- 1.2-rc3 = [1,2,-1,3]
- 1.2-beta = [1,2,-2]
- 1.2-alpha = [1,2,-3]
- 1.2-alpha4 = [1,2,-3,4]

suit-condition-version is OPTIONAL to implement.

7.11.8. SUIT_Condition_Custom

SUIT_Condition_Custom describes any proprietary, application specific condition. This is encoded as a negative integer, chosen by the firmware developer, and a bstr that encodes the parameters passed to the system that evaluates the condition matching that integer. SUIT_Condition_Custom is OPTIONAL to implement.

7.11.9. Identifiers

Many 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 explicitly NOT human-readable. They are for machine-based matching only.

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

A more complete example: 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.

7.11.9.1. Creating UUIDs:

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)

The RECOMMENDED method to create a class ID is: Class ID = UUID5(Vendor ID, Class-Specific-Information)

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

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

7.11.10. SUIT_Condition CDDL

The following CDDL describes SUIT_Condition:

```
SUIT_Condition //= (suit-condition-vendor-identifier, nil)
SUIT_Condition //= (suit-condition-class-identifier, nil)
SUIT_Condition //= (suit-condition-device-identifier, nil)
SUIT_Condition //= (suit-condition-image-match, nil)
SUIT_Condition //= (suit-condition-image-not-match, nil)
SUIT_Condition //= (suit-condition-use-before, uint)
SUIT_Condition //= (suit-condition-minimum-battery, uint)
SUIT_Condition //= (suit-condition-update-authorized, int)
SUIT_Condition //= (suit-condition-version, SUIT_Condition_Version_Argument)
SUIT_Condition //= (suit-condition-component-offset, uint)
SUIT_Condition //= (suit-condition-custom, bstr)

SUIT_Condition_Version_Argument = [
    suit-condition-version-comparison: SUIT_Condition_Version_Comparison_Types,
    suit-condition-version-comparison: SUIT_Condition_Version_Comparison_Value
]
SUIT_Condition_Version_Comparison_Types /= suit-condition-version-comparison-greater
SUIT_Condition_Version_Comparison_Types /= suit-condition-version-comparison-greater-equal
SUIT_Condition_Version_Comparison_Types /= suit-condition-version-comparison-equal
SUIT_Condition_Version_Comparison_Types /= suit-condition-version-comparison-less-equal
SUIT_Condition_Version_Comparison_Types /= suit-condition-version-comparison-less
SUIT_Condition_Version_Comparison_Value = [+int]
```

7.12. SUIT_Directive

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

Directive Code	Directive Name
12	Set Component Index
13	Set Dependency Index
14	Abort
15	Try Each
16	Reserved
17	Reserved
18	Process Dependency
19	Set Parameters
20	Override Parameters
21	Fetch
22	Copy
23	Run
29	Wait
30	Run Sequence
31	Run with Arguments
32	Swap

When a Recipient executes a Directive, it MUST report a success code. If the Directive reports failure, then the current Command Sequence MUST terminate.

7.12.1. suit-directive-set-component-index

Set Component Index defines the component to which successive directives and conditions will apply. The supplied argument MUST be either a boolean or an unsigned integer index into the concatenation of suit-components and suit-dependency-components. If the following directives apply to ALL components, then the boolean value "True" is used instead of an index. True does not apply to dependency

components. If the following directives apply to NO components, then the boolean value "False" is used. When `suit-directive-set-dependency-index` is used, `suit-directive-set-component-index = False` is implied. When `suit-directive-set-component-index` is used, `suit-directive-set-dependency-index = False` is implied.

The following CDDL describes the argument to `suit-directive-set-component-index`.

`SUIT_Directive_Set_Component_Index_Argument = uint/bool`

7.12.2. `suit-directive-set-dependency-index`

Set Dependency Index defines the manifest to which successive directives and conditions will apply. The supplied argument MUST be either a boolean or an unsigned integer index into the dependencies. If the following directives apply to ALL dependencies, then the boolean value "True" is used instead of an index. If the following directives apply to NO dependencies, then the boolean value "False" is used. When `suit-directive-set-component-index` is used, `suit-directive-set-dependency-index = False` is implied. When `suit-directive-set-dependency-index` is used, `suit-directive-set-component-index = False` is implied.

Typical operations that require `suit-directive-set-dependency-index` include setting a source URI, invoking "Fetch," or invoking "Process Dependency" for an individual dependency.

The following CDDL describes the argument to `suit-directive-set-dependency-index`.

`SUIT_Directive_Set_Manifest_Index_Argument = uint/bool`

7.12.3. `suit-directive-abort`

Unconditionally fail. This operation is typically used in conjunction with `suit-directive-try-each`.

7.12.4. `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.

The following CDDL describes the `SUIT_Run_Sequence` argument.

```
SUIT_Directive_Run_Sequence_Argument = bstr .cbor SUIT_Command_Sequence
```

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

`SUIT_Parameter_Coerce_Condition_Failure` defaults to False when `suit-directive-run-sequence` begins. Its value is discarded when `suit-directive-run-sequence` terminates.

7.12.5. `suit-directive-try-each`

This command runs several `suit-directive-run-sequence` 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_Coerce_Condition_Failure` is initialised 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 following CDDL describes the `SUIT_Try_Each` argument.

```
SUIT_Directive_Try_Each_Argument = [  
  + bstr .cbor SUIT_Command_Sequence,  
  nil / bstr .cbor SUIT_Command_Sequence  
]
```

7.12.6. `suit-directive-process-dependency`

Execute the commands in the common section of the current dependency, followed by the commands in the equivalent section of the current dependency. For example, if the current section is "fetch payload," this will execute "common" in the current dependency, then "fetch payload" in the current dependency. Once this is complete, the command following `suit-directive-process-dependency` will be processed.

If the current dependency is False, this directive has no effect. If the current dependency is True, then this directive applies to all

dependencies. If the current section is "common," this directive MUST have no effect.

When `SUIT_Process_Dependency` completes, it forwards the last status code that occurred in the dependency.

The argument to `suit-directive-process-dependency` is defined in the following CDDL.

```
SUIT_Directive_Process_Dependency_Argument = nil
```

7.12.7. `suit-directive-set-parameters`

`suit-directive-set-parameters` allows the manifest to configure behaviour of future directives by changing parameters that are read by those directives. When dependencies are used, `suit-directive-set-parameters` also allows a manifest to modify the behaviour of its dependencies.

Available parameters are defined in Section 7.5.

If a parameter is already set, `suit-directive-set-parameters` will skip setting the parameter to its argument. This provides the core of the override mechanism, allowing dependent manifests to change the behaviour of a manifest.

The argument to `suit-directive-set-parameters` is defined in the following CDDL.

```
SUIT_Directive_Set_Parameters_Argument = {+ SUIT_Parameters}
```

N.B.: A directive code is reserved for an optimisation: a way to set a parameter to the contents of another parameter, optionally with another component ID.

7.12.8. `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 7.5.

The argument to `suit-directive-override-parameters` is defined in the following CDDL.

```
SUIT_Directive_Override_Parameters_Argument = {+ SUIT_Parameters}
```

7.12.9. 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 manifests and one or more payloads. suit-directive-fetch retrieves each component and each manifest listed in component-index and manifest-index, respectively. If component-index or manifest-index is True, instead of an integer, then all current manifest components/manifests are fetched. The current manifest's dependent-components are not automatically fetched. In order to pre-fetch these, they MUST be specified in a component-index integer.

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

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

The behaviour of suit-directive-fetch can be modified by setting one or more of SUIT_Parameter_Encryption_Info, SUIT_Parameter_Compression_Info, SUIT_Parameter_Unpack_Info. These three parameters each activate and configure a processing step that can be applied to the data that is transferred during suit-directive-fetch.

The argument to suit-directive-fetch is defined in the following CDDL.

```
SUIT_Directive_Fetch_Argument = nil/bstr
```

7.12.10. suit-directive-copy

suit-directive-copy instructs the manifest processor to obtain one or more payloads, as specified by the component index. suit-directive-copy retrieves each component listed in component-index, respectively. If component-index is True, instead of an integer, then all current manifest components are copied. The current manifest's dependent-components are not automatically copied. In order to copy these, they MUST be specified in a component-index integer.

The behaviour of suit-directive-copy can be modified by setting one or more of SUIT_Parameter_Encryption_Info, SUIT_Parameter_Compression_Info, SUIT_Parameter_Unpack_Info. These

three parameters each activate and configure a processing step that can be applied to the data that is transferred during suit-directive-copy.

N.B. Fetch and Copy are very similar. Merging them into one command may be appropriate.

suit-directive-copy reads its source from
SUIT_Parameter_Source_Component.

The argument to suit-directive-copy is defined in the following CDDL.

SUIT_Directive_Copy_Argument = nil

7.12.11. 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. 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.

7.12.12. 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 provided to Run 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.

The argument to suit-directive-run is defined in the following CDDL.

SUIT_Directive_Run_Argument = nil/bstr

7.12.13. suit-directive-wait

suit-directive-wait directs the manifest processor to pause until a specified event occurs. Some possible events include:

1. Authorisation
2. External Power
3. Network availability
4. Other Device Firmware Version
5. Time
6. Time of Day
7. Day of Week

The following CDDL defines the encoding of these events.

```
SUIT_Wait_Events //= (suit-wait-event-authorisation => int)
SUIT_Wait_Events //= (suit-wait-event-power => int)
SUIT_Wait_Events //= (suit-wait-event-network => int)
SUIT_Wait_Events //= (suit-wait-event-other-device-version
=> SUIT_Wait_Event_Argument_Other_Device_Version)
SUIT_Wait_Events //= (suit-wait-event-time => uint); Timestamp
SUIT_Wait_Events //= (suit-wait-event-time-of-day
=> uint); Time of Day (seconds since 00:00:00)
SUIT_Wait_Events //= (suit-wait-event-day-of-week
=> uint); Days since Sunday

SUIT_Wait_Event_Argument_Authorisation = int ; priority
SUIT_Wait_Event_Argument_Power = int ; Power Level
SUIT_Wait_Event_Argument_Network = int ; Network State
SUIT_Wait_Event_Argument_Other_Device_Version = [
    other-device: bstr,
    other-device-version: [+int]
]
SUIT_Wait_Event_Argument_Time = uint ; Timestamp
SUIT_Wait_Event_Argument_Time_Of_Day = uint ; Time of Day (seconds since 00:00:00)
SUIT_Wait_Event_Argument_Day_Of_Week = uint ; Days since Sunday
```

7.12.14. SUIT_Directive CDDL

The following CDDL describes SUIT_Directive:

```

SUIT_Directive //= (suit-directive-set-component-index, uint/bool)
SUIT_Directive //= (suit-directive-set-dependency-index, uint/bool)
SUIT_Directive //= (suit-directive-run-sequence,
                    bstr .cbor SUIT_Command_Sequence)
SUIT_Directive //= (suit-directive-try-each,
                    SUIT_Directive_Try_Each_Argument)
SUIT_Directive //= (suit-directive-process-dependency, nil)
SUIT_Directive //= (suit-directive-set-parameters,
                    {+ SUIT_Parameters})
SUIT_Directive //= (suit-directive-override-parameters,
                    {+ SUIT_Parameters})
SUIT_Directive //= (suit-directive-fetch, nil)
SUIT_Directive //= (suit-directive-copy, nil)
SUIT_Directive //= (suit-directive-run, nil)
SUIT_Directive //= (suit-directive-wait,
                    { + SUIT_Wait_Events })
SUIT_Directive //= (suit-directive-run-with-arguments, bstr)

SUIT_Directive_Try_Each_Argument = [
    + bstr .cbor SUIT_Command_Sequence,
    nil / bstr .cbor SUIT_Command_Sequence
]

SUIT_Wait_Events //= (suit-wait-event-authorisation => int)
SUIT_Wait_Events //= (suit-wait-event-power => int)
SUIT_Wait_Events //= (suit-wait-event-network => int)
SUIT_Wait_Events //= (suit-wait-event-other-device-version
    => SUIT_Wait_Event_Argument_Other_Device_Version)
SUIT_Wait_Events //= (suit-wait-event-time => uint); Timestamp
SUIT_Wait_Events //= (suit-wait-event-time-of-day
    => uint); Time of Day (seconds since 00:00:00)
SUIT_Wait_Events //= (suit-wait-event-day-of-week
    => uint); Days since Sunday

SUIT_Wait_Event_Argument_Authorisation = int ; priority
SUIT_Wait_Event_Argument_Power = int ; Power Level
SUIT_Wait_Event_Argument_Network = int ; Network State
SUIT_Wait_Event_Argument_Other_Device_Version = [
    other-device: bstr,
    other-device-version: [+int]
]
SUIT_Wait_Event_Argument_Time = uint ; Timestamp
SUIT_Wait_Event_Argument_Time_Of_Day = uint ; Time of Day (seconds since 00:00:00)
SUIT_Wait_Event_Argument_Day_Of_Week = uint ; Days since Sunday

```

8. Dependency processing

Dependencies need careful handling on constrained systems. A dependency tree that is too deep can cause recursive handling to overflow stack space. Systems that parse all dependencies into an object tree can easily fill up available memory. Too many dependencies can overrun available storage space.

The dependency handling system in this document is designed to address as many of these problems as possible.

Dependencies MAY be addressed in one of three ways:

1. Iterate by component
2. Iterate by manifest
3. Out-of-order

Because each manifest has a list of components and a list of components defined by its dependencies, it is possible for the manifest processor to handle one component at a time, traversing the manifest tree once for each listed component. This, however consumes significant processing power.

Alternatively, it is possible for a device with sufficient memory to accumulate all parameters for all listed component IDs. This will naturally consume more memory, but it allows the device to process the manifests in a single pass.

It is expected that the simplest and most power sensitive devices will use option 2, with a fixed maximum number of components.

Advanced devices may make use of the Strict Order parameter and enable parallel processing of some segments, or it may reorder some segments. To perform parallel processing, once the Strict Order parameter is set to False, the device may fork a process for each command until the Strict Order parameter is returned to True or the command sequence ends. Then, it joins all forked processes before continuing processing of commands. To perform out-of-order processing, a similar approach is used, except the device 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.

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 device, authenticated by a trusted party or stored on the device. This ACL grants access rights for specific component IDs or component ID prefixes to the listed identities or identity groups. Any identity may verify an image digest, but fetching into or fetching from a component ID requires approval from the ACL.

A third model allows a device to provide even more fine-grained controls: The ACL lists the component ID or component ID prefix that an identity may use, and also lists the commands that the identity may use in combination with that component ID.

10. SUIT digest container

RFC 8152 [RFC8152] provides containers for signature, MAC, and encryption, but no basic digest container. The container needed for a digest requires a type identifier and a container for the raw digest data. Some forms of digest may require additional parameters. These can be added following the digest. This structure is described by the following CDDL.

The algorithms listed are sufficient for verifying integrity of Firmware Updates as of this writing, however this may change over time.


```
SUIT_Digest = [
  suit-digest-algorithm-id : $suit-digest-algorithm-ids,
  suit-digest-bytes : bytes,
  ? suit-digest-parameters : any
]
```

```
digest-algorithm-ids /= algorithm-id-sha224
digest-algorithm-ids /= algorithm-id-sha256
digest-algorithm-ids /= algorithm-id-sha384
digest-algorithm-ids /= algorithm-id-sha512
digest-algorithm-ids /= algorithm-id-sha3-224
digest-algorithm-ids /= algorithm-id-sha3-256
digest-algorithm-ids /= algorithm-id-sha3-384
digest-algorithm-ids /= algorithm-id-sha3-512
```

```
algorithm-id-sha224 = 1
algorithm-id-sha256 = 2
algorithm-id-sha384 = 3
algorithm-id-sha512 = 4
algorithm-id-sha3-224 = 5
algorithm-id-sha3-256 = 6
algorithm-id-sha3-384 = 7
algorithm-id-sha3-512 = 8
```

11. Creating conditional sequences

For some use cases, it is important to provide a sequence that can fail without terminating an update. For example, a dual-image XIP MCU may require an update that can be placed at one of two offsets. This has two implications, first, the digest of each offset will be different. Second, the image fetched for each offset will have a different URI. Conditional sequences allow this to be resolved in a simple way.

The following JSON representation of a manifest demonstrates how this would be represented. It assumes that the bootloader and manifest processor take care of A/B switching and that the manifest is not aware of this distinction.

```
{
  "structure-version" : 1,
  "sequence-number" : 7,
  "common" : {
    "components" : [
      [b'0']
    ],
    "common-sequence" : [
      {
```

```
    "directive-set-var" : {
      "size": 32567
    },
  },
  {
    "try-each" : [
      [
        {"condition-component-offset" : "<offset A>"},
        {
          "directive-set-var": {
            "digest" : "<SHA256 A>"
          }
        }
      ],
      [
        {"condition-component-offset" : "<offset B>"},
        {
          "directive-set-var": {
            "digest" : "<SHA256 B>"
          }
        }
      ],
      [{ "abort" : null }]
    ]
  }
]
}
"fetch" : [
  {
    "try-each" : [
      [
        {"condition-component-offset" : "<offset A>"},
        {
          "directive-set-var": {
            "uri" : "<URI A>"
          }
        }
      ],
      [
        {"condition-component-offset" : "<offset B>"},
        {
          "directive-set-var": {
            "uri" : "<URI B>"
          }
        }
      ],
      [{ "directive-abort" : null }]
    ]
  }
]
```

```

    },
    "fetch" : null
  ]
}

```

12. Full CDDL

In order to create a valid SUIF Manifest document the structure of the corresponding CBOR message MUST adhere to the following CDDL data definition.

```

SUIF_Outer_Wrapper = {
  suit-authentication-wrapper => bstr .cbor SUIF_Authentication_Wrapper / nil,
  suit-manifest                => bstr .cbor SUIF_Manifest,
  suit-dependency-resolution   => bstr .cbor SUIF_Command_Sequence,
  suit-payload-fetch           => bstr .cbor SUIF_Command_Sequence,
  suit-install                 => bstr .cbor SUIF_Command_Sequence,
  suit-text                    => bstr .cbor SUIF_Text_Map,
  suit-coswid                  => bstr .cbor concise-software-identity
}
suit-authentication-wrapper = 1
suit-manifest = 2
suit-dependency-resolution = 7
suit-payload-fetch = 8
suit-install = 9
suit-text = 13
suit-coswid = 14

SUIF_Authentication_Wrapper = [ * (
  COSE_Mac_Tagged /
  COSE_Sign_Tagged /
  COSE_Mac0_Tagged /
  COSE_Sign1_Tagged) ]

COSE_Mac_Tagged = any
COSE_Sign_Tagged = any
COSE_Mac0_Tagged = any
COSE_Sign1_Tagged = any
COSE_Encrypt_Tagged = any
COSE_Encrypt0_Tagged = any

SUIF_Digest = [
  suit-digest-algorithm-id : $suit-digest-algorithm-ids,
  suit-digest-bytes : bytes,
  ? suit-digest-parameters : any
]

```

```

; Named Information Hash Algorithm Identifiers
suit-digest-algorithm-ids /= algorithm-id-sha256
suit-digest-algorithm-ids /= algorithm-id-sha256-128
suit-digest-algorithm-ids /= algorithm-id-sha256-120
suit-digest-algorithm-ids /= algorithm-id-sha256-96
suit-digest-algorithm-ids /= algorithm-id-sha256-64
suit-digest-algorithm-ids /= algorithm-id-sha256-32
suit-digest-algorithm-ids /= algorithm-id-sha384
suit-digest-algorithm-ids /= algorithm-id-sha512
suit-digest-algorithm-ids /= algorithm-id-sha3-224
suit-digest-algorithm-ids /= algorithm-id-sha3-256
suit-digest-algorithm-ids /= algorithm-id-sha3-384
suit-digest-algorithm-ids /= algorithm-id-sha3-512

SUIT_Manifest = {
    suit-manifest-version          => 1,
    suit-manifest-sequence-number => uint,
    ? suit-dependencies            => [ + SUIT_Dependency ],
    ? suit-components              => [ + SUIT_Component ],
    ? suit-dependency-components   => [ + SUIT_Component_Reference ],
    ? suit-common                  => bstr .cbor SUIT_Command_Sequence,
    ? suit-dependency-resolution   => SUIT_Digest / bstr .cbor SUIT_Command_Seque
nce,
    ? suit-payload-fetch           => SUIT_Digest / bstr .cbor SUIT_Command_Seque
nce,
    ? suit-install                 => SUIT_Digest / bstr .cbor SUIT_Command_Seque
nce
    ? suit-validate                => bstr .cbor SUIT_Command_Sequence
    ? suit-load                    => bstr .cbor SUIT_Command_Sequence
    ? suit-run                     => bstr .cbor SUIT_Command_Sequence
    ? suit-text-info               => SUIT_Digest / bstr .cbor SUIT_Text_Map
    ? suit-coswid                  => SUIT_Digest / bstr .cbor concise-software-i
dentity
}

suit-manifest-version = 1
suit-manifest-sequence-number = 2
suit-dependencies = 3
suit-components = 4
suit-dependency-components = 5
suit-common = 6
suit-dependency-resolution = 7
suit-payload-fetch = 8
suit-install = 9
suit-validate = 10
suit-load = 11
suit-run = 12
suit-text-info = 13
suit-coswid = 14

concise-software-identity = any

```

```
SUIT_Dependency = {
    suit-dependency-digest => SUIT_Digest,
    suit-dependency-prefix => SUIT_Component_Identifier,
}

suit-dependency-digest = 1
suit-dependency-prefix = 2

SUIT_Component_Identifier = [* bstr]

SUIT_Component = {
    suit-component-identifier => SUIT_Component_Identifier,
    ? suit-component-size => uint,
    ? suit-component-digest => SUIT_Digest,
}

suit-component-identifier = 1
suit-component-size = 2
suit-component-digest = 3

SUIT_Component_Reference = {
    suit-component-identifier => SUIT_Component_Identifier,
    suit-component-dependency-index => uint
}

suit-component-dependency-index = 2

SUIT_Command_Sequence = [ + { SUIT_Condition // SUIT_Directive // SUIT_Command_Custom } ]

SUIT_Command_Custom = (nint => bstr)

SUIT_Condition //= (SUIT_Condition_Vendor_Identifier => RFC4122_UUID) ; SUIT_Condition_Vendor_Identifier
SUIT_Condition //= (2 => RFC4122_UUID) ; SUIT_Condition_Class_Identifier
SUIT_Condition //= (3 => RFC4122_UUID) ; SUIT_Condition_Device_Identifier
SUIT_Condition //= (4 => SUIT_Digest) ; SUIT_Condition_Image_Match
SUIT_Condition //= (5 => SUIT_Digest) ; SUIT_Condition_Image_Not_Match
SUIT_Condition //= (6 => uint) ; SUIT_Condition_Use_Before
SUIT_Condition //= (7 => uint) ; SUIT_Condition_Minimum_Battery
SUIT_Condition //= (8 => int) ; SUIT_Condition_Update_Authorised
SUIT_Condition //= (9 => SUIT_Condition_Version_Argument) ; SUIT_Condition_Version
SUIT_Condition //= (10 => uint) ; SUIT_Condition_Component_Offset
SUIT_Condition //= (nint => bstr) ; SUIT_Condition_Custom

SUIT_Condition_Vendor_Identifier = 1
RFC4122_UUID = bstr .size 16

SUIT_Condition_Version_Argument = [
    suit-condition-version-comparison: SUIT_Condition_Version_Comparison_Types,
```

```

    suit-condition-version-comparison: SUIT_Condition_Version_Comparison_Value
]
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Gre
ater
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Gre
ater_Equal
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Equ
al
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Les
ser_Equal
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Les
ser

SUIT_Condition_Version_Comparison_Greater = 1
SUIT_Condition_Version_Comparison_Greater_Equal = 2
SUIT_Condition_Version_Comparison_Equal = 3
SUIT_Condition_Version_Comparison_Lesser_Equal = 4
SUIT_Condition_Version_Comparison_Lesser = 5

SUIT_Condition_Version_Comparison_Value = [+int]

SUIT_Directive //= (11 => uint/bool) ; SUIT_Directive_Set_Component_Index
SUIT_Directive //= (12 => uint/bool) ; SUIT_Directive_Set_Manifest_Index
SUIT_Directive //= (13 => bstr .cbor SUIT_Command_Sequence) ; SUIT_Directive_Run
_Sequence
SUIT_Directive //= (14 => bstr .cbor SUIT_Command_Sequence) ; SUIT_Directive_Run
_Sequence_Conditional
SUIT_Directive //= (15 => nil) ; SUIT_Directive_Process_Dependency
SUIT_Directive //= (16 => {+ SUIT_Parameters}) ; SUIT_Directive_Set_Parameters
SUIT_Directive //= (19 => {+ SUIT_Parameters}) ; SUIT_Directive_Override_Paramet
ers
SUIT_Directive //= (20 => nil/bstr) ; SUIT_Directive_Fetch
SUIT_Directive //= (21 => nil/bstr) ; SUIT_Directive_Copy
SUIT_Directive //= (22 => nil/bstr) ; SUIT_Directive_Run
SUIT_Directive //= (23 => { + SUIT_Wait_Events }) ; SUIT_Directive_Wait

SUIT_Wait_Events //= (1 => SUIT_Wait_Event_Argument_Authorisation)
SUIT_Wait_Events //= (2 => SUIT_Wait_Event_Argument_Power)
SUIT_Wait_Events //= (3 => SUIT_Wait_Event_Argument_Network)
SUIT_Wait_Events //= (4 => SUIT_Wait_Event_Argument_Other_Device_Version)
SUIT_Wait_Events //= (5 => SUIT_Wait_Event_Argument_Time)
SUIT_Wait_Events //= (6 => SUIT_Wait_Event_Argument_Time_Of_Day)
SUIT_Wait_Events //= (7 => SUIT_Wait_Event_Argument_Day_Of_Week)

SUIT_Wait_Event_Argument_Authorisation = int ; priority
SUIT_Wait_Event_Argument_Power = int ; Power Level
SUIT_Wait_Event_Argument_Network = int ; Network State
SUIT_Wait_Event_Argument_Other_Device_Version = [
    other-device: bstr,
    other-device-version: [+int]
]
SUIT_Wait_Event_Argument_Time = uint ; Timestamp
SUIT_Wait_Event_Argument_Time_Of_Day = uint ; Time of Day (seconds since 00:00:0
0)
SUIT_Wait_Event_Argument_Day_Of_Week = uint ; Days since Sunday

```

```
SUIT_Parameters //= (1 => bool) ; SUIT_Parameter_Strict_Order
SUIT_Parameters //= (2 => bool) ; SUIT_Parameter_Coerce_Condition_Failure
SUIT_Parameters //= (3 => bstr) ; SUIT_Parameter_Vendor_ID
SUIT_Parameters //= (4 => bstr) ; SUIT_Parameter_Class_ID
SUIT_Parameters //= (5 => bstr) ; SUIT_Parameter_Device_ID
SUIT_Parameters //= (6 => bstr .cbor SUIT_URI_List) ; SUIT_Parameter_URI_List
SUIT_Parameters //= (7 => bstr .cbor SUIT_Encryption_Info) ; SUIT_Parameter_Encr
yption_Info
SUIT_Parameters //= (8 => bstr .cbor SUIT_Compression_Info) ; SUIT_Parameter_Com
pression_Info
SUIT_Parameters //= (9 => bstr .cbor SUIT_Unpack_Info) ; SUIT_Parameter_Unpack_I
nfo
SUIT_Parameters //= (10 => bstr .cbor SUIT_Component_Identifier) ; SUIT_Paramete
r_Source_Component
SUIT_Parameters //= (11 => bstr .cbor SUIT_Digest) ; SUIT_Parameter_Image_Digest
SUIT_Parameters //= (12 => uint) ; SUIT_Parameter_Image_Size
SUIT_Parameters //= (nint => int/bool/bstr) ; SUIT_Parameter_Custom

SUIT_URI_List = [ + [priority: int, uri: tstr] ]

SUIT_Encryption_Info = COSE_Encrypt_Tagged/COSE_Encrypt0_Tagged
SUIT_Compression_Info = {
    suit-compression-algorithm => SUIT_Compression_Algorithms
    ? suit-compression-parameters => bstr
}
suit-compression-algorithm = 1
suit-compression-parameters = 2

SUIT_Compression_Algorithms /= SUIT_Compression_Algorithm_gzip
SUIT_Compression_Algorithms /= SUIT_Compression_Algorithm_bzip2
SUIT_Compression_Algorithms /= SUIT_Compression_Algorithm_lz4
SUIT_Compression_Algorithms /= SUIT_Compression_Algorithm_lzma

SUIT_Compression_Algorithm_gzip = 1
SUIT_Compression_Algorithm_bzip2 = 2
SUIT_Compression_Algorithm_deflate = 3
SUIT_Compression_Algorithm_lz4 = 4
SUIT_Compression_Algorithm_lzma = 7

SUIT_Unpack_Info = {
    suit-unpack-algorithm => SUIT_Unpack_Algorithms
    ? suit-unpack-parameters => bstr
}
suit-unpack-algorithm = 1
suit-unpack-parameters = 2

SUIT_Unpack_Algorithms /= SUIT_Unpack_Algorithm_Delta
SUIT_Unpack_Algorithms /= SUIT_Unpack_Algorithm_Hex
SUIT_Unpack_Algorithms /= SUIT_Unpack_Algorithm_Elf

SUIT_Unpack_Algorithm_Delta = 1
SUIT_Unpack_Algorithm_Hex = 2
```

```
SUIT_Unpack_Algorithm_Elf = 3
```

```
SUIT_Text_Map = {int => tstr}
```

13. Examples

The following examples demonstrate a small subset of the functionality of the manifest. However, despite this, even a simple manifest processor can execute most of these manifests.

None of these examples include authentication. This is provided via RFC 8152 [RFC8152], and is omitted for clarity.

13.1. Example 0:

Secure boot only.

The following JSON shows the intended behaviour of the manifest.

```
{
  "structure-version": 1,
  "sequence-number": 1,
  "run-image": [
    { "directive-set-component": 0 },
    { "condition-image": null },
    { "directive-run": null }
  ],
  "common": {
    "common-sequence": [
      {
        "directive-set-var": {
          "digest": "00112233445566778899aabbccddeeff"
                    "0123456789abcdeffedcba9876543210",
          "size": 34768
        }
      }
    ],
    "components": [
      [
        "Flash",
        78848
      ]
    ]
  }
}
```

Converted into the SUIT manifest, this produces:


```

{
  / auth object / 1 : None
  / manifest / 3 : h'a40101020103583ca2024c818245466c6173684300340104'
                  h'582a8213a20b582000112233445566778899aabbccddeeff'
                  h'0123456789abcdeffedcba98765432100c1987d00c47860c'
                  h'0003f617f6' \
  {
    / structure-version / 1 : 1
    / sequence-number / 2 : 1
    / common / 3 : h'a2024c818245466c6173684300340104582a8213a20b58'
                  h'2000112233445566778899aabbccddeeff0123456789ab'
                  h'cdeffedcba98765432100c1987d0' \ {
      / components / 2 : h'818245466c61736843003401' \
      [
        [h'466c617368', h'003401'],
      ],
      / common / 4 : h'8213a20b582000112233445566778899aabbccddee'
                  h'ff0123456789abcdeffedcba98765432100c1987d0'
      \ [
        / set-vars / 19, {
          / digest / 11 : h'00112233445566778899aabbccddeeff01'
                      h'23456789abcdeffedcba9876543210',
          / size / 12 : 34768
        },
      ],
    },
    / run-image / 12 : h'860c0003f617f6' \ [
      / set-component-index / 12, 0,
      / condition-image / 3, None,
      / run / 23, None,
    ],
  }
}

```

Total size of outer wrapper without COSE authentication object: 83

Outer:

```

a201f603584da40101020103583ca2024c818245466c6173684300340104582a8213a20b
582000112233445566778899aabbccddeeff0123456789abcdeffedcba98765432100c19
87d00c47860c0003f617f6

```

13.2. Example 1:

Simultaneous download and installation of payload.

The following JSON shows the intended behaviour of the manifest.

```
{
  "structure-version": 1,
  "sequence-number": 2,
  "apply-image": [
    { "directive-set-component": 0 },
    {
      "directive-set-var": {
        "uri": "http://example.com/file.bin"
      }
    },
    { "directive-fetch": null }
  ],
  "common": {
    "common-sequence": [
      {
        "directive-set-var": {
          "digest": "00112233445566778899aabbccddeeff"
            "0123456789abcdeffedcba9876543210",
          "size": 34768
        }
      }
    ],
    "components": [
      [
        "Flash",
        78848
      ]
    ]
  }
}
```

Converted into the SUIT manifest, this produces:

```

{
  / auth object / 1 : None
  / manifest / 3 : h'a40101020203583ca2024c818245466c6173684300340104'
                    h'582a8213a20b582000112233445566778899aabbccddeeff'
                    h'0123456789abcdeffedcba98765432100c1987d009582586'
                    h'0c0013a106781b687474703a2f2f6578616d706c652e636f'
                    h'6d2f66696c652e62696e15f6' \
  {
    / structure-version / 1 : 1
    / sequence-number / 2 : 2
    / common / 3 : h'a2024c818245466c6173684300340104582a8213a20b58'
                  h'2000112233445566778899aabbccddeeff0123456789ab'
                  h'cdeffedcba98765432100c1987d0' \ {
      / components / 2 : h'818245466c61736843003401' \
      [
        [h'466c617368', h'003401'],
      ],
      / common / 4 : h'8213a20b582000112233445566778899aabbccddee'
                    h'ff0123456789abcdeffedcba98765432100c1987d0'
      \ [
        / set-vars / 19, {
          / digest / 11 :h'00112233445566778899aabbccddeeff01'
                      h'23456789abcdeffedcba9876543210',
          / size / 12 : 34768
        },
      ],
    },
    / apply-image / 9 : h'860c0013a106781b687474703a2f2f6578616d70'
                      h'6c652e636f6d2f66696c652e62696e15f6' \ [
      / set-component-index / 12, 0,
      / set-vars / 19, {
        / uri / 6 : http://example.com/file.bin
      },
      / fetch / 21, None,
    ],
  },
}

```

Total size of outer wrapper without COSE authentication object: 114

Outer:

```

a201f603586ca40101020203583ca2024c818245466c6173684300340104582a8213a20b
582000112233445566778899aabbccddeeff0123456789abcdeffedcba98765432100c19
87d0095825860c0013a106781b687474703a2f2f6578616d706c652e636f6d2f66696c65
2e62696e15f6

```

13.3. Example 2:

Compatibility test, simultaneous download and installation, and secure boot.

The following JSON shows the intended behaviour of the manifest.

```
{
  "structure-version": 1,
  "sequence-number": 3,
  "common": {
    "common-sequence": [
      {
        "directive-set-var": {
          "vendor-id": "fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe",
          "class-id": "1492af14-2569-5e48-bf42-9b2d51f2ab45",
          "digest": "00112233445566778899aabbccddeeff"
            "0123456789abcdeffedcba9876543210",
          "size": 34768
        }
      },
      { "condition-vendor-id": null },
      { "condition-class-id": null }
    ],
    "components": [
      [
        "Flash",
        78848
      ]
    ]
  },
  "apply-image": [
    { "directive-set-component": 0 },
    {
      "directive-set-var": {
        "uri": "http://example.com/file.bin"
      }
    },
    { "directive-fetch": null }
  ],
  "run-image": [
    { "directive-set-component": 0 },
    { "condition-image": null },
    { "directive-run": null }
  ]
}
```

Converted into the SUIT manifest, this produces:

```

{
  / auth object / 1 : None
  / manifest / 3 : h'a501010203035864a2024c818245466c6173684300340104'
                    h'58528613a40350fa6b4a53d5ad5fdfe9de663e4d41ffe04'
                    h'501492af1425695e48bf429b2d51f2ab450b582000112233'
                    h'445566778899aabbccddeeff0123456789abcdeffedcba98'
                    h'765432100c1987d001f602f6095825860c0013a106781b68'
                    h'7474703a2f2f6578616d706c652e636f6d2f66696c652e62'
                    h'696e15f60c47860c0003f617f6' \
  {
    / structure-version / 1 : 1
    / sequence-number / 2 : 3
    / common / 3 : h'a2024c818245466c617368430034010458528613a40350'
                  h'fa6b4a53d5ad5fdfe9de663e4d41ffe04501492af1425'
                  h'695e48bf429b2d51f2ab450b5820001122334455667788'
                  h'99aabbccddeeff0123456789abcdeffedcba9876543210'
                  h'0c1987d001f602f6' \ {
      / components / 2 : h'818245466c61736843003401' \
      [
        [h'466c617368', h'003401'],
      ],
      / common / 4 : h'8613a40350fa6b4a53d5ad5fdfe9de663e4d41ffe'
                    h'04501492af1425695e48bf429b2d51f2ab450b5820'
                    h'00112233445566778899aabbccddeeff0123456789'
                    h'abcdeffedcba98765432100c1987d001f602f6' \ [
        / set-vars / 19, {
          / vendor-id / 3 : h'fa6b4a53d5ad5fdfe9de663e4d41f'
                          h'fe'
          / class-id / 4 : h'1492af1425695e48bf429b2d51f2ab45'
          / digest / 11 : h'00112233445566778899aabbccddeeff01'
                        h'23456789abcdeffedcba9876543210',
          / size / 12 : 34768
        },
        / condition-vendor-id / 1, None,
        / condition-class-id / 2, None,
      ],
    },
    / apply-image / 9 : h'860c0013a106781b687474703a2f2f6578616d70'
                      h'6c652e636f6d2f66696c652e62696e15f6' \ [
      / set-component-index / 12, 0,
      / set-vars / 19, {
        / uri / 6 : http://example.com/file.bin
      },
      / fetch / 21, None,
    ],
    / run-image / 12 : h'860c0003f617f6' \ [
      / set-component-index / 12, 0,
      / condition-image / 3, None,
    ],
  },
}

```

```

    / run / 23, None,
  ],
}
}

```

Total size of outer wrapper without COSE authentication object: 163

Outer:

```

a201f603589da501010203035864a2024c818245466c617368430034010458528613a403
50fa6b4a53d5ad5fdfbe9de663e4d41ffe04501492af1425695e48bf429b2d51f2ab450b
582000112233445566778899aabbccddeeff0123456789abcdeffedcba98765432100c19
87d001f602f6095825860c0013a106781b687474703a2f2f6578616d706c652e636f6d2f
66696c652e62696e15f60c47860c0003f617f6

```

13.4. Example 3:

Compatibility test, simultaneous download and installation, load from external storage, and secure boot.

The following JSON shows the intended behaviour of the manifest.

```

{
  "structure-version": 1,
  "sequence-number": 4,
  "common": {
    "common-sequence": [
      {
        "directive-set-var": {
          "vendor-id": "fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe",
          "class-id": "1492af14-2569-5e48-bf42-9b2d51f2ab45"
        }
      },
      { "directive-set-component": 0 },
      {
        "directive-set-var": {
          "digest": "00112233445566778899aabbccddeeff"
                    "0123456789abcdeffedcba9876543210",
          "size": 34768
        }
      },
      { "directive-set-component": 1 },
      {
        "directive-set-var": {
          "digest": "00112233445566778899aabbccddeeff"
                    "0123456789abcdeffedcba9876543210",
          "size": 34768
        }
      }
    ]
  }
}

```

```

    },
    { "condition-vendor-id": null },
    { "condition-class-id": null }
  ],
  "components": [
    [
      "Flash",
      78848
    ],
    [
      "RAM",
      1024
    ]
  ]
},
"apply-image": [
  { "directive-set-component": 0 },
  {
    "directive-set-var": {
      "uri": "http://example.com/file.bin"
    }
  },
  { "directive-fetch": null }
],
"run-image": [
  { "directive-set-component": 0 },
  { "condition-image": null },
  { "directive-set-component": 1 },
  {
    "directive-set-var": {
      "source-index": 0
    }
  },
  { "directive-fetch": null },
  { "condition-image": null },
  { "directive-run": null }
]
}

```

Converted into the SUIT manifest, this produces:

```

{
  / auth object / 1 : None
  / manifest / 3 : h'a50101020403589ba20254828245466c6173684300340182'
                  h'4352414d4200040458818e13a20350fa6b4a53d5ad5fdfbe'
                  h'9de663e4d41ffe04501492af1425695e48bf429b2d51f2ab'
                  h'450c0013a20b582000112233445566778899aabbccddeeff'
                  h'0123456789abcdeffedcba98765432100c1987d00c0113a2'
}

```

```

        h'0b582000112233445566778899aabbccddeeff0123456789'
        h'abcdeffedcba98765432100c1987d001f602f6095825860c'
        h'0013a106781b687474703a2f2f6578616d706c652e636f6d'
        h'2f66696c652e62696e15f60c518e0c0003f60c0113a10a00'
        h'15f603f617f6' \
    {
        / structure-version / 1 : 1
        / sequence-number / 2 : 4
        / common / 3 : h'a20254828245466c61736843003401824352414d420004'
                        h'0458818e13a20350fa6b4a53d5ad5fdfe9de663e4d41f'
                        h'fe04501492af1425695e48bf429b2d51f2ab450c0013a2'
                        h'0b582000112233445566778899aabbccddeeff01234567'
                        h'89abcdeffedcba98765432100c1987d00c0113a20b5820'
                        h'00112233445566778899aabbccddeeff0123456789abcd'
                        h'effedcba98765432100c1987d001f602f6' \ {
        / components / 2 : h'828245466c61736843003401824352414d4200'
                           h'04' \
        [
            [h'466c617368', h'003401'],
            [h'52414d', h'0004'],
        ],
        / common / 4 : h'8e13a20350fa6b4a53d5ad5fdfe9de663e4d41ffe'
                        h'04501492af1425695e48bf429b2d51f2ab450c0013'
                        h'a20b582000112233445566778899aabbccddeeff01'
                        h'23456789abcdeffedcba98765432100c1987d00c01'
                        h'13a20b582000112233445566778899aabbccddeeff'
                        h'0123456789abcdeffedcba98765432100c1987d001'
                        h'f602f6' \ [
        / set-vars / 19, {
            / vendor-id / 3 : h'fa6b4a53d5ad5fdfe9de663e4d41f'
                           h'fe'
            / class-id / 4 : h'1492af1425695e48bf429b2d51f2ab45'
        },
        / set-component-index / 12, 0,
        / set-vars / 19, {
            / digest / 11 :h'00112233445566778899aabbccddeeff01'
                           h'23456789abcdeffedcba9876543210',
            / size / 12 : 34768
        },
        / set-component-index / 12, 1,
        / set-vars / 19, {
            / digest / 11 :h'00112233445566778899aabbccddeeff01'
                           h'23456789abcdeffedcba9876543210',
            / size / 12 : 34768
        },
        / condition-vendor-id / 1, None,
        / condition-class-id / 2, None,
    ],

```



```

    },
    / apply-image / 9 : h'860c0013a106781b687474703a2f2f6578616d70'
                          h'6c652e636f6d2f66696c652e62696e15f6' \ [
        / set-component-index / 12, 0,
        / set-vars / 19, {
            / uri / 6 : http://example.com/file.bin
        },
        / fetch / 21, None,
    ],
    / run-image / 12 : h'8e0c0003f60c0113a10a0015f603f617f6' \ [
        / set-component-index / 12, 0,
        / condition-image / 3, None,
        / set-component-index / 12, 1,
        / set-vars / 19, {
            / source-component / 10 : 0
        },
        / fetch / 21, None,
        / condition-image / 3, None,
        / run / 23, None,
    ],
}
}

```

Total size of outer wrapper without COSE authentication object: 228

Outer:

```

a201f60358dea50101020403589ba20254828245466c61736843003401824352414d4200
040458818e13a20350fa6b4a53d5ad5fdfbe9de663e4d41ffe04501492af1425695e48bf
429b2d51f2ab450c0013a20b582000112233445566778899aabbccddeeff0123456789ab
cdeffedcba98765432100c1987d00c0113a20b582000112233445566778899aabbccdde
ff0123456789abcdeffedcba98765432100c1987d001f602f6095825860c0013a106781b
687474703a2f2f6578616d706c652e636f6d2f66696c652e62696e15f60c518e0c0003f6
0c0113a10a0015f603f617f6

```

13.5. Example 4:

Compatibility test, simultaneous download and installation, load and decompress from external storage, and secure boot.

The following JSON shows the intended behaviour of the manifest.

```

{
  "structure-version": 1,
  "sequence-number": 5,
  "common": {
    "common-sequence": [
      {

```

```

        "directive-set-var": {
            "vendor-id": "fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe",
            "class-id": "1492af14-2569-5e48-bf42-9b2d51f2ab45"
        },
    },
    { "directive-set-component": 0 },
    {
        "directive-set-var": {
            "digest": "00112233445566778899aabbccddeeff"
                    "0123456789abcdeffedcba9876543210",
            "size": 34768
        },
    },
    { "directive-set-component": 1 },
    {
        "directive-set-var": {
            "digest": "0123456789abcdeffedcba9876543210"
                    "00112233445566778899aabbccddeeff",
            "size": 34768
        },
    },
    { "condition-vendor-id": null },
    { "condition-class-id": null }
],
"components": [
    [
        "Flash",
        78848
    ],
    [
        "RAM",
        1024
    ]
],
},
"apply-image": [
    { "directive-set-component": 0 },
    {
        "directive-set-var": {
            "uri": "http://example.com/file.bin"
        },
    },
    { "directive-fetch": null }
],
"load-image": [
    { "directive-set-component": 0 },
    { "condition-image": null },
    { "directive-set-component": 1 },

```

```

    {
        "directive-set-var": {
            "source-index": 0,
            "compression-info": {
                "algorithm": "gzip"
            }
        }
    },
    { "directive-copy": null }
],
"run-image": [
    { "condition-image": null },
    { "directive-run": null }
]
}

```

Converted into the SUIT manifest, this produces:

```

{
  / auth object / 1 : None
  / manifest / 3 : h'a60101020503589ba20254828245466c6173684300340182'
                  h'4352414d4200040458818e13a20350fa6b4a53d5ad5fdfbe'
                  h'9de663e4d41ffe04501492af1425695e48bf429b2d51f2ab'
                  h'450c0013a20b582000112233445566778899aabbccddeeff'
                  h'0123456789abcdeffedcba98765432100c1987d00c0113a2'
                  h'0b58200123456789abcdeffedcba98765432100011223344'
                  h'5566778899aabbccddeeff0c1987d001f602f6095825860c'
                  h'0013a106781b687474703a2f2f6578616d706c652e636f6d'
                  h'2f66696c652e62696e15f60b508a0c0003f60c0113a20841'
                  h'f60a0016f60c458403f617f6' \
  {
    / structure-version / 1 : 1
    / sequence-number / 2 : 5
    / common / 3 : h'a20254828245466c61736843003401824352414d420004'
                  h'0458818e13a20350fa6b4a53d5ad5fdfbe9de663e4d41f'
                  h'fe04501492af1425695e48bf429b2d51f2ab450c0013a2'
                  h'0b582000112233445566778899aabbccddeeff01234567'
                  h'89abcdeffedcba98765432100c1987d00c0113a20b5820'
                  h'0123456789abcdeffedcba987654321000112233445566'
                  h'778899aabbccddeeff0c1987d001f602f6' \ {
    / components / 2 : h'828245466c61736843003401824352414d4200'
                      h'04' \
    [
      [h'466c617368', h'003401'],
      [h'52414d', h'0004'],
    ],
    / common / 4 : h'8e13a20350fa6b4a53d5ad5fdfbe9de663e4d41ffe'
                  h'04501492af1425695e48bf429b2d51f2ab450c0013'
  }
}

```

```

        h'a20b582000112233445566778899aabbccddeeff01'
        h'23456789abcdeffedcba98765432100c1987d00c01'
        h'13a20b58200123456789abcdeffedcba9876543210'
        h'00112233445566778899aabbccddeeff0c1987d001'
        h'f602f6' \ [
/ set-vars / 19, {
    / vendor-id / 3 : h'fa6b4a53d5ad5fdfbe9de663e4d41f'
                        h'fe'
    / class-id / 4 : h'1492af1425695e48bf429b2d51f2ab45'
},
/ set-component-index / 12, 0,
/ set-vars / 19, {
    / digest / 11 :h'00112233445566778899aabbccddeeff01'
                    h'23456789abcdeffedcba9876543210',
    / size / 12 : 34768
},
/ set-component-index / 12, 1,
/ set-vars / 19, {
    / digest / 11 :h'0123456789abcdeffedcba987654321000'
                    h'112233445566778899aabbccddeeff',
    / size / 12 : 34768
},
/ condition-vendor-id / 1, None,
/ condition-class-id / 2, None,
],
},
/ apply-image / 9 : h'860c0013a106781b687474703a2f2f6578616d70'
                    h'6c652e636f6d2f66696c652e62696e15f6' \ [
    / set-component-index / 12, 0,
    / set-vars / 19, {
        / uri / 6 : http://example.com/file.bin
    },
    / fetch / 21, None,
],
/ load-image / 11 : h'8a0c0003f60c0113a20841f60a0016f6' \ [
    / set-component-index / 12, 0,
    / condition-image / 3, None,
    / set-component-index / 12, 1,
    / set-vars / 19, {
        / unknown / 8 : h'f6'
        / source-component / 10 : 0
    },
    / copy / 22, None,
],
/ run-image / 12 : h'8403f617f6' \ [
    / condition-image / 3, None,
    / run / 23, None,
],

```

```

    }
}

```

Total size of outer wrapper without COSE authentication object: 234

Outer:

```

a201f60358e4a60101020503589ba20254828245466c61736843003401824352414d4200
040458818e13a20350fa6b4a53d5ad5fdfbe9de663e4d41ffe04501492af1425695e48bf
429b2d51f2ab450c0013a20b582000112233445566778899aabbccddee0123456789ab
cdeffedcba98765432100c1987d00c0113a20b58200123456789abcdeffedcba98765432
1000112233445566778899aabbccddee0c1987d001f602f6095825860c0013a106781b
687474703a2f2f6578616d706c652e636f6d2f66696c652e62696e15f60b508a0c0003f6
0c0113a20841f60a0016f60c458403f617f6

```

13.6. Example 5:

Compatibility test, download, installation, and secure boot.

The following JSON shows the intended behaviour of the manifest.

```

{
  "structure-version": 1,
  "sequence-number": 6,
  "common": {
    "common-sequence": [
      {
        "directive-set-var": {
          "vendor-id": "fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe",
          "class-id": "1492af14-2569-5e48-bf42-9b2d51f2ab45"
        }
      },
      { "directive-set-component": 0 },
      {
        "directive-set-var": {
          "digest": "00112233445566778899aabbccddee"
                    "0123456789abcdeffedcba9876543210",
          "size": 34768
        }
      },
      { "directive-set-component": 1 },
      {
        "directive-set-var": {
          "digest": "0123456789abcdeffedcba9876543210"
                    "00112233445566778899aabbccddee",
          "size": 34768
        }
      }
    ],
  },
}

```

```

        { "condition-vendor-id": null },
        { "condition-class-id": null }
    ],
    "components": [
        [
            "ext-Flash",
            78848
        ],
        [
            "Flash",
            1024
        ]
    ]
},
"apply-image": [
    { "directive-set-component": 0 },
    {
        "directive-set-var": {
            "uri": "http://example.com/file.bin"
        }
    },
    { "directive-fetch": null }
],
"load-image": [
    { "directive-set-component": 1 },
    { "condition-not-image": null },
    { "directive-set-component": 0 },
    { "condition-image": null },
    { "directive-set-component": 1 },
    {
        "directive-set-var": {
            "source-index": 0
        }
    },
    { "directive-fetch": null }
],
"run-image": [
    { "directive-set-component": 1 },
    { "condition-image": null },
    { "directive-run": null }
]
}

```

Converted into the SUIT manifest, this produces:

```

{
  / auth object / 1 : None
  / manifest / 3 : h'a60101020603589ea202578282467b1b4595ab2143003401'
}

```

```

h'8245466c6173684200040458818e13a20350fa6b4a53d5ad'
h'5fdfe9de663e4d41ffe04501492af1425695e48bf429b2d'
h'51f2ab450c0013a20b582000112233445566778899aabbcc'
h'ddeeff0123456789abcdeffedcba98765432100c1987d00c'
h'0113a20b58200123456789abcdeffedcba98765432100011'
h'2233445566778899aabbccddee0c1987d001f602f60958'
h'25860c0013a106581b687474703a2f2f6578616d706c652e'
h'636f6d2f66696c652e62696e15f60b528e0c011819f60c00'
h'03f60c0113a10a0015f60c47860c0103f617f6' \
{
  / structure-version / 1 : 1
  / sequence-number / 2 : 6
  / common / 3 : h'a202578282467b1b4595ab21430034018245466c617368'
                  h'4200040458818e13a20350fa6b4a53d5ad5fdfe9de663'
                  h'e4d41ffe04501492af1425695e48bf429b2d51f2ab450c'
                  h'0013a20b582000112233445566778899aabbccddee01'
                  h'23456789abcdeffedcba98765432100c1987d00c0113a2'
                  h'0b58200123456789abcdeffedcba987654321000112233'
                  h'445566778899aabbccddee0c1987d001f602f6' \ {
  / components / 2 : h'8282467b1b4595ab21430034018245466c6173'
                     h'68420004' \
  [
    [h'7b1b4595ab21', h'003401'],
    [h'466c617368', h'0004'],
  ],
  / common / 4 : h'8e13a20350fa6b4a53d5ad5fdfe9de663e4d41ffe'
                  h'04501492af1425695e48bf429b2d51f2ab450c0013'
                  h'a20b582000112233445566778899aabbccddee01'
                  h'23456789abcdeffedcba98765432100c1987d00c01'
                  h'13a20b58200123456789abcdeffedcba9876543210'
                  h'00112233445566778899aabbccddee0c1987d001'
                  h'f602f6' \ [
  / set-vars / 19, {
    / vendor-id / 3 : h'fa6b4a53d5ad5fdfe9de663e4d41f'
                     h'fe'
    / class-id / 4 : h'1492af1425695e48bf429b2d51f2ab45'
  },
  / set-component-index / 12, 0,
  / set-vars / 19, {
    / digest / 11 :h'00112233445566778899aabbccddee01'
                  h'23456789abcdeffedcba9876543210',
    / size / 12 : 34768
  },
  / set-component-index / 12, 1,
  / set-vars / 19, {
    / digest / 11 :h'0123456789abcdeffedcba987654321000'
                  h'112233445566778899aabbccddee01',
    / size / 12 : 34768
  }
}

```

```

        },
        / condition-vendor-id / 1, None,
        / condition-class-id / 2, None,
    ],
},
/ apply-image / 9 : h'860c0013a106581b687474703a2f2f6578616d70'
                    h'6c652e636f6d2f66696c652e62696e15f6' \ [
    / set-component-index / 12, 0,
    / set-vars / 19, {
        / uri / 6 : h'687474703a2f2f6578616d706c652e636f6d2f66'
                    h'696c652e62696e'
    },
    / fetch / 21, None,
],
/ load-image / 11 : h'8e0c011819f60c0003f60c0113a10a0015f6' \ [
    / set-component-index / 12, 1,
    / condition-not-image / 25, None,
    / set-component-index / 12, 0,
    / condition-image / 3, None,
    / set-component-index / 12, 1,
    / set-vars / 19, {
        / source-component / 10 : 0
    },
    / fetch / 21, None,
],
/ run-image / 12 : h'860c0103f617f6' \ [
    / set-component-index / 12, 1,
    / condition-image / 3, None,
    / run / 23, None,
],
}
}

```

Total size of outer wrapper without COSE authentication object: 241

Outer:

```

a201f60358eba60101020603589ea202578282467b1b4595ab21430034018245466c6173
684200040458818e13a20350fa6b4a53d5ad5fdfe9de663e4d41ffe04501492af142569
5e48bf429b2d51f2ab450c0013a20b582000112233445566778899aabbccddeeff012345
6789abcdeffedcba98765432100c1987d00c0113a20b58200123456789abcdeffedcba98
7654321000112233445566778899aabbccddeeff0c1987d001f602f6095825860c0013a1
06581b687474703a2f2f6578616d706c652e636f6d2f66696c652e62696e15f60b528e0c
011819f60c0003f60c0113a10a0015f60c47860c0103f617f6

```


13.7. Example 6:

Compatibility test, 2 images, simultaneous download and installation, and secure boot.

The following JSON shows the intended behaviour of the manifest.

```
{
  "structure-version": 1,
  "sequence-number": 7,
  "common": {
    "common-sequence": [
      {
        "directive-set-var": {
          "vendor-id": "fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe",
          "class-id": "1492af14-2569-5e48-bf42-9b2d51f2ab45"
        }
      },
      { "directive-set-component": 0 },
      {
        "directive-set-var": {
          "digest": "00112233445566778899aabbccddeeff"
                    "0123456789abcdeffedcba9876543210",
          "size": 34768
        }
      },
      { "directive-set-component": 1 },
      {
        "directive-set-var": {
          "digest": "0123456789abcdeffedcba9876543210"
                    "00112233445566778899aabbccddeeff",
          "size": 76834
        }
      },
      { "condition-vendor-id": null },
      { "condition-class-id": null }
    ],
    "components": [
      [
        "Flash",
        78848
      ],
      [
        "Flash",
        132096
      ]
    ]
  }
},
```

```

"apply-image": [
  { "directive-set-component": 0 },
  {
    "directive-set-var": {
      "uri": "http://example.com/file1.bin"
    }
  },
  { "directive-set-component": 1 },
  {
    "directive-set-var": {
      "uri": "http://example.com/file2.bin"
    }
  },
  { "directive-set-component": true },
  { "directive-fetch": null }
],
"run-image": [
  { "directive-set-component": true },
  { "condition-image": null },
  { "directive-set-component": 0 },
  { "directive-run": null }
]
}

```

Converted into the SUIT manifest, this produces:

```

{
  / auth object / 1 : None
  / manifest / 3 : h'a5010102070358a0a20257828245466c6173684300340182'
                    h'45466c617368430004020458838e13a20350fa6b4a53d5ad'
                    h'5fdfbe9de663e4d41ffe04501492af1425695e48bf429b2d'
                    h'51f2ab450c0013a20b582000112233445566778899aabbcc'
                    h'ddeeff0123456789abcdeffedcba98765432100c1987d00c'
                    h'0113a20b58200123456789abcdeffedcba98765432100011'
                    h'2233445566778899aabbccddeeff0c1a00012c2201f602f6'
                    h'09584b8c0c0013a106781c687474703a2f2f6578616d706c'
                    h'652e636f6d2f66696c65312e62696e0c0113a106781c6874'
                    h'74703a2f2f6578616d706c652e636f6d2f66696c65322e62'
                    h'696e0cf515f60c49880cf503f60c0017f6' \
  {
    / structure-version / 1 : 1
    / sequence-number / 2 : 7
    / common / 3 : h'a20257828245466c617368430034018245466c61736843'
                   h'0004020458838e13a20350fa6b4a53d5ad5fdfbe9de663'
                   h'e4d41ffe04501492af1425695e48bf429b2d51f2ab450c'
                   h'0013a20b582000112233445566778899aabbccddeeff01'
                   h'23456789abcdeffedcba98765432100c1987d00c0113a2'
                   h'0b58200123456789abcdeffedcba987654321000112233'

```

```

        h'445566778899aabbccddeeff0c1a00012c2201f602f6'
\ {
  / components / 2 : h'828245466c617368430034018245466c617368'
                        h'43000402' \
  [
    [h'466c617368', h'003401'],
    [h'466c617368', h'000402'],
  ],
  / common / 4 : h'8e13a20350fa6b4a53d5ad5fdfbe9de663e4d41ffe'
                  h'04501492af1425695e48bf429b2d51f2ab450c0013'
                  h'a20b582000112233445566778899aabbccddeeff01'
                  h'23456789abcdeffedcba98765432100c1987d00c01'
                  h'13a20b58200123456789abcdeffedcba9876543210'
                  h'00112233445566778899aabbccddeeff0c1a00012c'
                  h'2201f602f6' \ [
    / set-vars / 19, {
      / vendor-id / 3 : h'fa6b4a53d5ad5fdfbe9de663e4d41f'
                        h'fe'
      / class-id / 4 : h'1492af1425695e48bf429b2d51f2ab45'
    },
    / set-component-index / 12, 0,
    / set-vars / 19, {
      / digest / 11 :h'00112233445566778899aabbccddeeff01'
                    h'23456789abcdeffedcba9876543210',
      / size / 12 : 34768
    },
    / set-component-index / 12, 1,
    / set-vars / 19, {
      / digest / 11 :h'0123456789abcdeffedcba987654321000'
                    h'112233445566778899aabbccddeeff',
      / size / 12 : 76834
    },
    / condition-vendor-id / 1, None,
    / condition-class-id / 2, None,
  ],
},
/ apply-image / 9 : h'8c0c0013a106781c687474703a2f2f6578616d70'
                    h'6c652e636f6d2f66696c65312e62696e0c0113a1'
                    h'06781c687474703a2f2f6578616d706c652e636f'
                    h'6d2f66696c65322e62696e0cf515f6' \ [
  / set-component-index / 12, 0,
  / set-vars / 19, {
    / uri / 6 : http://example.com/file1.bin
  },
  / set-component-index / 12, 1,
  / set-vars / 19, {
    / uri / 6 : http://example.com/file2.bin
  },
},

```

```

        / set-component-index / 12, True,
        / fetch / 21, None,
    ],
    / run-image / 12 : h'880cf503f60c0017f6' \ [
        / set-component-index / 12, True,
        / condition-image / 3, None,
        / set-component-index / 12, 0,
        / run / 23, None,
    ],
}
}

```

Total size of outer wrapper without COSE authentication object: 264

Outer:

```

a201f603590101a5010102070358a0a20257828245466c617368430034018245466c6173
68430004020458838e13a20350fa6b4a53d5ad5fdfe9de663e4d41ffe04501492af1425
695e48bf429b2d51f2ab450c0013a20b582000112233445566778899aabbccddeeff0123
456789abcdeffedcba98765432100c1987d00c0113a20b58200123456789abcdeffedcba
987654321000112233445566778899aabbccddeeff0c1a00012c2201f602f609584b8c0c
0013a106781c687474703a2f2f6578616d706c652e636f6d2f66696c65312e62696e0c01
13a106781c687474703a2f2f6578616d706c652e636f6d2f66696c65322e62696e0cf515
f60c49880cf503f60c0017f6

```

14. IANA Considerations

Several registries will be required for:

- standard Commands
- standard Parameters
- standard Algorithm identifiers
- standard text values

15. Security Considerations

This document is about a manifest format describing and protecting firmware images and as such it is part of a larger solution for offering a standardized way of delivering firmware updates to IoT devices. A more detailed discussion about security can be found in the architecture document [Architecture] and in [Information].

16. Mailing List Information

The discussion list for this document is located at the e-mail address suit@ietf.org [1]. Information on the group and information on how to subscribe to the list is at <https://www1.ietf.org/mailman/listinfo/suit> [2]

Archives of the list can be found at: <https://www.ietf.org/mail-archive/web/suit/current/index.html> [3]

17. 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 Chruscinski
- Andrzej Puzdrowski
- Michael Richardson
- David Brown
- Emmanuel Baccelli

18. References

18.1. 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>>.
- [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>>.

18.2. Informative References

- [Architecture]
Moran, B., "A Firmware Update Architecture for Internet of Things Devices", January 2019, <<https://tools.ietf.org/html/draft-ietf-suit-architecture-02>>.
- [Information]
Moran, B., "Firmware Updates for Internet of Things Devices - An Information Model for Manifests", January 2019, <<https://tools.ietf.org/html/draft-ietf-suit-information-model-02>>.
- [RFC6920] Farrell, S., Kutscher, D., Dannewitz, C., Ohlman, B., Keranen, A., and P. Hallam-Baker, "Naming Things with Hashes", RFC 6920, DOI 10.17487/RFC6920, April 2013, <<https://www.rfc-editor.org/info/rfc6920>>.

18.3. URIs

- [1] <mailto:suit@ietf.org>
- [2] <https://www1.ietf.org/mailman/listinfo/suit>
- [3] <https://www.ietf.org/mail-archive/web/suit/current/index.html>

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