SUIT
Internet-Draft
Intended status: Informational
Expires: July 22, 2021

B. Moran
H. Tschofenig
Arm Limited
D. Brown
Linaro
M. Meriac
Consultant

January 18, 2021

A Firmware Update Architecture for Internet of Things
draft-ietf-suit-architecture-15

Abstract

Vulnerabilities with Internet of Things (IoT) devices have raised the need for a solid and secure firmware update mechanism that is also suitable for constrained devices. Incorporating such update mechanism to fix vulnerabilities, to update configuration settings as well as adding new functionality is recommended by security experts.

This document lists requirements and describes an architecture for a firmware update mechanism suitable for IoT devices. The architecture is agnostic to the transport of the firmware images and associated meta-data.

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 22, 2021.

Table of Contents

1. Introduction

When developing Internet of Things (IoT) devices, one of the most difficult problems to solve is how to update firmware on the device. Once the device is deployed, firmware updates play a critical part in its lifetime, particularly when devices have a long lifetime, are

deployed in remote or inaccessible areas where manual intervention is cost prohibitive or otherwise difficult. Updates to the firmware of an IoT device are done to fix bugs in software, to add new functionality, and to re-configure the device to work in new environments or to behave differently in an already deployed context.

The firmware update process, among other goals, has to ensure that

- The firmware image is authenticated and integrity protected. Attempts to flash a modified firmware image or an image from an unknown source are prevented.

- The firmware image can be confidentiality protected so that attempts by an adversary to recover the plaintext binary can be prevented. Obtaining the firmware is often one of the first steps to mount an attack since it gives the adversary valuable insights into used software libraries, configuration settings and generic functionality (even though reverse engineering the binary can be a tedious process).

This version of the document assumes asymmetric cryptography and a public key infrastructure. Future versions may also describe a symmetric key approach for very constrained devices.

While the standardization work has been informed by and optimised 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. Software update and delivery of arbitrary data, such as configuration information and keys, can equally be managed by manifests.

More details about the security goals are discussed in Section 5 and requirements are described in Section 3.

2. Conventions and Terminology

This document uses the following terms:

- Manifest: The manifest contains meta-data about the firmware image. The manifest is protected against modification and provides information about the author.

- Firmware Image: The firmware image, or 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. Firmware is the more universal term. The terms, firmware image, firmware, and image, are used in this document and are interchangeable.

- Software: The terms "software" and "firmware" are used interchangeably.

- Bootloader: A bootloader is a piece of software that is executed once a microcontroller has been reset. It is responsible for deciding whether to boot a firmware image that is present or whether to obtain and verify a new firmware image. Since the bootloader is a security critical component its functionality may be split into separate stages. Such a multi-stage bootloader may offer very basic functionality in the first stage and resides in ROM whereas the second stage may implement more complex functionality and resides in flash memory so that it can be updated in the future (in case bugs have been found). The exact split of components into the different stages, the number of firmware images stored by an IoT device, and the detailed functionality varies throughout different implementations. A more detailed discussion is provided in Section 8.

- 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 for these types of devices.

- System on Chip (SoC): An SoC is an integrated circuit that integrates all components of a computer, such as CPU, memory, input/output ports, secondary storage, etc.

- Homogeneous Storage Architecture (HoSA): A device that stores all firmware components in the same way, for example in a file system or in flash memory.

- Heterogeneous Storage Architecture (HeSA): A device that stores at least one firmware component differently from the rest, for example a device with an external, updatable radio, or a device with internal and external flash memory.

- Trusted Execution Environments (TEEs): An execution environment that runs alongside of, but is isolated from, an REE.

- 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.

- Trusted applications (TAs): An application component that runs in a TEE.

For more information about TEEs see [I-D.ietf-teep-architecture].

The following entities are used:

- Author: The author is the entity that creates the firmware image. There may be multiple authors in a system either when a device consists of multiple micro-controllers or when the the final firmware image consists of software components from multiple companies.

- 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, if present.

- (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 succesfully perform an update. The terms device and firmware consumer are used interchangably since the firmware consumer is one software component running on an MCU on the device.

- Status Tracker: The status tracker offers device management functionality to retrieve information about the installed firmware on a device and other device characteristics (including free memory and hardware components), to obtain the state of the firmware update cycle the device is currently in, and to trigger the update process. The deployment of status trackers is flexible and they may be used as cloud-based servers, on-premise servers, embedded in edge computing device (such as Internet access gateways or protocol translation gateways), or even in smart phones and tablets. While the IoT device itself runs the client-side of the status tracker it will most likely not run a status tracker itself unless it acts as a proxy for other IoT devices in a protocol translation or edge computing device node. How much functionality a status tracker includes depends on the selected

configuration of the device management functionality and the communication environment it is used in. In a generic networking environment the protocol used between the client and the server-side of the status tracker need to deal with Internet communication challenges involving firewall and NAT traversal. In other cases, the communication interaction may be rather simple. This architecture document does not impose requirements on the status tracker.

- 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 but those entities are often physically separated on different devices for scalability reasons.

- Device Operator: The actor responsible for the day-to-day operation of a fleet of IoT devices.

- Network Operator: The actor responsible for the operation of a network to which IoT devices connect.

In addition to the entities in the list above there is an orthogonal infrastructure with a Trust Provisioning Authority (TPA) distributing trust anchors and authorization permissions to various entities in the system. The TPA may also delegate rights to install, update, enhance, or delete trust anchors and authorization permissions to other parties in the system. This infrastructure overlaps the communication architecture and different deployments may empower certain entities while other deployments may not. For example, in some cases, the Original Design Manufacturer (ODM), which is a company that designs and manufactures a product, may act as a TPA and may decide to remain in full control over the firmware update process of their products.

The terms 'trust anchor' and 'trust anchor store' are defined in [RFC6024]:

- "A trust anchor 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."

- "A trust anchor store 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.

3. Requirements

The firmware update mechanism described in this specification was designed with the following requirements in mind:

- Agnostic to how firmware images are distributed
- Friendly to broadcast delivery
- Use state-of-the-art security mechanisms
- Rollback attacks must be prevented
- High reliability
- Operate with a small bootloader
- Small Parsers
- Minimal impact on existing firmware formats
- Robust permissions
- Diverse modes of operation
- Suitability to software and personalization data

3.1. Agnostic to how firmware images are distributed

Firmware images can be conveyed to devices in a variety of ways, including USB, UART, WiFi, BLE, low-power WAN technologies, etc. and use different protocols (e.g., CoAP, HTTP). The specified mechanism needs to be agnostic to the distribution of the firmware images and manifests.

3.2. Friendly to broadcast delivery

This architecture does not specify any specific broadcast 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.

3.3. Use state-of-the-art security mechanisms

End-to-end security between the author and the device is shown in Section 5.

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.

Integrity protection ensures that no third party can modify the manifest or the firmware image.

For confidentiality protection of the firmware image, it must be done in such a way that every intended recipient can decrypt it. The information that is encrypted individually for each device must maintain friendliness to Content Distribution Networks, bulk storage, and broadcast protocols.

A manifest specification must support different cryptographic algorithms and algorithm extensibility. Due of the nature of unchangeable code in ROM for use with bootloaders the use of post-quantum secure signature mechanisms, such as hash-based signatures [RFC8778], are attractive. These algorithms maintain security in presence of quantum computers.

A mandatory-to-implement set of algorithms will be specified in the manifest specification [I-D.ietf-suit-manifest]}.

3.4. Rollback attacks must be prevented

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.

3.5. High reliability

A power failure at any time must not cause a failure of the device. A failure to validate any part of an update must not cause a failure of the device. One way to achieve this functionality is to provide a minimum of two storage locations for firmware and one bootable

location for firmware. An alternative approach is to use a 2nd stage bootloader with build-in full featured firmware update functionality such that it is possible to return to the update process after power down.

Note: This is an implementation requirement rather than a requirement on the manifest format.

3.6. Operate with a small bootloader

Throughout this document we assume that the bootloader itself is distinct from the role of the firmware consumer and therefore does not manage the firmware update process. This may give the impression that the bootloader itself is a completely separate component, which is mainly responsible for selecting a firmware image to boot.

The overlap between the firmware update process and the bootloader functionality comes in two forms, namely

- First, a bootloader 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 bootloader 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 bootloader may well be the same manifest obtained with the firmware image during the update process (with the severable fields stripped off).

- Second, an IoT device needs a recovery strategy in case the firmware update / boot process fails. The recovery strategy may include storing two or more firmware images on the device or offering the ability to have a second stage bootloader perform the firmware update process again using firmware updates over serial, USB or even wireless connectivity like a limited version of Bluetooth Smart. In the latter case the firmware consumer functionality is contained in the second stage bootloader and requires the necessary functionality for executing the firmware update process, including manifest parsing.

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 risk in reliability.

All information necessary for a device to make a decision about the installation of a firmware update must fit into the available RAM of a constrained IoT device. This prevents flash write exhaustion. This is typically not a difficult requirement to accomplish because there are not other task/processing running while the bootloader is

active (unlike it may be the case when running the application firmware).

Note: This is an implementation requirement.

3.7. Small Parsers

Since parsers are known sources of bugs they must be minimal. Additionally, it must be easy to parse only those fields that are required to validate at least one signature or MAC with minimal exposure.

3.8. Minimal impact on existing firmware formats

The design of the firmware update mechanism must not require changes to existing firmware formats.

3.9. Robust permissions

When a device obtains a monolithic firmware image from a single author without any additional approval steps then 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.

3.10. Operating modes

There are three broad classifications of update operating modes.

- Client-initiated Update
- Server-initiated Update
- Hybrid Update

Client-initiated updates take the form of a firmware consumer on a device proactively checking (polling) for new firmware images.

Server-initiated updates are important to consider because timing of updates may need to be tightly controlled in some high- reliability environments. In this case the status tracker determines what devices qualify for a firmware update. Once those devices have been selected the firmware server distributes updates to the firmware consumers.

Note: This assumes that the status tracker is able to reach the device, which may require devices to keep reachability information at the status tracker up-to-date. This may also require keeping state at NATs and stateful packet filtering firewalls alive.

Hybrid updates are those that require an interaction between the firmware consumer and the status tracker. The status tracker pushes notifications of availability of an update to the firmware consumer, and it then downloads the image from a firmware server as soon as possible.

An alternative view to the operating modes is to consider the steps a device has to go through in the course of an update:

- Notification

- Pre-authorisation

- Dependency resolution

- Download

- Installation

The notification step consists of the status tracker informing the firmware consumer that an update is available. This can be accomplished via polling (client-initiated), push notifications (server-initiated), or more complex mechanisms.

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-based firmware server is client-initiated. Pushing a manifest and firmware image to the transfer to the Package resource of the LwM2M Firmware Update object [LwM2M] is server-initiated.

If the firmware consumer has downloaded a new firmware image and is ready to install it, it may need to wait for a trigger from the status tracker to initiate the installation, may trigger the update automatically, or may go through a more complex decision making process to determine the appropriate timing for an update (such as delaying the update process to a later time when end users are less impacted by the update process).

Installation is the act of processing the payload into a format that the IoT device can recognise and the bootloader is responsible for then booting from the newly installed firmware image.

Each of these steps may require different permissions.

3.11. Suitability to software and personalization data

The work on a standardized manifest format initially focused on the most constrained IoT devices and those devices contain code put together by a single author (although that author may obtain code from other developers, some of it only in binary form).

Later it turns out that other use cases may benefit from a standardized manifest format also for conveying software and even personalization data alongside software. Trusted Execution Environments (TEEs), for example, greatly 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.

To support this wider range of use cases the manifest format should therefore be extensible to convey other forms of payloads as well.

4. Claims

Claims in the manifest offer a way to convey instructions to a device that impact the firmware update process. To have any value the manifest containing those claims must be authenticated and integrity protected. The credential used must be directly or indirectly related to the trust anchor installed at the device by the Trust Provisioning Authority.

The baseline claims for all manifests are described in [I-D.ietf-suit-information-model]. For example, there are:

- Do not install firmware with earlier metadata than the current metadata.
- Only install firmware with a matching vendor, model, hardware revision, software version, etc.
- Only install firmware that is before its best-before timestamp.
- Only allow a firmware installation if dependencies have been met.
- Choose the mechanism to install the firmware, based on the type of firmware it is.

5. Communication Architecture

Figure 1 shows the communication architecture where a firmware image is created by an author, and uploaded to a firmware server. The firmware image/manifest is distributed to the device either in a push or pull manner using the firmware consumer residing on the device. The device operator keeps track of the process using the status tracker. This allows the device operator to know and control what devices have received an update and which of them are still pending an update.

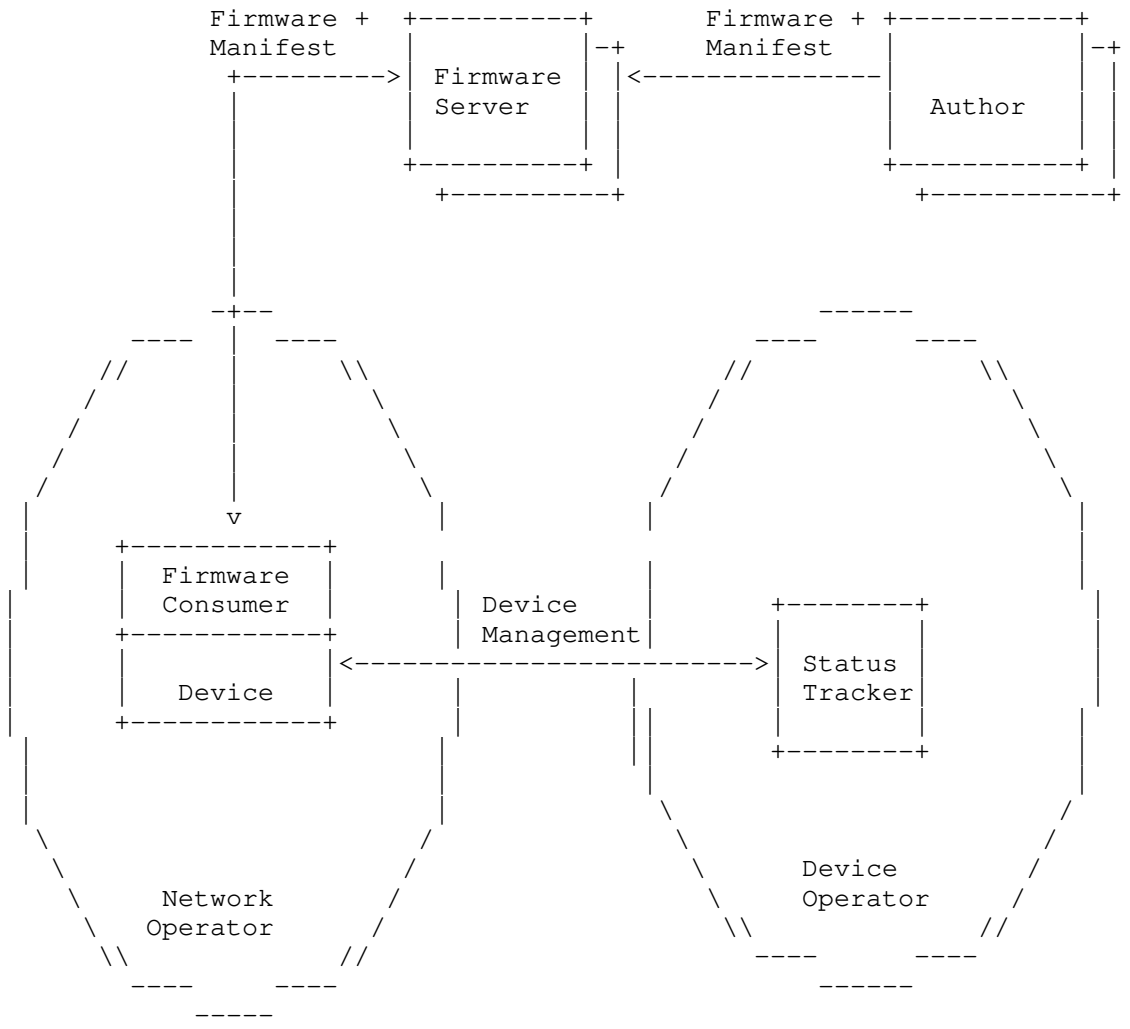


Figure 1: Architecture.

End-to-end security mechanisms are used to protect the firmware image and the manifest although Figure 2 does not show the manifest itself since it may be distributed independently.

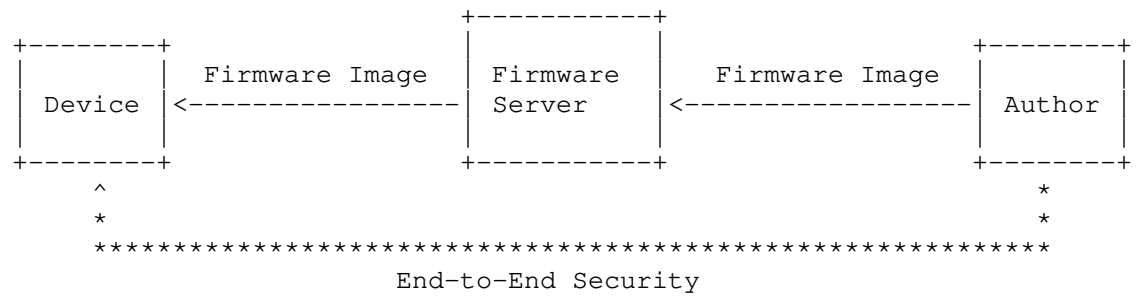


Figure 2: End-to-End Security.

Whether the firmware image and the manifest is pushed to the device or fetched by the device is a deployment specific decision.

The following assumptions are made to allow the firmware consumer to verify the received firmware image and manifest before updating software:

- 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.

- 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 also a duty of the Trust Provisioning Authority.

- 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 deployment specific.

There are different types of delivery modes, which are illustrated based on examples below.

There is an option for embedding a firmware image into a 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 stick or via Bluetooth Smart. Figure 3 shows this delivery mode graphically.

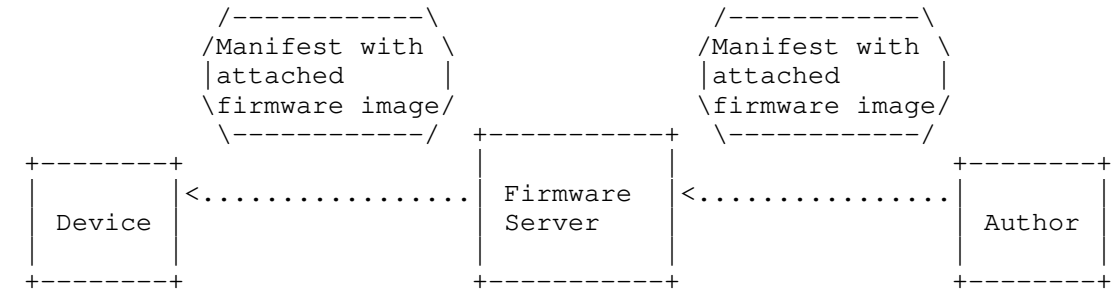


Figure 3: Manifest with attached firmware.

Figure 4 shows an option for remotely updating a device where the device fetches the firmware image from some file server. The manifest itself is delivered independently and provides information about the firmware image(s) to download.

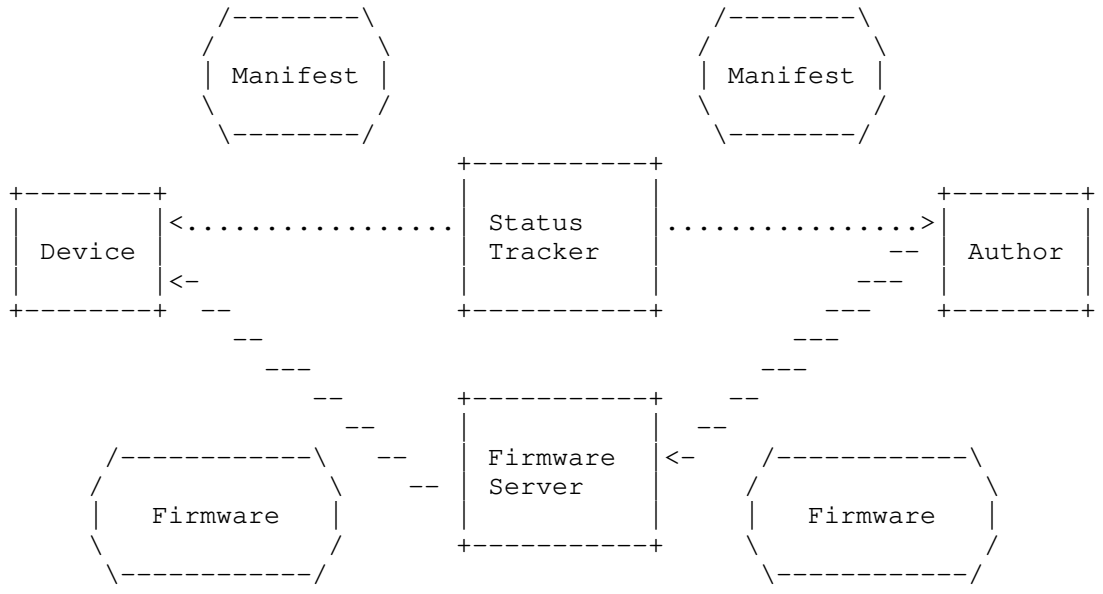


Figure 4: Independent retrieval of the firmware image.

This architecture does not mandate a specific delivery mode but a solution must support both types.

6. Manifest

In order for a device to apply an update, it has to make several decisions about the update:

- Does it trust the author of the update?
- Has the firmware been corrupted?
- Does the firmware update apply to this device?
- Is the update older than the active firmware?
- When should the device apply the update?
- How should the device apply the update?
- What kind of firmware binary is it?
- Where should the update be obtained?
- Where should the firmware be stored?

The manifest encodes the information that devices need in order to make these decisions. It is a data structure that contains the following information:

- information about the device(s) the firmware image is intended to be applied to,
- information about when the firmware update has to be applied,
- information about when the manifest was created,
- dependencies on other manifests,
- pointers to the firmware image and information about the format,
- information about where to store the firmware image,
- cryptographic information, such as digital signatures or message authentication codes (MACs).

The manifest information model is described in [I-D.ietf-suit-information-model].

7. Device Firmware Update Examples

Although these documents attempt to define a firmware update architecture that is applicable to both existing systems, as well as yet-to-be-conceived systems; it is still helpful to consider existing architectures.

7.1. Single CPU SoC

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. These systems either have a single firmware image, or an immutable bootloader that runs a single image. A notable characteristic of these SoCs is that the primary code is generally execute in place (XIP). Combined with the non-relocatable nature of the code, firmware updates need to be done in place.

7.2. 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.

7.3. Dual CPU, shared memory

This configuration has two or more CPUs in a single SoC that share memory (flash and RAM). Generally, they will be a protection mechanism to prevent one CPU from accessing the other's memory. Upgrades in this case will typically be done by one of the CPUs, and is similar to the single CPU with secure mode.

7.4. Dual CPU, other bus

This configuration has two or more 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 a master, 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.

8. Bootloader

More devices today than ever before are being 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 RS232. Updating a device over the Internet requires the device to fetch not only the firmware image but also the manifest. Hence, the following building blocks are necessary for a firmware update solution:

- the Internet protocol stack for firmware downloads (*),
- the capability to write the received firmware image to persistent storage (most likely flash memory) prior to performing the update,
- the ability to unpack, decompress or otherwise process the received firmware image,
- the features to verify an image and a manifest, including digital signature verification or checking a message authentication code,
- a manifest parsing library, and
- integration of the device into a device management server to perform automatic firmware updates and to track their progress.

(*) Because firmware images are often multiple kilobytes, sometimes exceeding one hundred kilobytes, in size for low end IoT devices and even several megabytes large 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.

All these features are most likely offered by the application, i.e. firmware consumer, running on the device (except for basic security algorithms that may run either on a trusted execution environment or on a separate hardware security MCU/module) rather than by the bootloader itself.

Once manifests have been processed and firmware images successfully downloaded and verified the device needs to hand control over to the bootloader. In most cases this requires the MCU to restart. Once

the MCU has initiated a restart, the bootloader takes over control and determines whether the newly downloaded firmware image should be executed.

The boot process is security sensitive because the firmware images may, for example, be stored in off-chip flash memory giving attackers easy access to the image for reverse engineering and potentially also for modifying the binary. The bootloader will therefore have to perform security checks on the firmware image before it can be booted. These security checks by the bootloader happen in addition to the security checks that happened when the firmware image and the manifest were downloaded.

The manifest may have been stored alongside the firmware image to allow re-verification of the firmware image during every boot attempt. Alternatively, secure boot-specific meta-data may have been created by the application after a successful firmware download and verification process. Whether to re-use the standardized manifest format that was used during the initial firmware retrieval process or whether it is better to use a different format for the secure boot-specific meta-data depends on the system design. The manifest format does, however, have the capability to serve also as a building block for secure boot with its severable elements that allow shrinking the size of the manifest by stripping elements that are no longer needed.

If the application image contains the firmware consumer functionality, as described above, then it is necessary that a working image is left on the device. This allows the bootloader to roll back to a working firmware image to execute a firmware download if the bootloader itself does not have enough functionality to fetch a firmware image plus manifest from a firmware server over the Internet. A multi-stage bootloader may soften this requirement at the expense of a more sophisticated boot process.

For a bootloader to offer a secure boot mechanism it needs to provide the following features:

- ability to access security algorithms, such as SHA-256 to compute a fingerprint over the firmware image and a digital signature algorithm.

- access keying material directly or indirectly to utilize the digital signature. The device needs to have a trust anchor store.

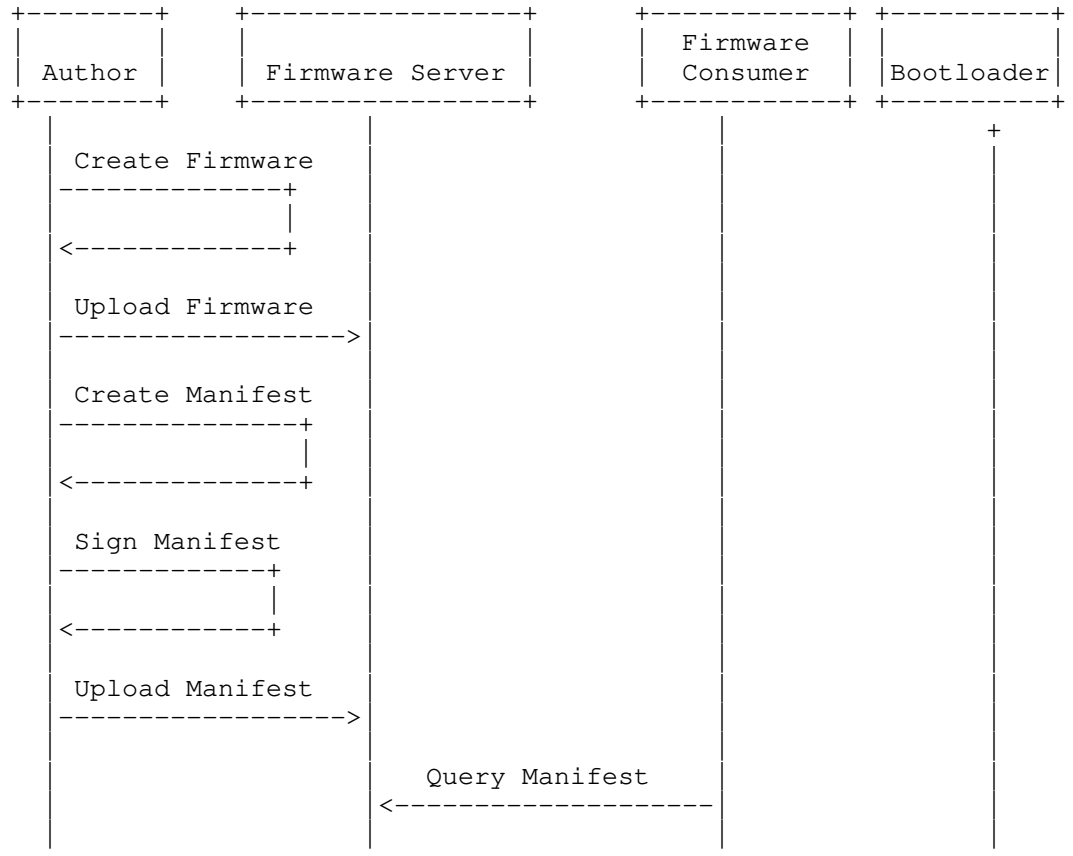
- ability to expose boot process-related data to the application firmware (such as to the device management software). This allows a device management server to determine whether the firmware update has been successful and, if not, what errors occurred.

- to (optionally) offer attestation information (such as measurements).

While the software architecture of the bootloader and its security mechanisms are implementation-specific, the manifest can be used to control the firmware download from the Internet in addition to augmenting secure boot process. These building blocks are highly relevant for the design of the manifest.

9. Example

Figure 5 illustrates an example message flow for distributing a firmware image to a device starting with an author uploading the new firmware to firmware server and creating a manifest. The firmware and manifest are stored on the same firmware server. This setup does not use a status tracker and the firmware consumer component is therefore responsible for periodically checking whether a new firmware image is available for download.



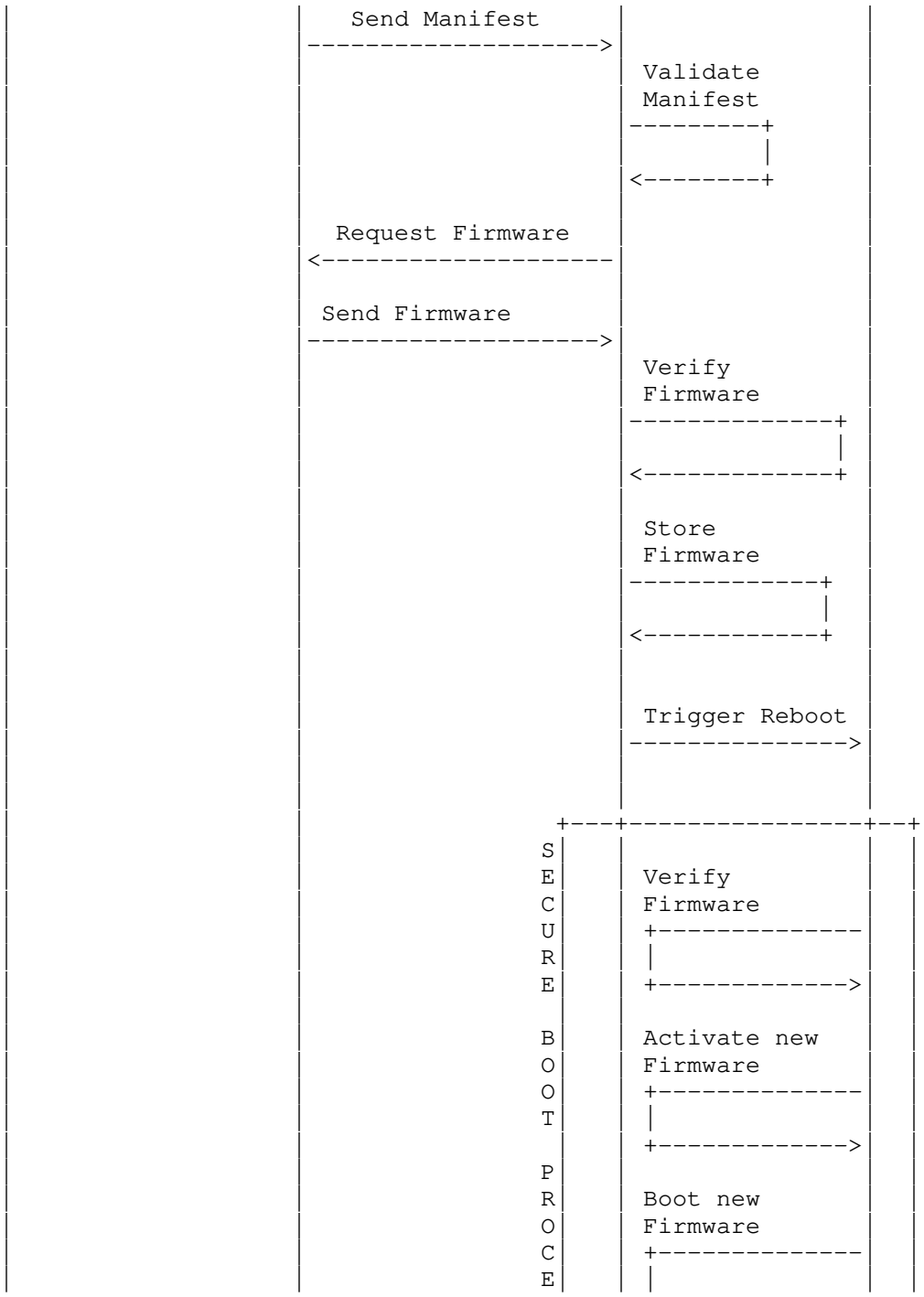
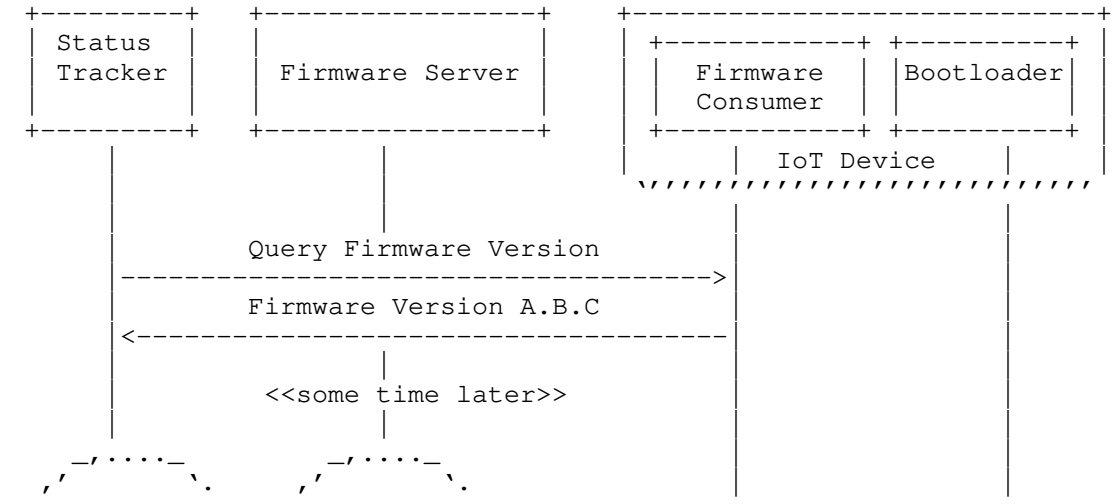

Send Manifest
Validate Manifest
Request Firmware
Send Firmware
Verify Firmware
Store Firmware
Trigger Reboot
SECURE BOOT PROCE
Verify Firmware
Activate new Firmware
Boot new Firmware

Figure 5: First Example Flow for a Firmware Upate.

Figure 6 shows an example follow with the device using a status tracker. For editorial reasons the author publishing the manifest at the status tracker and the firmware image at the firmware server is not shown. Also omitted is the secure boot process following the successful firmware update process.

The exchange starts with the device interacting with the status tracker; the details of such exchange will vary with the different device management systems being used. In any case, the status tracker learns about the firmware version of the devices it manages. In our example, the device under management is using firmware version A.B.C. At a later point in time the author uploads a new firmware along with the manifest to the firmware server and the status tracker, respectively. While there is no need to store the manifest and the firmware on different servers this example shows a common pattern used in the industry. The status tracker may then automatically, based on human intervention or based on a more complex policy decide to inform the device about the newly available firmware image. In our example, it does so by pushing the manifest to the firmware consumer. 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.



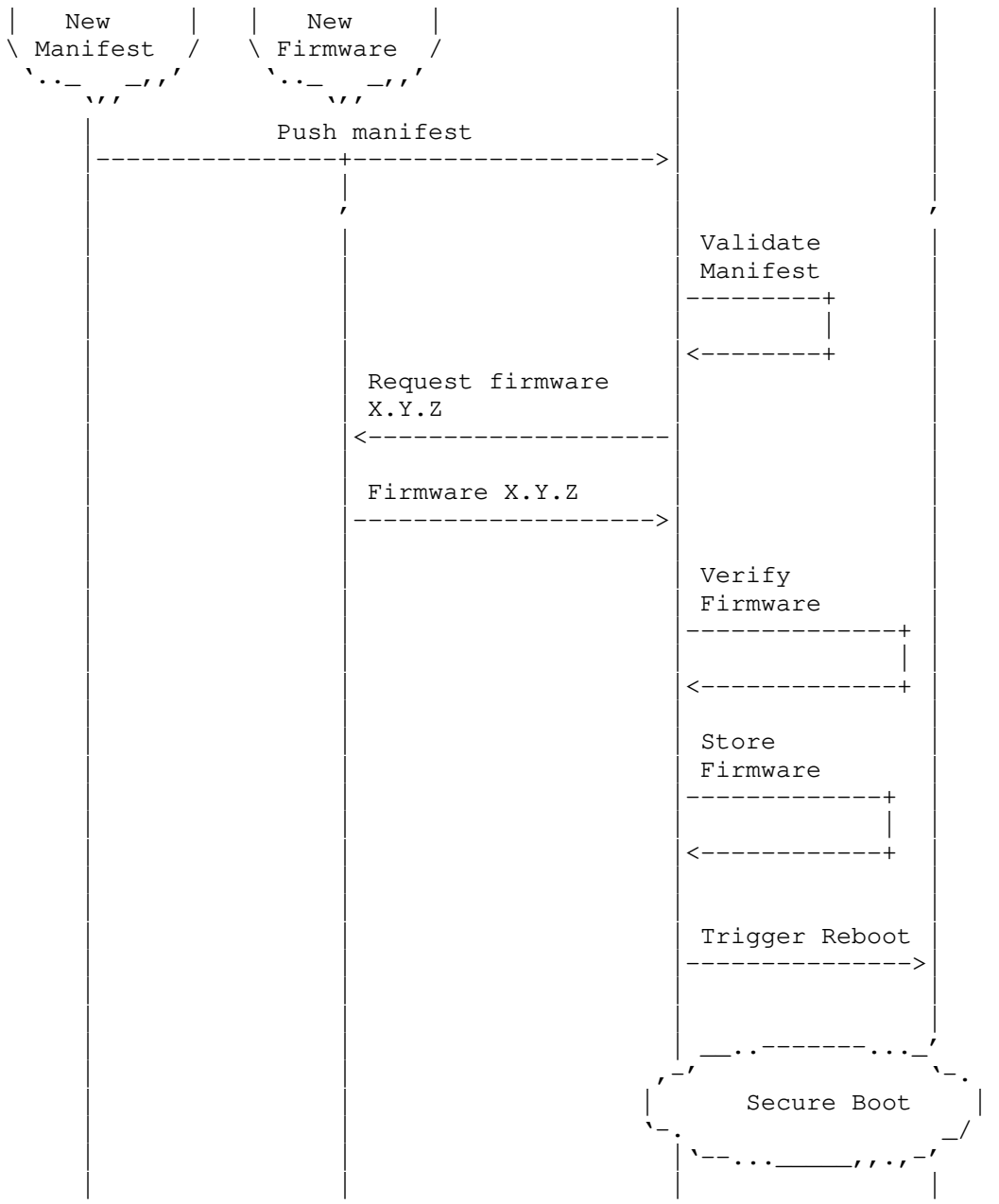


Figure 6: Second Example Flow for a Firmware Upate.

10. IANA Considerations

This document does not require any actions by IANA.

11. Security Considerations

Firmware updates fix security vulnerabilities and are considered to be an important building block in securing IoT devices. Due to the importance of firmware updates for IoT devices the Internet Architecture Board (IAB) organized a 'Workshop on Internet of Things (IoT) Software Update (IOTSU)', which took place at Trinity College Dublin, Ireland on the 13th and 14th of June, 2016 to take a look at the big picture. A report about this workshop can be found at [RFC8240]. A standardized firmware manifest format providing end-to-end security from the author to the device will be specified in a separate document.

There are, however, many other considerations raised during the workshop. Many of them are outside the scope of standardization organizations since they fall into the realm of product engineering, regulatory frameworks, and business models. The following considerations are outside the scope of this document, namely

- installing firmware updates in a robust fashion so that the update does not break the device functionality of the environment this device operates in.

- installing firmware updates in a timely fashion considering the complexity of the decision making process of updating devices, potential re-certification requirements, and the need for user consent to install updates.

- the distribution of the actual firmware update, potentially in an efficient manner to a large number of devices without human involvement.

- energy efficiency and battery lifetime considerations.

- key management required for verifying the digital signature protecting the manifest.

- incentives for manufacturers to offer a firmware update mechanism as part of their IoT products.

12. 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

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

13. Informative References

[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>.

[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>.

[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: May 1, 2021

B. Moran
H. Tschofenig
Arm Limited
H. Birkholz
Fraunhofer SIT

October 28, 2020

An Information Model for Firmware Updates in IoT Devices
draft-ietf-suit-information-model-08

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.

Table of Contents

1. Introduction

The information model describes all the information elements required to secure firmware updates of IoT devices from the threats described in Section 4.1 and enables the user stories captured in Section 4.4. 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. The information model does not define the serialization, encoding, ordering, or structure of information elements, only their semantics.

Because the information model 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 could be considered optional to implement and optional to use, depending on which threats exist in a particular domain of application and which user stories are required. Elements marked as REQUIRED provide baseline security and usability properties that are expected to be required for most applications. Those elements are required to be implemented and used. Elements marked as RECOMMENDED provide important security or usability properties that are needed on most devices. Elements marked as OPTIONAL enable security or usability properties that are useful in some applications.

The definition of some of the information elements include examples that illustrate their semantics and how they are intended to be used.

2. Conventions and 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 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.

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.

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. Manifest Element: Version ID of the manifest structure

An identifier that describes which iteration of the manifest format is contained in the structure.

This element is REQUIRED in order to allow devices to identify the version of the manifest data model that is in use.

3.2. Manifest Element: Monotonic Sequence Number

A monotonically increasing sequence number. For convenience, the monotonic sequence number MAY be a UTC timestamp. This allows global synchronisation of sequence numbers without any additional management. This number MUST be possible to extract with a simple, minimal parser so that code choosing one out of several manifests can choose which is the latest without fully parsing a complex structure.

This element is REQUIRED and is necessary to prevent malicious actors from reverting a firmware update against the policies of the relevant authority.

Implements: REQ.SEC.SEQUENCE (Section 4.3.1)

3.3. Manifest Element: Vendor ID

Vendor IDs must be unique. This is to prevent similarly, or identically named entities from different geographic regions from colliding in their customer's infrastructure. 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.

Vendor ID is not intended to be a human-readable element. It is intended for binary match/mismatch comparison only.

The use of a Vendor ID is RECOMMENDED. It helps to distinguish between identically named products from different vendors.

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

3.3.1. Example: Domain Name-based UUIDs

Vendor A creates a UUID based on their domain name:

vendorId = UUID5(DNS, "vendor-a.com")

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. Manifest Element: Class ID

A device "Class" is a set of different device types that can accept the same firmware update without modification. 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:

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

The Class Identifier UUID SHOULD use the Vendor ID as the name space ID. Other options include version 1 and 4 UUIDs. Classes MAY be more 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.

The use of Class ID is RECOMMENDED. It allows devices to determine applicability of a firmware in an unambiguous way.

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

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:

- vendorId = UUID5(DNS, "vendor-a.com")
- ZclassId = UUID5(vendorId, "Product Z")
- 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:

- vendorId = UUID5(DNS, "vendor-a.com")
- XclassId = UUID5(vendorId, "Product X")
- 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:

- vendorId = UUID5(DNS, "vendor-a.com")
- XclassId = UUID5(vendorId, "Product X")
- YclassId = UUID5(vendorId, "Product Y")
- 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:

- vendorIdA = UUID5(DNS, "vendor-a.com")
- classIdA = UUID5(vendorIdA, "Product A-Unlabelled")
- vendorIdB = UUID5(DNS, "vendor-b.com")
- 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. Manifest Element: Precursor Image Digest Condition

When a precursor image is required by the payload format (for example, differential updates), a precursor image digest condition MUST be present. The precursor image MAY be installed or stored as a candidate.

This element is OPTIONAL to implement.

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

3.6. Manifest Element: Required Image Version List

When a payload applies to multiple versions of a firmware, the required image version list specifies which 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 be applied.

Where an update can only be applied over specific predecessor versions, that version MUST be specified by the Required Image Version List.

This element is OPTIONAL to implement.

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

3.7. Manifest Element: 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. The last valid firmware should not have an expiration time.

This element is OPTIONAL to implement.

Implements: REQ.SEC.EXP (Section 4.3.3)

3.8. Manifest Element: Payload Format

The format of the payload MUST be indicated to devices in an unambiguous way. This element provides a mechanism to describe the payload format, within the signed metadata.

This element is REQUIRED and MUST be present to enable devices to decode payloads correctly.

Implements: REQ.SEC.AUTH.IMG_TYPE (Section 4.3.5), REQ.USE.IMG.FORMAT (Section 4.5.5)

3.9. Manifest Element: 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 algorithm(s) is used and any additional parameters required by the algorithm(s). The representation MAY group Processing Steps together in predefined combinations.

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

Processing steps are RECOMMENDED to implement.

Implements: REQ.USE.IMG.NESTED (Section 4.5.6)

3.10. Manifest Element: 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 and MUST be present to enable devices to store payloads to the correct location.

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:

- "OS"
- "APP"

3.10.2. Example 2: File System

A device supports a full 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. Manifest Element: 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 which part of the storage architecture is targeted by the payload.

This element is OPTIONAL and only necessary in devices with multiple storage subsystems.

N.B. A serialization MAY choose to combine Component Identifier and Storage Location (Section 3.10)

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

3.12. Manifest Element: Resource Indicator

This element provides the information required for the device to acquire the resource. This can be encoded in several ways:

- One URI
- A list of URIs

- A prioritised list of URIs
- A list of signed URIs

This element is OPTIONAL and only needed when the target device does not intrinsically know where to find the payload.

N.B. Devices will typically require URIs.

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

3.13. Manifest Element: 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). 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 to implement and fundamentally necessary to ensure the authenticity and integrity of the payload. Support for more than one digest is OPTIONAL to implement in a recipient device.

Implements: REQ.SEC.AUTHENTIC (Section 4.3.4), REQ.USE.IMG.SELECT (Section 4.5.8)

3.14. Manifest Element: Size

The size of the payload in bytes.

Variable-size storage locations MUST be set to exactly the size listed in this element.

This element is REQUIRED and informs the target device how big of a payload to expect. Without it, devices are exposed to some classes of denial of service attack.

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

3.15. Manifest Envelope Element: Signature

The Signature element MUST contain all the information necessary to cryptographically verify the contents of the manifest against a root of trust. 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.

This element MAY be provided either by the manifest envelope serialization or by another serialization of authentication objects, such as a COSE ([RFC8152]) or CMS ([RFC5652]) signature object. The Signature element MUST support multiple actors and multiple authentication methods. It is NOT REQUIRED for a serialization to authenticate multiple manifests with a single Signature element.

This element is REQUIRED in non-dependency manifests and represents the foundation of all security properties of the manifest. Manifests which are included as dependencies by another manifest SHOULD include a signature so that the recipient can distinguish between different actors with different permissions.

A manifest MUST NOT be considered authenticated by channel security even if it contains only channel information (such as URIs). If the authenticated remote or channel were compromised, the threat actor could induce recipients to execute queries over any accessible network. Where public key operations require too many resources, the recommended authentication mechanism is MAC with a per-device pre-shared key.

Implements: REQ.SEC.AUTHENTIC (Section 4.3.4), REQ.SEC.RIGHTS (Section 4.3.11), REQ.USE.MFST.MULTI_AUTH (Section 4.5.4)

3.16. Manifest Element: Additional installation instructions

Instructions that 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 Element: 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 and enables some user stories.

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

3.18. Manifest Element: Dependencies

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

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

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

3.19. Manifest Element: 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 MAY be included in a decryption step contained in Processing Steps (Section 3.9).

This element is REQUIRED to use for encrypted payloads,

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

3.20. Manifest Element: XIP Address

In order to support 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 to implement.

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

3.21. Manifest Element: 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:

- the source
- the destination
- the destination address

- cryptographic information
- decompression information
- 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 to implement.

Implements: REQ.USE.LOAD (Section 4.5.10)

3.22. Manifest Element: Run-time metadata

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

This element is OPTIONAL to implement.

Implements: REQ.USE.EXEC (Section 4.5.9)

3.23. Manifest Element: Payload

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

This element is OPTIONAL to implement.

Implements: REQ.USE.PAYLOAD (Section 4.5.11)

3.24. Manifest Envelope Element: Delegation Chain

The Signature (Section 3.15) is NOT REQUIRED to cover the delegation chain. The delegation chain offers enhanced authorization functionality via authorization tokens. Each token itself is protected and does not require another layer of protection and because the delegation chain is needed to verify the signature, it must be placed in the Manifest Envelope, rather than the Manifest.

This element is OPTIONAL to implement.

Implements: REQ.USE.DELEGATION (Section 4.5.13)

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.

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:

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

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)

4.2.3.1. 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 does not know where to obtain the payload for an update, it may be redirected to an attacker's server. This would allow an attacker to provide broken 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 threat.

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, 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

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, Device Operator, or Network 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 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)

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 4.3.18), REQ.SEC.MFST.TRUSTED (Section 4.3.19)

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 4.3.20)

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.

Note: This is not a firmware version. 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 with fine granularity that a given update applies to their vendor, model, hardware revision, software revision. Human-readable identifiers are often error-prone in this regard, so unique identifiers SHOULD be used.

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. Devices MAY provide 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: if a firmware will not be updated again, for example if a business stops issuing updates to a device. The last valid firmware should not have an expiration time.

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 authenticated. 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 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 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 (which may be independent of format) 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), Storage Location (Section 3.10)

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 Resource Location

The location where a target should find a payload MUST be authenticated.

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

Implemented by: Resource 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 3.19)

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

It MUST be possible to encrypt part or all of the manifest. This may be accomplished with either transport encryption or with at-rest encryption.

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

Implemented by: External 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 SHOULD 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)

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)

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

Manifests SHOULD be parsed and examined prior to deployment to validate that their contents have not been modified during creation and signing.

Mitigates: THREAT.MFST.MODIFICATION (Section 4.2.17)

4.3.19. 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)

4.3.20. 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)

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:

- Use a table of hashes to ensure that each block of the payload is validate before writing.
- Do not report progress.
- Pre-cache the update, but do not install.
- Install the pre-cached update matching this manifest.
- 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:

- URIs (Section 3.12): this allows the Device Operator to direct devices to their own infrastructure in order to reduce network load.
- Conditions: this allows the Device Operator to pose additional constraints on the installation of the manifest.
- Directives (Section 3.16): this allows the Device Operator to add more instructions such as time of installation.
- 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: USER_STORY.OVERRIDE (Section 4.4.3), REQ.USE.MFST.COMPONENT (Section 4.5.3)

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 4.5.3)

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 4.5.4), 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 4.5.5)

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

As a firmware author, I want to prevent confidential information from being disclosed during firmware updates. It is assumed that channel security or at-rest encryption is adequate to protect the manifest itself against information disclosure.

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 Resource Indicator (Section 3.12).

Satisfied by: REQ.USE.IMG.FORMAT (Section 4.5.5), REQ.USE.IMG.NESTED (Section 4.5.6), REQ.USE.MFST.OVERRIDE_REMOTE (Section 4.5.2)

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 4.5.7)

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 4.5.8)

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 4.5.9)

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 4.5.10)

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 encryption keys, configuration information, public keys, authorization tokens, or X.509 certificates.

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

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 4.5.12)

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 4.5.13)

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)

4.5. Usability Requirements

The following usability requirements satisfy the user stories listed above.

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

It MUST be possible for a manifest author to place ALL information required to process an update in the manifest.

For example: Information about which precursor image is required for a differential update MUST be placed in the manifest, not in the differential compression header.

For example: Information about an installation-time confirmation system that must be used to allow the installation to proceed.

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.OVERRIDE_REMOTE: Override Remote Resource Location

It MUST be possible 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 3.17)

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

It MUST be possible 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 effectively means 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 Manifest Element: Dependencies, StorageIdentifier, ComponentIdentifier

4.5.3.1. Example 1: Multiple Microcontrollers

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

4.5.3.2. 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.

4.5.3.3. 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.

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

It MUST be possible to authenticate a manifest multiple times 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)

4.5.5. 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:

- Binary
- Executable and Linkable Format (ELF)
- Differential
- Compressed
- Packed configuration
- Intel HEX
- 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)

4.5.6. 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)

4.5.7. 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)

4.5.8. 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 3.20)

4.5.9. REQ.USE.EXEC: Executable Manifest

It MUST be possible to describe an executable system with a manifest on both Execute-In-Place microcontrollers and on complex operating systems. This requires the manifest to specify the digest of each statically linked dependency. 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 3.22)

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

It MUST be possible to specify additional metadata for load time processing of a payload, such as cryptographic information, load-address, and compression algorithm.

N.B. load comes before exec/boot.

Satisfies: USER_STORY.EXEC.DECOMPRESS (Section 4.4.12)

Implemented by: Load-time metadata (Section 3.21)

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

It MUST be possible to place 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, wrapped encryption keys, configuration information, public keys, authorization tokens, or X.509 certificates.

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 4.3.20)), 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 4.3.20).

Satisfies: USER_STORY.MFST.IMG (Section 4.4.13)

Implemented by: Payload (Section 3.23)

4.5.12. REQ.USE.PARSE: Simple Parsing

The structure of the manifest MUST be simple to parse, without need for a general-purpose parser.

Satisfies: USER_STORY.MFST.PARSE (Section 4.4.14)

Implemented by: N/A

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

Any manifest format MUST enable the delivery of a key claim with, but not authenticated by, a manifest. This key claim 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 3.24)

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 Waehlisch, Max Groening, Daniel Petry, Gaetan Harter, Ralph Hamm, Steve Patrick, Fabio Utzig, Paul Lambert, Benjamin Kaduk, 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.

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-14 (work in progress), October 2020.

[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>.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <https://www.rfc-editor.org/info/rfc5652>.

[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>.

7.2. Informative References

[STRIDE] Microsoft, "The STRIDE Threat Model", May 2018, <https://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx>.

Authors' Addresses

Brendan Moran
Arm Limited

EMail: Brendan.Moran@arm.com

Hannes Tschofenig
Arm Limited

EMail: hannes.tschofenig@gmx.net

Henk Birkholz
Fraunhofer SIT

EMail: henk.birkholz@sit.fraunhofer.de

SUIT
Internet-Draft
Intended status: Standards Track
Expires: June 11, 2021

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

December 08, 2020

A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest
draft-ietf-suit-manifest-11

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on June 11, 2021.

Table of Contents

1. Introduction

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

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

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

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

- Meet the requirements defined in [I-D.ietf-suit-information-model].

- Simple to parse on a constrained node

- Simple to process on a constrained node

- Compact encoding

- Comprehensible by an intermediate system

- Expressive enough to enable advanced use cases on advanced nodes

- Extensible

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

- a Firmware Author to reason about releasing a firmware.

- a Network Operator to reason about compatibility of a firmware.

- a Device Operator to reason about the impact of a firmware.

- the Device Operator to manage distribution of firmware to devices.

- a Plant Manager to reason about timing and acceptance of firmware updates.

- a device to reason about the authority & authenticity of a firmware prior to installation.

- a device to reason about the applicability of a firmware.

- a device to reason about the installation of a firmware.

- a device to reason about the authenticity & encoding of a firmware at boot.

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

It is assumed that the reader is familiar with the high-level firmware update architecture [I-D.ietf-suit-architecture] and the threats, requirements, and user stories in [I-D.ietf-suit-information-model].

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

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

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

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

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

2. Conventions and Terminology

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

Additionally, the following terminology is used throughout this document:

- SUIT: Software Update for the Internet of Things, also the IETF working group for this standard.

- Payload: A piece of information to be delivered. Typically Firmware for the purposes of SUIT.

- Resource: A piece of information that is used to construct a payload.

- Manifest: A manifest is a bundle of metadata about the firmware for an IoT device, where to find the firmware, and the devices to which it applies.

- Envelope: A container with the manifest, an authentication wrapper with cryptographic information protecting the manifest, authorization information, and severable elements (see: TBD).

- Update: One or more manifests that describe one or more payloads.

- Update Authority: The owner of a cryptographic key used to sign updates, trusted by Recipients.

- Recipient: The system, typically an IoT device, that receives and processes a manifest.

- Manifest Processor: A component of the Recipient that consumes Manifests and executes the commands in the Manifest.

- Component: An updatable logical block of the Firmware, Software, configuration, or data of the Recipient.

- Component Set: A group of interdependent Components that must be updated simultaneously.

- Command: A Condition or a Directive.

- Condition: A test for a property of the Recipient or its Components.

- Directive: An action for the Recipient to perform.

- Trusted Invocation: A process by which a system ensures that only trusted code is executed, for example secure boot or launching a Trusted Application.

- A/B images: Dividing a Recipient’s storage into two or more bootable images, at different offsets, such that the active image can write to the inactive image(s).

- Record: The result of a Command and any metadata about it.

- Report: A list of Records.

- Procedure: The process of invoking one or more sequences of commands.

- Update Procedure: A procedure that updates a Recipient by fetching dependencies and images, and installing them.

- Invocation Procedure: A procedure in which a Recipient verifies dependencies and images, loading images, and invokes one or more image.

- Software: Instructions and data that allow a Recipient to perform a useful function.

- Firmware: Software that is typically changed infrequently, stored in nonvolatile memory, and small enough to apply to [RFC7228] Class 0-2 devices.

- Image: Information that a Recipient uses to perform its function, typically firmware/software, configuration, or resource data such as text or images. Also, a Payload, once installed is an Image.

- Slot: One of several possible storage locations for a given Component, typically used in A/B image systems

- Abort: An event in which the Manifest Processor immediately halts execution of the current Procedure. It creates a Record of an error condition.

3. How to use this Document

This specification covers five aspects of firmware update:

- Section 4 describes the device constraints, use cases, and design principles that informed the structure of the manifest.

- Section 5 gives a general overview of the metadata structure to inform the following sections

- Section 6 describes what actions a Manifest processor should take.

- Section 7 describes the process of creating a Manifest.

- Section 8 specifies the content of the Envelope and the Manifest.

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

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

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

4. Background

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

4.1. IoT Firmware Update Constraints

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

- limited processing power and storage may require a simple representation of metadata.

- bandwidth constraints may require firmware compression or partial update support.

- bootloader complexity constraints may require simple selection between two bootable images.

- small internal storage may require external storage support.

- multiple microcontrollers may require coordinated update of all applications.

- large storage and complex functionality may require parallel update of many software components.

- extra information may need to be conveyed in the manifest in the earlier stages of the device lifecycle before those data items are stripped when the manifest is delivered to a constrained device.

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

4.2. SUIT Workflow Model

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

- Compatibility must be checked before any other operation is performed.
- All dependency manifests should be present before any payload is fetched.
- In some applications, payloads must be fetched and validated prior to installation.

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

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

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

1. Verify the signature of the manifest.
2. Verify the applicability of the manifest.
3. 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).

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

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. Metadata Structure Overview

This section provides a high level overview of the manifest structure. The full description of the manifest structure is in Section 8.6

The manifest is structured from several key components:

1. The Envelope (see Section 5.1) contains Delegation Chains, the Authentication Block, the Manifest, any Severable Elements, and any Integrated Payloads or Dependencies.

2. Delegation Chains (see Section 5.2) allow a Recipient to work from one of its Trust Anchors to an authority of the Authentication Block.

3. The Authentication Block (see Section 5.3) contains a list of signatures or MACs of the manifest..

4. The Manifest (see Section 5.4) contains all critical, non-severable metadata that the Recipient requires. It is further broken down into:

    1. Critical metadata, such as sequence number.

    2. Common metadata, including lists of dependencies and affected components.

3. Command sequences, directing the Recipient how to install and use the payload(s).

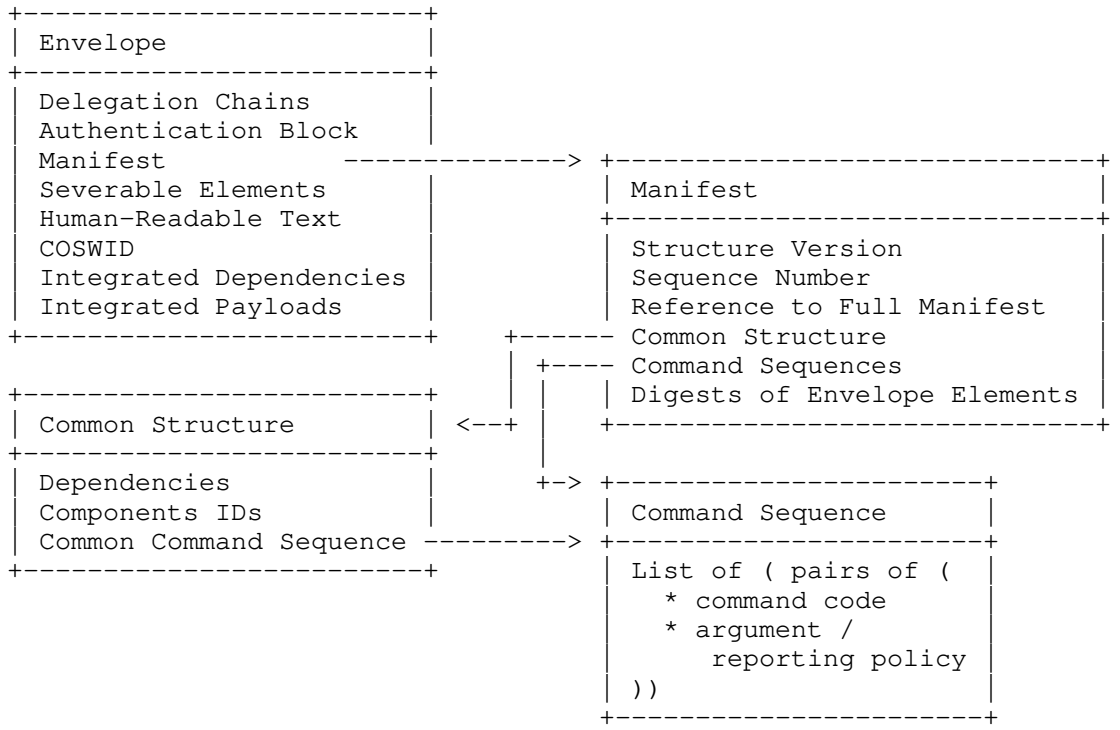
4. Integrity check values for severable elements.

5. Severable elements (see Section 5.5).

6. Integrated dependencies (see Section 5.6).

7. Integrated payloads (see Section 5.6).

The diagram below illustrates the hierarchy of the Envelope.



5.1. Envelope

The SUIT Envelope is a container that encloses Delegation Chains, the Authentication Block, the Manifest, any Severable Elements, and any integrated payloads or dependencies. The Envelope is used instead of conventional cryptographic envelopes, such as COSE_Envelope because it allows modular processing, severing of elements, and integrated payloads in a way that would add substantial complexity with existing

solutions. See Appendix C.1 for a description of the reasoning for this.

See Section 8.2 for more detail.

5.2. Delegation Chains

Delegation Chains allow a Recipient to establish a chain of trust from a Trust Anchor to the signer of a manifest by validating delegation claims. Each delegation claim is a [RFC8392] CBOR Web Tokens (CWTs). The first claim in each list is signed by a Trust Anchor. Each subsequent claim in a list is signed by the public key claimed in the preceding list element. The last element in each list claims a public key that can be used to verify a signature in the Authentication Block (Section 5.3).

See Section 8.3 for more detail.

5.3. Authentication Block

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

- COSE_Sign_Tagged
- COSE_Sign1_Tagged
- COSE_Mac_Tagged
- COSE_Mac0_Tagged

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

See Section 8.4 for more detail.

5.4. Manifest

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

See Section 8.6 for more detail.

5.4.1. Critical Metadata

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

See Section 8.6.1, Section 8.6.2, and Section 8.6.3 for more detail.

5.4.2. Common

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

See Section 8.7.2, and Section 8.7.2.1 for more detail.

5.4.3. Command Sequences

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

Command sequences are broken up into three groups: Common Command Sequence (see Section 5.4.2), update commands, and secure boot commands.

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

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

Command Sequences are grouped into these sets to ensure that there is common coordination between dependencies and dependents on when to execute each command.

See Section 8.7.3 for more detail.

5.4.4. Integrity Check Values

To enable Section 5.5, there needs to be a mechanism to verify integrity of any metadata outside the manifest. Integrity Check Values are used to verify the integrity of metadata that is not contained in the manifest. This MAY include Severable Command Sequences, Concise Software Identifiers (CoSWID [I-D.ietf-sacm-coswid]), or Text data. Integrated Dependencies and Integrated Payloads are integrity-checked using Command Sequences, so they do not have Integrity Check Values present in the Manifest.

See Section 8.7.8 for more detail.

5.4.5. Human-Readable Text

Text is typically a Severable Element (Section 5.5). It contains all the text that describes the update. Because text is explicitly for human consumption, it is all grouped together so that it can be Severed easily. The text section has space both for describing the manifest as a whole and for describing each individual component.

See Section 8.6.4 for more detail.

5.5. Severable Elements

Severable Elements are elements of the Envelope (Section 5.1) that have Integrity Check Values (Section 5.4.4) in the Manifest (Section 5.4).

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

- A management system severs the Text and CoSWID sections before sending an Envelope to a constrained Recipient, which saves Recipient bandwidth.

- A Recipient severs the Installation section after installing the Update, which saves storage space.

See Section 8.8 for more detail.

5.6. Integrated Dependencies and Payloads

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

- When an update is delivered via a comparatively unconstrained medium, such as a removable mass storage device, it may be beneficial to bundle updates into single files.

- When a manifest requires encryption, it must be referenced as a dependency, so a trivial manifest may be used to enclose the encrypted manifest. The encrypted manifest may be contained in the dependent manifest's envelope.

- When a manifest transports a small payload, such as an encrypted key, that payload may be placed in the manifest's envelope.

See Section 7.9.1, Section 8.5 for more detail.

6. Manifest Processor Behavior

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

6.1. Manifest Processor Setup

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

- Whenever the manifest processor can choose between several manifests, it MUST select the latest valid, authentic manifest.

- If the latest valid, authentic manifest fails, it MAY select the next latest valid, authentic manifest, according to application-specific policy.

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

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

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

Selecting an older manifest in the event of failure of the latest valid manifest is a robustness mechanism that is necessary for supporting the requirements in [I-D.ietf-suit-architecture], section 3.5. It may not be appropriate for all applications. In particular Trusted Execution Environments MAY require a failure to invoke a new installation, rather than a rollback approach. See [I-D.ietf-suit-information-model], Section 4.2.1 for more discussion on the security considerations that apply to rollback.

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

6.2. Required Checks

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

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

conferred by that authenticity. Where a device supports only one level of access, it MAY choose to skip signature verification of dependencies, since they are referenced by digest. Where a device supports more than one trusted party, it MAY choose to defer the verification of signatures of dependencies until the list of affected components is known so that it can skip redundant signature verifications. For example, a dependency signed by the same author as the dependent does not require a signature verification. Similarly, if the signer of the dependent has full rights to the device, according to the ACL, then no signature verification is necessary on the dependency.

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

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

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

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

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

If a dependency is specified, then the manifest processor MUST perform the following checks:

1. At the beginning of each section in the dependent: all previous sections of each dependency have been executed.

2. At the end of each section in the dependent: The corresponding section in each dependency has been executed.

If the interpreter does not support dependencies and a manifest specifies a dependency, then the interpreter MUST reject the manifest.

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

1. have sufficient permissions imparted by their signatures

2. specify a digest and a payload for every Component in the Component Set.

The single dependent manifest is sometimes called a Root Manifest.

6.2.1. Minimizing Signature Verifications

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

- The Authentication Block contains a COSE_Sign_Tagged or COSE_Sign1_Tagged

- The Recipient receives manifests over an unauthenticated channel, exposing it to more inauthentic or incompatible manifests, and

- The Recipient has a power budget that makes signature verification undesirable

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

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

Alternatively, a Recipient MAY rely on network infrastructure to filter inapplicable manifests.

6.3. Interpreter Fundamental Properties

The interpreter has a small set of design goals:

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

2. Executing a Trusted Invocation MUST either result in an error, or an invoked image.

3. Executing the same manifest on multiple Recipients MUST result in the same system state.

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

6.4. Abstract Machine Description

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

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

Conditions MUST NOT have any side-effects other than informing the interpreter of success or failure. The Interpreter does not Abort if the Soft Failure flag (Section 8.7.5.23) is set when a Condition reports failure.

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

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

```
current := components\[component-index\]
    if component-index is not false
    else dependencies\[dependency-index\]
```

As a result, Set Component Index is described as current := components[arg]. The actual operation performed for Set Component Index is described by the following pseudocode, however, because of the definition of current (above), these are semantically equivalent.

```
component-index := arg
dependency-index := false
```

Similarly, Set Dependency Index is semantically equivalent to current := dependencies[arg]

The following table describes the behavior of each command. "params" represents the parameters for the current component or dependency. Most commands operate on either a component or a dependency. Setting the Component Index clears the Dependency Index. Setting the Dependency Index clears the Component Index.

Command Name	Semantic of the Operation
Check Vendor Identifier	assert(binary-match(current, current.params[vendor-id]))
Check Class Identifier	assert(binary-match(current, current.params[class-id]))
Verify Image	assert(binary-match(digest(current), current.params[digest]))
Set Component Index	current := components[arg]
Override Parameters	current.params[k] := v for-each k,v in arg
Set Dependency Index	current := dependencies[arg]
Set Parameters	current.params[k] := v if not k in params for-each k,v in arg

Process Dependency	exec(current[common]); exec(current[current-segment])
Run	run(current)
Fetch	store(current, fetch(current.params[uri]))
Use Before	assert(now() < arg)
Check Component Offset	assert(offsetof(current) == arg)
Check Device Identifier	assert(binary-match(current, current.params[device-id]))
Check Image Not Match	assert(not binary-match(digest(current), current.params[digest]))
Check Minimum Battery	assert(battery >= arg)
Check Update Authorized	assert(isAuthorized())
Check Version	assert(version_check(current, arg))
Abort	assert(0)
Try Each	try-each-done if exec(seq) is not error for-each seq in arg
Copy	store(current, current.params[src-component])
Swap	swap(current, current.params[src-component])
Wait For Event	until event(arg), wait
Run Sequence	exec(arg)
Run with Arguments	run(current, arg)

6.5. Special Cases of Component Index and Dependency Index

Component Index and Dependency Index can each take on one of three types:

1. Integer

2. Array of integers

3. True

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

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

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

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

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

6.6. Serialized Processing Interpreter

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

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

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

6.7. Parallel Processing Interpreter

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

Under each of these scenarios the parallel processing MUST halt until all issued commands have completed:

- Set Parameters.
- Override Parameters.
- Set Strict Order = True.
- Set Dependency Index.
- Set Component Index.

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

6.8. Processing Dependencies

As described in Section 6.2, each manifest must invoke each of its dependencies sections from the corresponding section of the dependent. Any changes made to parameters by the dependency persist in the dependent.

When a Process Dependency command is encountered, the interpreter loads the dependency identified by the Current Dependency Index. The interpreter first executes the common-sequence section of the identified dependency, then it executes the section of the dependency that corresponds to the currently executing section of the dependent.

If the specified dependency does not contain the current section, Process Dependency succeeds immediately.

The Manifest Processor MUST also support a Dependency Index of True, which applies to every dependency, as described in Section 6.5

The interpreter also performs the checks described in Section 6.2 to ensure that the dependent is processing the dependency correctly.

6.9. Multiple Manifest Processors

When a system has multiple security domains, each domain might require independent verification of authenticity or security policies. Security domains might be divided by separation technology such as Arm TrustZone, Intel SGX, or another TEE technology. Security domains might also be divided into separate processors and memory spaces, with a communication interface between them.

For example, an application processor may have an attached communications module that contains a processor. The communications module might require metadata signed by a specific Trust Authority for regulatory approval. This may be a different Trust Authority than the application processor.

When there are two or more security domains (see [I-D.ietf-teep-architecture]), a manifest processor might be required in each. The first manifest processor is the normal manifest processor as described for the Recipient in Section 6.4. The second manifest processor only executes sections when the first manifest processor requests it. An API interface is provided from the second manifest processor to the first. This allows the first manifest processor to request a limited set of operations from the second. These operations are limited to: setting parameters, inserting an Envelope, invoking a Manifest Command Sequence. The second manifest processor declares a prefix to the first, which tells the first manifest processor when it should delegate to the second. These rules are enforced by underlying separation of privilege infrastructure, such as TEEs, or physical separation.

When the first manifest processor encounters a dependency prefix, that informs the first manifest processor that it should provide the second manifest processor with the corresponding dependency Envelope.

This is done when the dependency is fetched. The second manifest processor immediately verifies any authentication information in the dependency Envelope. When a parameter is set for any component that matches the prefix, this parameter setting is passed to the second manifest processor via an API. As the first manifest processor works through the Procedure (set of command sequences) it is executing, each time it sees a Process Dependency command that is associated with the prefix declared by the second manifest processor, it uses the API to ask the second manifest processor to invoke that dependency section instead.

This mechanism ensures that the two or more manifest processors do not need to trust each other, except in a very limited case. When parameter setting across security domains is used, it must be very carefully considered. Only parameters that do not have an effect on security properties should be allowed. The dependency manifest MAY control which parameters are allowed to be set by using the Override Parameters directive. The second manifest processor MAY also control which parameters may be set by the first manifest processor by means of an ACL that lists the allowed parameters. For example, a URI may be set by a dependent without a substantial impact on the security properties of the manifest.

7. Creating Manifests

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

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

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

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

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

NOTE: *A digest MUST always be set using Override Parameters, since this prevents a less-privileged dependent from replacing the digest.*

7.1. Compatibility Check Template

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

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

- Set Component Index directive (see Section 8.7.7.1)
- Set Parameters directive (see Section 8.7.7.5) for Vendor ID and Class ID (see Section 8.7.5)
- Check Vendor Identifier condition (see Section 8.7.5.2)
- Check Class Identifier condition (see Section 8.7.5.2)

7.2. Trusted Invocation Template

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

The following commands are placed into the common sequence:

- Set Component Index directive (see Section 8.7.7.1)
- Override Parameters directive (see Section 8.7.7.6) for Image Digest and Image Size (see Section 8.7.5)

Then, the run sequence contains the following commands:

- Set Component Index directive (see Section 8.7.7.1)
- Check Image Match condition (see Section 8.7.6.2)
- Run directive (see Section 8.7.7.12)

7.3. Component Download Template

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

The following commands are placed into the common sequence:

- Set Component Index directive (see Section 8.7.7.1)

- Override Parameters directive (see Section 8.7.7.6) for Image Digest and Image Size (see Section 8.7.5)

Then, the install sequence contains the following commands:

- Set Component Index directive (see Section 8.7.7.1)

- Set Parameters directive (see Section 8.7.7.5) for URI (see Section 8.7.5.13)

- Fetch directive (see Section 8.7.7.7)

- Check Image Match condition (see Section 8.7.6.2)

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

Optionally, the Set Parameters directive in the install sequence MAY also contain Encryption Info (see Section 8.7.5.10), Compression Info (see Section 8.7.5.11), or Unpack Info (see Section 8.7.5.12) to perform simultaneous download and decryption, decompression, or unpacking, respectively.

7.4. Install Template

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

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

Then, the install sequence contains the following commands:

- Set Component Index directive (see Section 8.7.7.1)

- Set Parameters directive (see Section 8.7.7.5) for Source Component (see Section 8.7.5.14)

- Copy directive (see Section 8.7.7.9)

- Check Image Match condition (see Section 8.7.6.2)

7.5. Install and Transform Template

The goal of the Install and Transform template is to use an image already stored in an identified component to decompress, decrypt, or unpack at time of installation.

This template is typically used with the Component Download template, however a modification to that template is required: all Component Download operations are moved from the common sequence and the install sequence to the fetch sequence. The Component Download template targets a download component identifier, while the Install and Transform template uses an install component identifier. In-place unpacking, decompression, and decryption is complex and vulnerable to power failure. Therefore, these identifiers SHOULD be different; in-place installation SHOULD NOT be used without establishing guarantees of robustness to power failure.

The following commands are placed into the common sequence:

- Set Component Index directive for install component identifier (see Section 8.7.7.1)
- Override Parameters directive (see Section 8.7.7.6) for Image Digest and Image Size (see Section 8.7.5)

Then, the install sequence contains the following commands:

- Set Component Index directive for install component identifier (see Section 8.7.7.1)
- Set Parameters directive (see Section 8.7.7.5) for:
  - Source Component for download component identifier (see Section 8.7.5.14)
  - Encryption Info (see Section 8.7.5.10)
  - Compression Info (see Section 8.7.5.11)
  - Unpack Info (see Section 8.7.5.12)
- Copy directive (see Section 8.7.7.9)
- Check Image Match condition (see Section 8.7.6.2)

7.6. Integrated Payload Template

The goal of the Integrated Payload template is to install a payload that is included in the manifest envelope. It is identical to the Component Download template (Section 7.3) except that it places an added restriction on the URI passed to the Set Parameters directive.

An implementer MAY choose to place a payload in the envelope of a manifest. The payload envelope key MAY be a positive or negative integer. The payload envelope key MUST NOT be a value between 0 and 24 and it MUST NOT be used by any other envelope element in the manifest. The payload MUST be serialized in a bstr element.

The URI for a payload enclosed in this way MUST be expressed as a fragment-only reference, as defined in [RFC3986], Section 4.4. The fragment identifier is the stringified envelope key of the payload. For example, an envelope that contains a payload a key 42 would use a URI "#42", key -73 would use a URI "#-73".

7.7. Load from Nonvolatile Storage Template

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

The following commands are placed into the load sequence:

- Set Component Index directive (see Section 8.7.7.1)
- Set Parameters directive (see Section 8.7.7.5) for Component Index (see Section 8.7.5)
- Copy directive (see Section 8.7.7.9)

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

7.8. Load & Decompress from Nonvolatile Storage Template

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

The following commands are placed into the load sequence:

- Set Component Index directive (see Section 8.7.7.1)

- Set Parameters directive (see Section 8.7.7.5) for Source Component Index and Compression Info (see Section 8.7.5)

- Copy directive (see Section 8.7.7.9)

This template is similar to Section 7.7 but additionally performs decompression. Hence, the only difference is in setting the Compression Info parameter.

This template can be modified for decryption or unpacking by adding Decryption Info or Unpack Info to the Set Parameters directive.

7.9. Dependency Template

The goal of the Dependency template is to obtain, verify, and process a dependency manifest as appropriate.

The following commands are placed into the dependency resolution sequence:

- Set Dependency Index directive (see Section 8.7.7.2)

- Set Parameters directive (see Section 8.7.7.5) for URI (see Section 8.7.5)

- Fetch directive (see Section 8.7.7.7)

- Check Image Match condition (see Section 8.7.6.2)

- Process Dependency directive (see Section 8.7.7.4)

Then, the validate sequence contains the following operations:

- Set Dependency Index directive (see Section 8.7.7.2)

- Check Image Match condition (see Section 8.7.6.2)

- Process Dependency directive (see Section 8.7.7.4)

NOTE: Any changes made to parameters in a dependency persist in the dependent.

7.9.1. Composite Manifests

An implementer MAY choose to place a dependency's envelope in the envelope of its dependent. The dependent envelope key for the dependency envelope MUST NOT be a value between 0 and 24 and it MUST NOT be used by any other envelope element in the dependent manifest.

The URI for a dependency enclosed in this way MUST be expressed as a fragment-only reference, as defined in [RFC3986], Section 4.4. The fragment identifier is the stringified envelope key of the dependency. For example, an envelope that contains a dependency at key 42 would use a URI "#42", key -73 would use a URI "#-73".

7.10. Encrypted Manifest Template

The goal of the Encrypted Manifest template is to fetch and decrypt a manifest so that it can be used as a dependency. To use an encrypted manifest, create a plaintext dependent, and add the encrypted manifest as a dependency. The dependent can include very little information.

The following operations are placed into the dependency resolution block:

- Set Dependency Index directive (see Section 8.7.7.2)
- Set Parameters directive (see Section 8.7.7.5) for
  - URI (see Section 8.7.5)
  - Encryption Info (see Section 8.7.5)
- Fetch directive (see Section 8.7.7.7)
- Check Image Match condition (see Section 8.7.6.2)
- Process Dependency directive (see Section 8.7.7.4)

Then, the validate block contains the following operations:

- Set Dependency Index directive (see Section 8.7.7.2)
- Check Image Match condition (see Section 8.7.6.2)
- Process Dependency directive (see Section 8.7.7.4)

A plaintext manifest and its encrypted dependency may also form a composite manifest (Section 7.9.1).

7.11. A/B Image Template

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

The following commands are placed in the common block:

- Set Component Index directive (see Section 8.7.7.1)
- Try Each
  - o First Sequence:
    - * Override Parameters directive (see Section 8.7.7.6, Section 8.7.5) for Offset A
    - * Check Offset Condition (see Section 8.7.6.5)
    - * Override Parameters directive (see Section 8.7.7.6) for Image Digest A and Image Size A (see Section 8.7.5)
  - o Second Sequence:
    - * Override Parameters directive (see Section 8.7.7.6, Section 8.7.5) for Offset B
    - * Check Offset Condition (see Section 8.7.6.5)
    - * Override Parameters directive (see Section 8.7.7.6) for Image Digest B and Image Size B (see Section 8.7.5)

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

- Set Component Index directive (see Section 8.7.7.1)
- Try Each
  - o First Sequence:
    - * Override Parameters directive (see Section 8.7.7.6, Section 8.7.5) for Offset A
    - * Check Offset Condition (see Section 8.7.6.5)
    - * Set Parameters directive (see Section 8.7.7.6) for URI A (see Section 8.7.5)
  - o Second Sequence:

  * Override Parameters directive (see Section 8.7.7.6, Section 8.7.5) for Offset B

  * Check Offset Condition (see Section 8.7.6.5)

  * Set Parameters directive (see Section 8.7.7.6) for URI B (see Section 8.7.5)

- Fetch

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

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

8. Metadata Structure

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

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

8.1. Encoding Considerations

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

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

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

8.2. Envelope

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

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

8.3. Delegation Chains

The suit-delegation element MAY carry one or more CBOR Web Tokens (CWTs) [RFC8392], with [RFC8747] cnf claims. They can be used to perform enhanced authorization decisions. The CWTs are arranged into a list of lists. Each list starts with a CWT authorized by a Trust Anchor, and finishes with a key used to authenticate the Manifest (see Section 8.4). This allows an Update Authority to delegate from a long term Trust Anchor, down through intermediaries, to a delegate without any out-of-band provisioning of Trust Anchors or intermediary keys.

A Recipient MAY choose to cache intermediaries and/or delegates. If an Update Distributor knows that a targeted Recipient has cached some intermediaries or delegates, it MAY choose to strip any cached intermediaries or delegates from the Delegation Chains in order to reduce bandwidth and energy.

8.4. Authenticated Manifests

The suit-authentication-wrapper contains a list containing a Section 10 and one or more cryptographic authentication wrappers for the Manifest. These are implemented as COSE_Mac_Tagged or COSE_Sign_Tagged blocks. Each of these blocks contains a SUIT_Digest of the Manifest. This enables modular processing of the manifest. The COSE_Mac_Tagged and COSE_Sign_Tagged blocks are described in RFC 8152 [RFC8152]. The suit-authentication-wrapper MUST come before any element in the SUIT_Envelope, except for the OPTIONAL suit-delegation, regardless of canonical encoding of CBOR. All validators MUST reject any SUIT_Envelope that begins with any element other than a suit-authentication-wrapper or suit-delegation.

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

A signing application MUST verify the suit-manifest element against the SUIT_Digest prior to signing.

8.5. Encrypted Manifests

To use an encrypted manifest, it must be a dependency of a plaintext manifest. This allows fine-grained control of what information is accessible to intermediate systems for the purposes of management, while still preserving the confidentiality of the manifest contents. This also means that a Recipient can process an encrypted manifest in the same way as an encrypted payload, allowing code reuse.

A template for using an encrypted manifest is covered in Encrypted Manifest Template (Section 7.10).

8.6. Manifest

The manifest contains:

- a version number (see Section 8.6.1)
- a sequence number (see Section 8.6.2)
- a reference URI (see Section 8.6.3)
- a common structure with information that is shared between command sequences (see Section 8.7.2)
- one or more lists of commands that the Recipient should perform (see Section 8.7.3)
- a reference to the full manifest (see Section 8.6.3)
- human-readable text describing the manifest found in the SUIT_Envelope (see Section 8.6.4)
- a Concise Software Identifier (CoSWID) found in the SUIT_Envelope (see Section 8.7.1)

The CoSWID, Text section, or any Command Sequence of the Update Procedure (Dependency Resolution, Image Fetch, Image Installation) can be either a CBOR structure or a SUIT_Digest. In each of these cases, the SUIT_Digest provides for a severable element. Severable elements are RECOMMENDED to implement. In particular, the human-readable text SHOULD be severable, since most useful text elements occupy more space than a SUIT_Digest, but are not needed by the Recipient. 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 has been severed. The key used for a severable element is the same in the SUIT_Manifest and in the

SUIT_Envelope so that a Recipient can easily identify the correct data in the envelope. See Section 8.7.8 for more detail.

8.6.1. suit-manifest-version

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

8.6.2. suit-manifest-sequence-number

The suit-manifest-sequence-number is a monotonically increasing anti-rollback counter. It also helps Recipients 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. For convenience, an implementer MAY use a UTC timestamp in seconds as the sequence number. suit-manifest-sequence-number is REQUIRED to implement.

8.6.3. suit-reference-uri

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

8.6.4. suit-text

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

- integer => text

- SUIT_Component_Identifier => map

Each SUIT_Component_Identifier => map entry contains a map of integer => text values. All SUIT_Component_Identifiers present in suit-text MUST also be present in suit-common (Section 8.7.2) or the suit-common of a dependency.

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

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

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

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

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

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

suit-text is OPTIONAL to implement.

8.7. text-version-required

suit-text-version-required is used to represent a version-based dependency on suit-parameter-version as described in Section 8.7.5.18 and Section 8.7.6.8. To describe a version dependency, a Manifest Author SHOULD populate the suit-text map with a SUIT_Component_Identifier key for the dependency component, and place in the corresponding map a suit-text-version-required key with a free text expression that is representative of the version constraints placed on the dependency. This text SHOULD be expressive enough that a device operator can be expected to understand the dependency. This is a free text field and there are no specific formatting rules.

By way of example only, to express a dependency on a component "['x', 'y']", where the version should be any v1.x later than v1.2.5, but not v2.0 or above, the author would add the following structure to the suit-text element. Note that this text is in cbor-diag notation.

```
[h'78',h'79'] : {
    7 : ">=1.2.5,<2"
}
```

8.7.1. suit-coswid

suit-coswid contains a Concise Software Identifier (CoSWID) as defined in [I-D.ietf-sacm-coswid]. This element SHOULD be made severable so that it can be discarded by the Recipient or an intermediary if it is not required by the Recipient.

suit-coswid typically requires no processing by the Recipient. However all Recipients MUST NOT fail if a suit-coswid is present.

8.7.2. suit-common

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

suit-dependencies is a list of Section 8.7.2.1 blocks that specify manifests that must be present before the current manifest can be processed. suit-dependencies is OPTIONAL to implement.

suit-components is a list of SUIT_Component_Identifier (Section 8.7.2.2) blocks that specify the component identifiers that will be affected by the content of the current manifest. suit-

components is REQUIRED to implement; at least one manifest in a dependency tree MUST contain a suit-components block.

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 Recipient identity and image digests when they are conditional (see Section 8.7.7.3 and Section 7.11 for more information on conditional sequences). suit-common-sequence is RECOMMENDED to implement. It is REQUIRED if the optimizations described in Section 6.2.1 will be used. Whenever a parameter or Try Each command is required by more than one Command Sequence, placing that parameter or commamd in suit-common-sequence results in a smaller encoding.

8.7.2.1. Dependencies

SUIT_Dependency specifies a manifest that describes a dependency of the current manifest. The Manifest is identified, but the Recipient should expect an Envelope when it acquires the dependency. This is because the Manifest is the one invariant element of the Envelope, where other elements may change by countersigning, adding authentication blocks, or severing elements.

The suit-dependency-digest specifies the dependency manifest uniquely by identifying a particular Manifest structure. This is identical to the digest that would be present as the payload of any suit-authentication-block in the dependency's Envelope. The digest is calculated over the Manifest structure instead of the COSE Sig_structure or Mac_structure. 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 (see Section 8.7.2.2). 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 Recipients without those Recipients needing consistent component hierarchy. This element is OPTIONAL for Recipients to implement.

A dependency prefix can be used with a component identifier. This allows complex systems to understand where dependencies need to be applied. The dependency prefix can be used in one of two ways. The

first simply prepends the prefix to all Component Identifiers in the dependency.

A dependency prefix can also be used to indicate when a dependency manifest needs to be processed by a secondary manifest processor, as described in Section 6.9.

8.7.2.2. SUIT_Component_Identifier

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

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

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

8.7.3. 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: 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 to implement and to use when suit-dependencies is present.

2. Payload Fetch: suit-payload-fetch is a SUIT_Command_Sequence to execute in order to obtain a payload. Some manifests may include these actions in the suit-install section instead if they operate in a streaming installation mode. This is particularly relevant for constrained devices without any temporary storage for staging the update. suit-payload-fetch is OPTIONAL to implement.

3. Payload Installation: suit-install is a SUIT_Command_Sequence to execute in order to install a payload. Typical actions include verifying a payload stored in temporary storage, copying a staged payload from temporary storage, and unpacking a payload. suit-install is OPTIONAL to implement.

4. Image Validation: suit-validate is a SUIT_Command_Sequence to execute in order to validate that the result of applying the update is correct. Typical actions involve image validation and manifest validation. suit-validate is REQUIRED to implement. If the manifest contains dependencies, one process-dependency invocation per dependency or one process-dependency invocation targeting all dependencies SHOULD be present in validate.

5. Image Loading: suit-load is a SUIT_Command_Sequence to execute in order to prepare a payload for execution. Typical actions include copying an image from permanent storage into RAM, optionally including actions such as decryption or decompression. suit-load is OPTIONAL to implement.

6. Run or Boot: 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 invocable manifest or the "process dependencies" directive for any dependents of the invocable manifest. suit-run is OPTIONAL to implement.

Goals 1,2,3 form the Update Procedure. Goals 4,5,6 form the Invocation Procedure.

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

Lists of commands are constructed from two kinds of element:

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

2. Directives that MUST be executed.

Each condition is composed of:

1. A command code identifier

2. A SUIT_Reporting_Policy (Section 8.7.4)

Each directive is composed of:

1. A command code identifier

2. An argument block or a SUIT_Reporting_Policy (Section 8.7.4)

Argument blocks are consumed only by flow-control directives:

- Set Component/Dependency Index

- Set/Override Parameters

- Try Each

- Run Sequence

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

Many conditions and directives apply to a given component, and these generally grouped together. Therefore, a special command to set the current component index is provided with a matching command to set the current dependency index. This index is a numeric index into the Component Identifier tables defined at the beginning of the manifest. For the purpose of setting the index, the two Component Identifier tables are considered to be concatenated together.

To facilitate optional conditions, a special directive, suit-directive-try-each (Section 8.7.7.3), is provided. It runs several new lists of conditions/directives, one after another, that are contained as an argument to the directive. By default, it assumes that a failure of a condition should not indicate a failure of the update/invocation, but a parameter is provided to override this behavior. See suit-parameter-soft-failure (Section 8.7.5.23).

8.7.4. Reporting Policy

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

Policy	Description
suit-send-record-on-success	Record when the command succeeds
suit-send-record-on-failure	Record when the command fails
suit-send-sysinfo-success	Add system information when the command succeeds
suit-send-sysinfo-failure	Add system information when the command fails

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

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

The information elements provided to the Reporting Engine are:

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

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

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

This specification does not define a particular format of Records or Reports. This specification only defines hints to the Reporting Engine for which Records it should aggregate into the Report. The

Reporting Engine MAY choose to ignore these hints and apply its own policy instead.

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

8.7.5. SUIT_Parameters

Many conditions and directives require additional information. That information is contained within parameters that can be set in a consistent way. This allows reduction of manifest size and replacement of parameters from one manifest to the next.

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

The defined manifest parameters are described below.

Name	CDDL Structure	Reference
Vendor ID	suit-parameter-vendor-identifier	Section 8.7.5.3
Class ID	suit-parameter-class-identifier	Section 8.7.5.4
Device ID	suit-parameter-device-identifier	Section 8.7.5.5
Image Digest	suit-parameter-image-digest	Section 8.7.5.6
Image Size	suit-parameter-image-size	Section 8.7.5.7
Use Before	suit-parameter-use-before	Section 8.7.5.8
Component Offset	suit-parameter-component-offset	Section 8.7.5.9
Encryption Info	suit-parameter-encryption-info	Section 8.7.5.10

Compression Info	suit-parameter-compression-info	Section 8.7.5.11
Unpack Info	suit-parameter-unpack-info	Section 8.7.5.12
URI	suit-parameter-uri	Section 8.7.5.13
Source Component	suit-parameter-source-component	Section 8.7.5.14
Run Args	suit-parameter-run-args	Section 8.7.5.15
Minimum Battery	suit-parameter-minimum-battery	Section 8.7.5.16
Update Priority	suit-parameter-update-priority	Section 8.7.5.17
Version	suit-parameter-version	Section 8.7.5.18
Wait Info	suit-parameter-wait-info	Section 8.7.5.19
URI List	suit-parameter-uri-list	Section 8.7.5.20
Fetch Arguments	suit-parameter-fetch-arguments	Section 8.7.5.21
Strict Order	suit-parameter-strict-order	Section 8.7.5.22
Soft Failure	suit-parameter-soft-failure	Section 8.7.5.23
Custom	suit-parameter-custom	Section 8.7.5.24

CBOR-encoded object parameters are still wrapped in a bstr. This is because it allows a parser that is aggregating parameters to reference the object with a single pointer and traverse it without understanding the contents. This is important for modularization and division of responsibility within a pull parser. The same

consideration does not apply to Directives because those elements are invoked with their arguments immediately

8.7.5.1. CBOR PEN UUID Namespace Identifier

The CBOR PEN UUID Namespace Identifier is constructed as follows:

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

```
D8 DE                # tag(111)
   45                # bytes(5)
      2B 06 01 04 01 # X.690 Clause 8.19
#    1.3  6  1  4  1  show component encoding
```

Computing a type 5 UUID from these produces:

```
NAMESPACE_CBOR_PEN = UUID5(NAMESPACE_OID, h'D86F452B06010401')
NAMESPACE_CBOR_PEN = 08cfcc43-47d9-5696-85b1-9c738465760e
```

8.7.5.2. Constructing UUIDs

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

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

Identifiers are used for compatibility checks. They MUST NOT be used as assertions of identity. They are evaluated by identifier conditions (Section 8.7.6.1).

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

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

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

1. Hardware model/revision

2. OS

3. WiFi module model/revision

4. Application

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

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

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

The RECOMMENDED method to create a vendor ID is:

Vendor ID = UUID5(DNS_PREFIX, vendor domain name)

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

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

If the Vendor ID is a CBOR PEN (see Section 8.7.5.3), the RECOMMENDED method to create a Class ID is:

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

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

- Model number.

- Hardware revision.

- Bootloader version (for immutable bootloaders).

8.7.5.3. suit-parameter-vendor-identifier

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

- A Private Enterprise Number
- A byte string containing a UUID ([RFC4122])

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

8.7.5.4. suit-parameter-class-identifier

A RFC 4122 UUID representing the class of the device or component. The UUID is encoded as a 16 byte bstr, containing the raw bytes of the UUID. It MUST be constructed as described in Section 8.7.5.2

8.7.5.5. suit-parameter-device-identifier

A RFC 4122 UUID representing the specific device or component. The UUID is encoded as a 16 byte bstr, containing the raw bytes of the UUID. It MUST be constructed as described in Section 8.7.5.2

8.7.5.6. suit-parameter-image-digest

A fingerprint computed over the component itself, encoded in the SUIT_Digest Section 10 structure. The SUIT_Digest is wrapped in a bstr, as required in Section 8.7.5.

8.7.5.7. suit-parameter-image-size

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

8.7.5.8. suit-parameter-use-before

An expiry date for the use of the manifest encoded as the positive integer number of seconds since 1970-01-01. Implementations that use this parameter MUST use a 64-bit internal representation of the integer.

8.7.5.9. suit-parameter-component-offset

This parameter sets the offset in a component. Some components support multiple possible Slots (offsets into a storage area). This parameter describes the intended Slot to use, identified by its

offset into the component's storage area. This offset MUST be encoded as a positive integer.

8.7.5.10. 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.

8.7.5.11. suit-parameter-compression-info

SUIT_Compression_Info defines any information that is required for a Recipient to perform decompression operations. SUIT_Compression_Info is a map containing this data. The only element defined for the map in this specification is the suit-compression-algorithm. This document defines the following suit-compression-algorithm's: ZLIB [RFC1950], Brotli [RFC7932], and ZSTD [I-D.kucherawy-rfc8478bis].

Additional suit-compression-algorithm's can be registered through the IANA-maintained registry. If such a format requires more data than an algorithm identifier, one or more new elements MUST be introduced by specifying an element for SUIT_Compression_Info-extensions.

8.7.5.12. suit-parameter-unpack-info

SUIT_Unpack_Info defines the information required for a Recipient to interpret a packed format. This document defines the use of the following binary encodings: Intel HEX [HEX], Motorola S-record [SREC], Executable and Linkable Format (ELF) [ELF], and Common Object File Format (COFF) [COFF].

Additional packing formats can be registered through the IANA-maintained registry.

8.7.5.13. suit-parameter-uri

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

8.7.5.14. suit-parameter-source-component

This parameter sets the source component to be used with either suit-directive-copy (Section 8.7.7.9) or with suit-directive-swap (Section 8.7.7.13). The current Component, as set by suit-directive-set-component-index defines the destination, and suit-parameter-source-component defines the source.

8.7.5.15. suit-parameter-run-args

This parameter contains an encoded set of arguments for suit-directive-run (Section 8.7.7.10). The arguments MUST be provided as an implementation-defined bstr.

8.7.5.16. suit-parameter-minimum-battery

This parameter sets the minimum battery level in mWh. This parameter is encoded as a positive integer. Used with suit-condition-minimum-battery (Section 8.7.6.6).

8.7.5.17. suit-parameter-update-priority

This parameter sets the priority of the update. This parameter is encoded as an integer. It is used along with suit-condition-update-authorized (Section 8.7.6.7) to ask an application for permission to initiate an update. This does not constitute a privilege inversion because an explicit request for authorization has been provided by the Update Authority in the form of the suit-condition-update-authorized command.

Applications MAY define their own meanings for the update priority. For example, critical reliability & vulnerability fixes MAY be given negative numbers, while bug fixes MAY be given small positive numbers, and feature additions MAY be given larger positive numbers, which allows an application to make an informed decision about whether and when to allow an update to proceed.

8.7.5.18. suit-parameter-version

Indicates allowable versions for the specified component. Allowable versions can be specified, either with a list or with range matching. This parameter is compared with version asserted by the current component when suit-condition-version (Section 8.7.6.8) is invoked. The current component may assert the current version in many ways, including storage in a parameter storage database, in a metadata object, or in a known location within the component itself.

The component version 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.

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.

Versions SHOULD be provided as follows:

1. The first integer represents the major number. This indicates breaking changes to the component.

2. The second integer represents the minor number. This is typically reserved for new features or large, non-breaking changes.

3. The third integer is the patch version. This is typically reserved for bug fixes.

4. The fourth integer is the build number.

Where Alpha (-3), Beta (-2), and Release Candidate (-1) are used, they are inserted as a negative number between Minor and Patch numbers. This allows these releases to compare correctly with final releases. For example, Version 2.0, RC1 should be lower than Version 2.0.0 and higher than any Version 1.x. By encoding RC as -1, this works correctly: [2,0,-1,1] compares as lower than [2,0,0]. Similarly, beta (-2) is lower than RC and alpha (-3) is lower than RC.

8.7.5.19. suit-parameter-wait-info

suit-directive-wait (Section 8.7.7.11) directs the manifest processor to pause until a specified event occurs. The suit-parameter-wait-info encodes the parameters needed for the directive.

The exact implementation of the pause is implementation-defined. For example, this could be done by blocking on a semaphore, registering an event handler and suspending the manifest processor, polling for a notification, or aborting the update entirely, then restarting when a notification is received.

suit-parameter-wait-info is encoded as a map of wait events. When ALL wait events are satisfied, the Manifest Processor continues. The wait events currently defined are described in the following table.

Name	Encoding	Description
suit-wait-event-authorization	int	Same as suit-parameter-update-priority
suit-wait-event-power	int	Wait until power state
suit-wait-event-network	int	Wait until network state
suit-wait-event-other-device-version	See below	Wait for other device to match version
suit-wait-event-time	uint	Wait until time (seconds since 1970-01-01)
suit-wait-event-time-of-day	uint	Wait until seconds since 00:00:00
suit-wait-event-time-of-day-utc	uint	Wait until seconds since 00:00:00 UTC
suit-wait-event-day-of-week	uint	Wait until days since Sunday
suit-wait-event-day-of-week-utc	uint	Wait until days since Sunday UTC

suit-wait-event-other-device-version reuses the encoding of suit-parameter-version-match. It is encoded as a sequence that contains

an implementation-defined bstr identifier for the other device, and a list of one or more SUIT_Parameter_Version_Match.

8.7.5.20. suit-parameter-uri-list

Indicates a list of URIs from which to fetch a resource. The URI list is encoded as a list of text string, in priority order. CBOR Tag 32 is not used because the meaning of the text string is unambiguous in this context. The Recipient should attempt to fetch the resource from each URI in turn, ruling out each, in order, if the resource is inaccessible or it is otherwise undesirable to fetch from that URI. suit-parameter-uri-list is consumed by suit-directive-fetch-uri-list (Section 8.7.7.8).

8.7.5.21. suit-parameter-fetch-arguments

An implementation-defined set of arguments to suit-directive-fetch (Section 8.7.7.7). Arguments are encoded in a bstr.

8.7.5.22. suit-parameter-strict-order

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

See Section 6.7 for behavioral description of Strict Order.

8.7.5.23. suit-parameter-soft-failure

When executing a command sequence inside suit-directive-try-each (Section 8.7.7.3) or suit-directive-run-sequence (Section 8.7.7.12) and a condition failure occurs, the manifest processor aborts the sequence. For suit-directive-try-each, if Soft Failure is True, the next sequence in Try Each is invoked, otherwise suit-directive-try-each fails with the condition failure code. In suit-directive-run-sequence, if Soft Failure is True the suit-directive-run-sequence simply halts with no side-effects and the Manifest Processor continues with the following command, otherwise, the suit-directive-run-sequence fails with the condition failure code.

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

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

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

8.7.5.24. suit-parameter-custom

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

8.7.6. 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. All Conditions specify a Reporting Policy as described Section 8.7.4. Conditions include:

Name	CDDL Structure	Reference
Vendor Identifier	suit-condition-vendor-identifier	Section 8.7.6.1
Class Identifier	suit-condition-class-identifier	Section 8.7.6.1
Device Identifier	suit-condition-device-identifier	Section 8.7.6.1
Image Match	suit-condition-image-match	Section 8.7.6.2
Image Not Match	suit-condition-image-not-match	Section 8.7.6.3
Use Before	suit-condition-use-before	Section 8.7.6.4
Component Offset	suit-condition-component-offset	Section 8.7.6.5
Minimum Battery	suit-condition-minimum-battery	Section 8.7.6.6
Update Authorized	suit-condition-update-authorized	Section 8.7.6.7
Version	suit-condition-version	Section 8.7.6.8
Abort	suit-condition-abort	Section 8.7.6.9
Custom Condition	suit-condition-custom	Section 8.7.6.10

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

Conditions compare parameters against properties of the system. These properties may be asserted in many different ways, including: calculation on-demand, volatile definition in memory, static definition within the manifest processor, storage in known location within an image, storage within a key storage system, storage in One-

Time-Programmable memory, inclusion in mask ROM, or inclusion as a register in hardware. Some of these assertion methods are global in scope, such as a hardware register, some are scoped to an individual component, such as storage at a known location in an image, and some assertion methods can be either global or component-scope, based on implementation.

Each condition MUST report a result code on completion. If a condition reports failure, then the current sequence of commands MUST terminate. A subsequent command or command sequence MAY continue executing if suit-parameter-soft-failure (Section 8.7.5.23) is set. If a condition requires additional information, this MUST be specified in one or more parameters before the condition is executed. If a Recipient attempts to process a condition that expects additional information and that information has not been set, it MUST report a failure. If a Recipient encounters an unknown condition, it MUST report a failure.

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

8.7.6.1. suit-condition-vendor-identifier, suit-condition-class-identifier, and suit-condition-device-identifier

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

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

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

8.7.6.2. suit-condition-image-match

Verify that the current component matches the suit-parameter-image-digest (Section 8.7.5.6) for the current component. The digest is verified against the digest specified in the Component's parameters list. If no digest is specified, the condition fails. suit-condition-image-match is REQUIRED to implement.

8.7.6.3. suit-condition-image-not-match

Verify that the current component does not match the suit-parameter-image-digest (Section 8.7.5.6). If no digest is specified, the condition fails. suit-condition-image-not-match is OPTIONAL to implement.

8.7.6.4. 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. The recipient evaluates the current time against the suit-parameter-use-before parameter (Section 8.7.5.8), which must have already been set as a parameter, encoded as seconds after 1970-01-01 00:00:00 UTC. Timestamp conditions MUST be evaluated in 64 bits, regardless of encoded CBOR size. suit-condition-use-before is OPTIONAL to implement.

8.7.6.5. suit-condition-component-offset

Verify that the offset of the current component matches the offset set in suit-parameter-component-offset (Section 8.7.5.9). This condition allows a manifest to select between several images to match a target offset.

8.7.6.6. suit-condition-minimum-battery

suit-condition-minimum-battery provides a mechanism to test a Recipient's battery level before installing an update. This condition is primarily 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. suit-condition-minimum-battery consumes suit-parameter-minimum-battery (Section 8.7.5.16).

8.7.6.7. suit-condition-update-authorized

Request Authorization from the application and fail if not authorized. This can allow a user to decline an update. suit-parameter-update-priority (Section 8.7.5.17) provides an integer priority level that the application can use to determine whether or not to authorize the update. Priorities are application defined. suit-condition-update-authorized is OPTIONAL to implement.

8.7.6.8. suit-condition-version

suit-condition-version allows comparing versions of firmware. Verifying image digests is preferred to version checks because digests are more precise. suit-condition-version examines a component's version against the version info specified in suit-parameter-version (Section 8.7.5.18)

8.7.6.9. suit-condition-abort

Unconditionally fail. This operation is typically used in conjunction with suit-directive-try-each (Section 8.7.7.3).

8.7.6.10. suit-condition-custom

suit-condition-custom describes any proprietary, application specific condition. This is encoded as a negative integer, chosen by the firmware developer. If additional information must be provided to the condition, it should be encoded in a custom parameter (a nint) as described in Section 8.7.5. SUIT_Condition_Custom is OPTIONAL to implement.

8.7.7. SUIT_Directive

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

Name	CDDL Structure	Reference
Set Component Index	suit-directive-set-component-index	Section 8.7.1
Set Dependency Index	suit-directive-set-dependency-index	Section 8.7.2
Try Each	suit-directive-try-each	Section 8.7.3
Process Dependency	suit-directive-process-dependency	Section 8.7.4
Set Parameters	suit-directive-set-parameters	Section 8.7.5
Override Parameters	suit-directive-override-parameters	Section 8.7.6
Fetch	suit-directive-fetch	Section 8.7.7
Fetch URI list	suit-directive-fetch-uri-list	Section 8.7.8
Copy	suit-directive-copy	Section 8.7.9
Run	suit-directive-run	Section 8.7.10
Wait For Event	suit-directive-wait	Section 8.7.11
Run Sequence	suit-directive-run-sequence	Section 8.7.12
Swap	suit-directive-swap	Section 8.7.13

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

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

8.7.7.1. suit-directive-set-component-index

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

1. An unsigned integer (REQUIRED to implement in parser)

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

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

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

If the following commands 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.

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

8.7.7.2. 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, or an array of unsigned integer indices into the list of 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.

If dependency index is set to True when a command is invoked, then the command applies to all dependencies, in the order they appear in suit-common-components. When the Manifest Processor invokes a command while the dependency index is set to True, the Manifest Processor MUST execute the command once for each possible dependency index, ensuring that the command receives the parameters corresponding to that dependency index. If the dependency index is set to an array of unsigned integers, then the Manifest Processor MUST execute the command once for each listed dependency index, ensuring that the command receives the parameters corresponding to that dependency index.

See Section 6.5 for more details.

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

8.7.7.3. suit-directive-try-each

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

suit-parameter-soft-failure (Section 8.7.5.23) is initialized to True at the beginning of each sequence. If one sequence aborts due to a condition failure, the next is started. If no sequence completes without condition failure, then suit-directive-try-each returns an error. If a particular application calls for all sequences to fail and still continue, then an empty sequence (nil) can be added to the Try Each Argument.

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

8.7.7.4. 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," then the command sequence MUST be terminated with an error.

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

8.7.7.5. suit-directive-set-parameters

suit-directive-set-parameters allows the manifest to configure behavior 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 behavior of its dependencies.

Available parameters are defined in Section 8.7.5.

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 behavior of a manifest.

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

8.7.7.6. suit-directive-override-parameters

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

Available parameters are defined in Section 8.7.5.

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

8.7.7.7. 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 dependency-index, respectively. If component-index or dependency-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 behavior. If an argument is needed, it must be wrapped in a bstr and set in suit-parameter-fetch-arguments.

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

The behavior 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.

8.7.7.8. suit-directive-fetch-uri-list

suit-directive-fetch-uri-list uses the same semantics as suit-directive-fetch (Section 8.7.7.7), except that it iterates over the URI List (Section 8.7.5.20) to select a URI to fetch from.

8.7.7.9. suit-directive-copy

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

suit-directive-copy reads its source from suit-parameter-source-component (Section 8.7.5.14).

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

8.7.7.10. suit-directive-run

suit-directive-run directs the manifest processor to transfer execution to the current Component Index. When this is invoked, the manifest processor MAY be unloaded and execution continues in the Component Index. Arguments are provided to suit-directive-run through suit-parameter-run-arguments (Section 8.7.5.15) and are forwarded to the executable code located in Component Index in an application-specific way. For example, this could form the Linux Kernel Command Line if booting a Linux device.

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

8.7.7.11. suit-directive-wait

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

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

8.7.7.12. suit-directive-run-sequence

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

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

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

suit-parameter-soft-failure (Section 8.7.5.23) defaults to False when suit-directive-run-sequence begins. Its value is discarded when suit-directive-run-sequence terminates.

8.7.7.13. suit-directive-swap

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

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

8.7.8. Integrity Check Values

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

Each Integrity Check Value covers the corresponding Envelope Element as described in Section 8.8.

8.8. Severable Elements

Because the manifest can be used by different actors at different times, some parts of the manifest can be removed or "Severed" without affecting later stages of the lifecycle. Severing of information is achieved by separating that information from the signed container so

that removing it does not affect the signature. This means that ensuring integrity of severable parts of the manifest is a requirement for the signed portion of the manifest. Severing some parts makes it possible to discard parts of the manifest that are no longer necessary. This is important because it allows the storage used by the manifest to be greatly reduced. For example, no text size limits are needed if text is removed from the manifest prior to delivery to a constrained device.

Elements are made severable by removing them from the manifest, encoding them in a bstr, and placing a 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 (Digest Bits)/8 + 4 SHOULD NOT be severable. Elements larger than (Digest Bits)/8 + 4 MAY be severable, while elements that are much larger than (Digest Bits)/8 + 4 SHOULD be severable.

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

9. Access Control Lists

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

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

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

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

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

10. 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.

The SUIT digest is a CBOR List containing two elements: a suit-digest-algorithm-id and a bstr containing the bytes of the digest.

11. IANA Considerations

IANA is requested to:

- allocate CBOR tag 48 in the CBOR Tags registry for the SUIT Envelope.
- allocate CBOR tag 480 in the CBOR Tags registry for the SUIT Manifest.
- allocate media type application/suit-envelope in the Media Types registry.
- setup several registries as described below.

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

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

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

11.1. SUIT Commands

Label	Name	Reference	
1	Vendor Identifier	Section 8.7.6.1	
2	Class Identifier	Section 8.7.6.1	
3	Image	Section 8.7.6.2	

	Match		
4	Use Before	Section 8.7.6.4	
5	Component Offset	Section 8.7.6.5	
12	Set Component Index	Section 8.7.7.1	
13	Set Dependency Index	Section 8.7.7.2	
14	Abort		
15	Try Each	Section 8.7.7.3	
16	Reserved		
17	Reserved		
18	Process Dependency	suit-directive-process-dependency	Section 8.7.7.4
19	Set Parameters	Section 8.7.7.5	
20	Override Parameters	Section 8.7.7.6	
21	Fetch	Section 8.7.7.7	
22	Copy	Section 8.7.7.9	
23	Run	Section 8.7.7.10	
24	Device Identifier	Section 8.7.6.1	
25	Image Not Match	Section 8.7.6.3	
26	Minimum Battery	Section 8.7.6.6	
27	Update	Section 8.7.6.7	

	Authorized		
28	Version	Section 8.7.6.8	
29	Wait For Event	Section 8.7.7.11	
30	Fetch URI List	Section 8.7.7.8	
31	Swap	Section 8.7.7.13	
32	Run Sequence	Section 8.7.7.12	
nint	Custom Condition	Section 8.7.6.10	

11.2. SUIT Parameters

Label	Name	Reference
1	Vendor ID	Section 8.7.5.3
2	Class ID	Section 8.7.5.4
3	Image Digest	Section 8.7.5.6
4	Use Before	Section 8.7.5.8
5	Component Offset	Section 8.7.5.9
12	Strict Order	Section 8.7.5.22
13	Soft Failure	Section 8.7.5.23
14	Image Size	Section 8.7.5.7
18	Encryption Info	Section 8.7.5.10
19	Compression Info	Section 8.7.5.11
20	Unpack Info	Section 8.7.5.12
21	URI	Section 8.7.5.13
22	Source Component	Section 8.7.5.14
23	Run Args	Section 8.7.5.15
24	Device ID	Section 8.7.5.5
26	Minimum Battery	Section 8.7.5.16
27	Update Priority	Section 8.7.5.17
28	Version	{{suit-parameter-version}}
29	Wait Info	Section 8.7.5.19
30	URI List	Section 8.7.5.20
nint	Custom	Section 8.7.5.24

11.3. SUIT Text Values

Label	Name	Reference
1	Manifest Description	Section 8.6.4
2	Update Description	Section 8.6.4
3	Manifest JSON Source	Section 8.6.4
4	Manifest YAML Source	Section 8.6.4
nint	Custom	Section 8.6.4

11.4. SUIT Component Text Values

Label	Name	Reference
1	Vendor Name	Section 8.6.4
2	Model Name	Section 8.6.4
3	Vendor Domain	Section 8.6.4
4	Model Info	Section 8.6.4
5	Component Description	Section 8.6.4
6	Component Version	Section 8.6.4
7	Component Version Required	Section 8.6.4
nint	Custom	Section 8.6.4

11.5. SUIT Algorithm Identifiers

11.5.1. SUIT Digest Algorithm Identifiers

Label	Name	
1	SHA224	Section 10
2	SHA256	Section 10
3	SHA384	Section 10
4	SHA512	Section 10
5	SHA3-224	Section 10
6	SHA3-256	Section 10
7	SHA3-384	Section 10
8	SHA3-512	Section 10

11.5.2. SUIT Compression Algorithm Identifiers

Label	Name	Reference
1	zlib	Section 8.7.5.11
2	Brotli	Section 8.7.5.11
3	zstd	Section 8.7.5.11

11.5.3. Unpack Algorithms

Label	Name	Reference
1	HEX	Section 8.7.5.12
2	ELF	Section 8.7.5.12
3	COFF	Section 8.7.5.12
4	SREC	Section 8.7.5.12

12. Security Considerations

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

13. Acknowledgements

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

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

14. References

14.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <https://www.rfc-editor.org/info/rfc4122>.

[RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <https://www.rfc-editor.org/info/rfc8152>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

14.2. Informative References

[COFF] Wikipedia, ., "Common Object File Format (COFF)", 2020, <https://en.wikipedia.org/wiki/COFF>.

[ELF] Wikipedia, ., "Executable and Linkable Format (ELF)", 2020, <https://en.wikipedia.org/wiki/Executable_and_Linkable_Format>.

[HEX] Wikipedia, ., "Intel HEX", 2020, <https://en.wikipedia.org/wiki/Intel_HEX>.

[I-D.ietf-cbor-tags-oid] Bormann, C. and S. Leonard, "Concise Binary Object Representation (CBOR) Tags for Object Identifiers", draft-ietf-cbor-tags-oid-03 (work in progress), November 2020.

[I-D.ietf-sacm-coswid] Birkholz, H., Fitzgerald-McKay, J., Schmidt, C., and D. Waltermire, "Concise Software Identification Tags", draft-ietf-sacm-coswid-16 (work in progress), November 2020.

[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-14 (work in progress), October 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-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.

[I-D.kucherawy-rfc8478bis]
Collet, Y. and M. Kucherawy, "Zstandard Compression and the application/zstd Media Type", draft-kucherawy-rfc8478bis-05 (work in progress), April 2020.

[RFC1950] Deutsch, P. and J-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3", RFC 1950, DOI 10.17487/RFC1950, May 1996, <https://www.rfc-editor.org/info/rfc1950>.

[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>.

[RFC7932] Alakuijala, J. and Z. Szabadka, "Brotli Compressed Data Format", RFC 7932, DOI 10.17487/RFC7932, July 2016, <https://www.rfc-editor.org/info/rfc7932>.

[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>.

[SREC] Wikipedia, ., "SREC (file format)", 2020, <https://en.wikipedia.org/wiki/SREC_(file_format)>.

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

Appendix A. A. Full CDDL

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

```
SUIT_Envelope_Tagged = #6.48(SUIT_Envelope)
SUIT_Envelope = {
  ? suit-delegation => bstr .cbor SUIT_Delegation,
  suit-authentication-wrapper => bstr .cbor SUIT_Authentication,
  suit-manifest  => bstr .cbor SUIT_Manifest,
  SUIT_Severable_Manifest_Members,
  * SUIT_Integrated_Payload,
  * SUIT_Integrated_Dependency,
  * $$SUIT_Envelope_Extensions,
  * (int => bstr)
}

SUIT_Delegation = [ + [ + bstr .cbor CWT ] ]

CWT = SUIT_Authentication_Block

SUIT_Authentication = [
    bstr .cbor SUIT_Digest,
    * bstr .cbor SUIT_Authentication_Block
]

SUIT_Digest = [
  suit-digest-algorithm-id : suit-digest-algorithm-ids,
  suit-digest-bytes : bstr,
  * $$SUIT_Digest-extensions
]

; Named Information Hash Algorithm Identifiers
suit-digest-algorithm-ids /= algorithm-id-sha224
suit-digest-algorithm-ids /= algorithm-id-sha256
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_Authentication_Block /= COSE_Mac_Tagged
SUIT_Authentication_Block /= COSE_Sign_Tagged
SUIT_Authentication_Block /= COSE_Mac0_Tagged
SUIT_Authentication_Block /= 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

SUIT_Severable_Manifest_Members = (
  ? 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 => bstr .cbor SUIT_Text_Map,
  ? suit-coswid => bstr .cbor concise-software-identity,
  * $$SUIT_severable-members-extensions,
)

SUIT_Integrated_Payload = (suit-integrated-payload-key => bstr)
SUIT_Integrated_Dependency = (
    suit-integrated-payload-key => bstr .cbor SUIT_Envelope
)
suit-integrated-payload-key = nint / uint .ge 24

SUIT_Manifest_Tagged = #6.480(SUIT_Manifest)

SUIT_Manifest = {
    suit-manifest-version         => 1,
    suit-manifest-sequence-number => uint,
    suit-common                   => bstr .cbor SUIT_Common,
    ? suit-reference-uri          => tstr,
    SUIT_Severable_Manifest_Members,
    SUIT_Severable_Members_Digests,
    SUIT_Unseverable_Members,
    * $$SUIT_Manifest_Extensions,
}

SUIT_Unseverable_Members = (
  ? suit-validate => bstr .cbor SUIT_Command_Sequence,
  ? suit-load => bstr .cbor SUIT_Command_Sequence,
  ? suit-run => bstr .cbor SUIT_Command_Sequence,
  * $$unserverble-manifest-member-extensions,
)

SUIT_Severable_Members_Digests = (
  ? suit-dependency-resolution => SUIT_Digest,
  ? suit-payload-fetch => SUIT_Digest,
  ? suit-install => SUIT_Digest,
  ? suit-text => SUIT_Digest,
  ? suit-coswid => SUIT_Digest,
```

```
  * $$severable-manifest-members-digests-extensions
)

SUIT_Common = {
    ? suit-dependencies           => SUIT_Dependencies,
    ? suit-components             => SUIT_Components,
    ? suit-common-sequence        => bstr .cbor SUIT_Common_Sequence,
    * $$SUIT_Common-extensions,
}

SUIT_Dependencies         = [ + SUIT_Dependency ]
SUIT_Components           = [ + SUIT_Component_Identifier ]

concise-software-identity = any

SUIT_Dependency = {
    suit-dependency-digest => SUIT_Digest,
    ? suit-dependency-prefix => SUIT_Component_Identifier,
    * $$SUIT_Dependency-extensions,
}

SUIT_Component_Identifier =  [* bstr]

SUIT_Common_Sequence = [
    + ( SUIT_Condition // SUIT_Common_Commands )
]

SUIT_Common_Commands //= (suit-directive-set-component-index,  IndexArg)
SUIT_Common_Commands //= (suit-directive-set-dependency-index, IndexArg)
SUIT_Common_Commands //= (suit-directive-run-sequence,
    bstr .cbor SUIT_Command_Sequence)
SUIT_Common_Commands //= (suit-directive-try-each,
    SUIT_Directive_Try_Each_Argument)
SUIT_Common_Commands //= (suit-directive-set-parameters,
    {+ SUIT_Parameters})
SUIT_Common_Commands //= (suit-directive-override-parameters,
    {+ SUIT_Parameters})

IndexArg /= uint
IndexArg /= bool
IndexArg /= [+uint]

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

SUIT_Command_Custom = (suit-command-custom, bstr/tstr/int/nil)
SUIT_Condition //= (suit-condition-vendor-identifier, SUIT_Rep_Policy)
```

```
SUIT_Condition //= (suit-condition-class-identifier,  SUIT_Rep_Policy)
SUIT_Condition //= (suit-condition-device-identifier, SUIT_Rep_Policy)
SUIT_Condition //= (suit-condition-image-match,       SUIT_Rep_Policy)
SUIT_Condition //= (suit-condition-image-not-match,   SUIT_Rep_Policy)
SUIT_Condition //= (suit-condition-use-before,        SUIT_Rep_Policy)
SUIT_Condition //= (suit-condition-minimum-battery,   SUIT_Rep_Policy)
SUIT_Condition //= (suit-condition-update-authorized, SUIT_Rep_Policy)
SUIT_Condition //= (suit-condition-version,           SUIT_Rep_Policy)
SUIT_Condition //= (suit-condition-component-offset,  SUIT_Rep_Policy)
SUIT_Condition //= (suit-condition-abort,             SUIT_Rep_Policy)

SUIT_Directive //= (suit-directive-set-component-index,  IndexArg)
SUIT_Directive //= (suit-directive-set-dependency-index, IndexArg)
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, SUIT_Rep_Policy)
SUIT_Directive //= (suit-directive-set-parameters,
    {+ SUIT_Parameters})
SUIT_Directive //= (suit-directive-override-parameters,
    {+ SUIT_Parameters})
SUIT_Directive //= (suit-directive-fetch,             SUIT_Rep_Policy)
SUIT_Directive //= (suit-directive-copy,              SUIT_Rep_Policy)
SUIT_Directive //= (suit-directive-swap,              SUIT_Rep_Policy)
SUIT_Directive //= (suit-directive-run,               SUIT_Rep_Policy)
SUIT_Directive //= (suit-directive-wait,              SUIT_Rep_Policy)
SUIT_Directive //= (suit-directive-fetch-uri-list,    SUIT_Rep_Policy)

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

SUIT_Rep_Policy = uint .bits suit-reporting-bits

suit-reporting-bits = &(
    suit-send-record-success : 0,
    suit-send-record-failure : 1,
    suit-send-sysinfo-success : 2,
    suit-send-sysinfo-failure : 3
)

SUIT_Wait_Event = { + SUIT_Wait_Events }

SUIT_Wait_Events //= (suit-wait-event-authorization => 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_Other_Device_Version = [
    other-device: bstr,
    other-device-version: [ + SUIT_Parameter_Version_Match ]
]

SUIT_Parameters //= (suit-parameter-vendor-identifier =>
    (RFC4122_UUID / cbor-pen))
cbor-pen = #6.112(bstr)

SUIT_Parameters //= (suit-parameter-class-identifier => RFC4122_UUID)
SUIT_Parameters //= (suit-parameter-image-digest
    => bstr .cbor SUIT_Digest)
SUIT_Parameters //= (suit-parameter-image-size => uint)
SUIT_Parameters //= (suit-parameter-use-before => uint)
SUIT_Parameters //= (suit-parameter-component-offset => uint)

SUIT_Parameters //= (suit-parameter-encryption-info
    => bstr .cbor SUIT_Encryption_Info)
SUIT_Parameters //= (suit-parameter-compression-info
    => bstr .cbor SUIT_Compression_Info)
SUIT_Parameters //= (suit-parameter-unpack-info
    => bstr .cbor SUIT_Unpack_Info)

SUIT_Parameters //= (suit-parameter-uri => tstr)
SUIT_Parameters //= (suit-parameter-source-component => uint)
SUIT_Parameters //= (suit-parameter-run-args => bstr)

SUIT_Parameters //= (suit-parameter-device-identifier => RFC4122_UUID)
SUIT_Parameters //= (suit-parameter-minimum-battery => uint)
SUIT_Parameters //= (suit-parameter-update-priority => uint)
SUIT_Parameters //= (suit-parameter-version =>
    SUIT_Parameter_Version_Match)
SUIT_Parameters //= (suit-parameter-wait-info =>
    bstr .cbor SUIT_Wait_Event)

SUIT_Parameters //= (suit-parameter-custom => int/bool/tstr/bstr)

SUIT_Parameters //= (suit-parameter-strict-order => bool)
SUIT_Parameters //= (suit-parameter-soft-failure => bool)
```

```
SUIT_Parameters //= (suit-parameter-uri-list =>
    bstr .cbor SUIT_URI_List)

RFC4122_UUID = bstr .size 16

SUIT_Parameter_Version_Match = [
    suit-condition-version-comparison-type:
        SUIT_Condition_Version_Comparison_Types,
    suit-condition-version-comparison-value:
        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]

SUIT_Encryption_Info = COSE_Encrypt_Tagged/COSE_Encrypt0_Tagged
SUIT_Compression_Info = {
    suit-compression-algorithm => SUIT_Compression_Algorithms,
    * $$SUIT_Compression_Info-extensions,
}

SUIT_Compression_Algorithms /= SUIT_Compression_Algorithm_zlib
SUIT_Compression_Algorithms /= SUIT_Compression_Algorithm_brotli
SUIT_Compression_Algorithms /= SUIT_Compression_Algorithm_zstd

SUIT_Compression_Algorithm_zlib = 1
SUIT_Compression_Algorithm_brotli = 2
SUIT_Compression_Algorithm_zstd = 3

SUIT_Unpack_Info = {
    suit-unpack-algorithm => SUIT_Unpack_Algorithms,
    * $$SUIT_Unpack_Info-extensions,
```

```
}

SUIT_Unpack_Algorithms /= SUIT_Unpack_Algorithm_Hex
SUIT_Unpack_Algorithms /= SUIT_Unpack_Algorithm_Elf
SUIT_Unpack_Algorithms /= SUIT_Unpack_Algorithm_Coff
SUIT_Unpack_Algorithms /= SUIT_Unpack_Algorithm_Srec

SUIT_Unpack_Algorithm_Hex = 1
SUIT_Unpack_Algorithm_Elf = 2
SUIT_Unpack_Algorithm_Coff = 3
SUIT_Unpack_Algorithm_Srec = 4

SUIT_URI_List = [+ tstr ]

SUIT_Text_Map = {
    * SUIT_Component_Identifier => {
        SUIT_Text_Component_Keys
    },
    SUIT_Text_Keys
}

SUIT_Text_Component_Keys = (
    ? suit-text-vendor-name           => tstr,
    ? suit-text-model-name            => tstr,
    ? suit-text-vendor-domain         => tstr,
    ? suit-text-model-info            => tstr,
    ? suit-text-component-description => tstr,
    ? suit-text-component-version     => tstr,
    ? suit-text-version-required      => tstr,
    * $$suit-text-component-key-extensions
)

SUIT_Text_Keys = (
    ? suit-text-manifest-description => tstr,
    ? suit-text-update-description   => tstr,
    ? suit-text-manifest-json-source => tstr,
    ? suit-text-manifest-yaml-source => tstr,
    * $$suit-text-key-extensions
)

suit-delegation = 1
suit-authentication-wrapper = 2
suit-manifest = 3

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

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

suit-dependencies = 1
suit-components = 2
suit-common-sequence = 4

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

suit-command-custom = nint

suit-condition-vendor-identifier = 1
suit-condition-class-identifier  = 2
suit-condition-image-match       = 3
suit-condition-use-before        = 4
suit-condition-component-offset  = 5

suit-condition-abort                    = 14
suit-condition-device-identifier        = 24
suit-condition-image-not-match          = 25
suit-condition-minimum-battery          = 26
suit-condition-update-authorized        = 27
suit-condition-version                  = 28

suit-directive-set-component-index      = 12
suit-directive-set-dependency-index     = 13
suit-directive-try-each                 = 15
;suit-directive-do-each                  = 16 ; TBD
;suit-directive-map-filter               = 17 ; TBD
suit-directive-process-dependency       = 18
suit-directive-set-parameters           = 19
suit-directive-override-parameters      = 20
```

```
suit-directive-fetch                  = 21
suit-directive-copy                   = 22
suit-directive-run                    = 23

suit-directive-wait                   = 29
suit-directive-fetch-uri-list         = 30
suit-directive-swap                   = 31
suit-directive-run-sequence           = 32

suit-wait-event-authorization = 1
suit-wait-event-power = 2
suit-wait-event-network = 3
suit-wait-event-other-device-version = 4
suit-wait-event-time = 5
suit-wait-event-time-of-day = 6
suit-wait-event-day-of-week = 7

suit-parameter-vendor-identifier = 1
suit-parameter-class-identifier  = 2
suit-parameter-image-digest      = 3
suit-parameter-use-before        = 4
suit-parameter-component-offset  = 5

suit-parameter-strict-order      = 12
suit-parameter-soft-failure      = 13
suit-parameter-image-size        = 14

suit-parameter-encryption-info   = 18
suit-parameter-compression-info  = 19
suit-parameter-unpack-info       = 20
suit-parameter-uri               = 21
suit-parameter-source-component  = 22
suit-parameter-run-args          = 23

suit-parameter-device-identifier = 24
suit-parameter-minimum-battery   = 26
suit-parameter-update-priority   = 27
suit-parameter-version           = 28
suit-parameter-wait-info         = 29
suit-parameter-uri-list          = 30

suit-parameter-custom = nint

suit-compression-algorithm = 1

suit-unpack-algorithm  = 1

suit-text-manifest-description  = 1
```

```
suit-text-update-description    = 2
suit-text-manifest-json-source  = 3
suit-text-manifest-yaml-source  = 4

suit-text-vendor-name           = 1
suit-text-model-name            = 2
suit-text-vendor-domain         = 3
suit-text-model-info            = 4
suit-text-component-description = 5
suit-text-component-version     = 6
suit-text-version-required      = 7
```

Appendix B. B. Examples

The following examples demonstrate a small subset of the functionality of the manifest. Even a simple manifest processor can execute most of these manifests.

The examples are signed using the following ECDSA secp256r1 key:

```
-----BEGIN PRIVATE KEY-----
MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgApZYjZCUGLM50VBC
CjYStX+09jGmnyJPrpDLTz/hiXOhRANCAASEloEarguqq9JhVxie7NomvqqL8Rtv
P+bitWWchdvArTsfKktsCYExwKNtrNHXi9OB3N+wnAUtszmR23M4tKiW
-----END PRIVATE KEY-----
```

The corresponding public key can be used to verify these examples:

```
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEhJaBGq4LqqvSYVcYnuzaJr6qi/Eb
bz/m4rVlnIXbwK07HypLbAmBMcCjbazR14vTgdzfsJwFLbM5kdtzOLSolg==
-----END PUBLIC KEY-----
```

Each example uses SHA256 as the digest function.

Note that reporting policies are declared for each non-flow-control command in these examples. The reporting policies used in the examples are described in the following tables.

Policy	Label
suit-send-record-on-success	Rec-Pass
suit-send-record-on-failure	Rec-Fail
suit-send-sysinfo-success	Sys-Pass
suit-send-sysinfo-failure	Sys-Fail

Command	Sys-Fail	Sys-Pass	Rec-Fail	Rec-Pass
suit-condition-vendor-identifier	1	1	1	1
suit-condition-class-identifier	1	1	1	1
suit-condition-image-match	1	1	1	1
suit-condition-component-offset	0	1	0	1
suit-directive-fetch	0	0	1	0
suit-directive-copy	0	0	1	0
suit-directive-run	0	0	1	0

B.1. Example 0: Secure Boot

This example covers the following templates:

- Compatibility Check (Section 7.1)

- Secure Boot (Section 7.2)

It also serves as the minimum example.

```
{
    / authentication-wrapper / 2:bstr .cbor ({        digest: bstr
.cbor ([
            / algorithm-id / 2 / "sha256" /,
```

```
            / digest-bytes /
  h'5c097ef64bf3bb9b494e71e1f2418eef8d466cc902f639a855ec9af3e9eddb99'
        ])        signatures: [
            bstr .cbor (18([
                    / protected / bstr .cbor ({
                        / alg / 1:-7 / "ES256" /,
                    }),
                    / unprotected / {
                    },
                    / payload / bstr .cbor ([
                        / algorithm-id / 2 / "sha256" /,
                        / digest-bytes /
  h'5c097ef64bf3bb9b494e71e1f2418eef8d466cc902f639a855ec9af3e9eddb99'
                    ]),
                    / signature / h'60f5c3d03a3aa759bfef2ef0f5f97a93b1
  f5e741f7463f4385af88513a5c2957bea2d6c4cfddd03392a267aab0fc0fd515560ed5
  8e33fad26ac32a024c5a7143'
                ]))
        ]
    }),
    / manifest / 3:bstr .cbor ({
        / manifest-version / 1:1,
        / manifest-sequence-number / 2:0,
        / common / 3:bstr .cbor ({
            / components / 2:[
                [h'00']
            ],
            / common-sequence / 4:bstr .cbor ([
                / directive-override-parameters / 20,{
                    / vendor-id /
  1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
  be9d-e663e4d41ffe /,
                    / class-id / 2:h'1492af1425695e48bf429b2d51f2ab45'
  / 1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
                    / image-digest / 3:bstr .cbor ([
                        / algorithm-id / 2 / "sha256" /,
                        / digest-bytes /
  h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
                    ]),
                    / image-size / 14:34768,
                } ,
                / condition-vendor-identifier / 1,15 ,
                / condition-class-identifier / 2,15
            ]),
        }),
        / validate / 10:bstr .cbor ([
            / condition-image-match / 3,15
        ]),
```

```
          / run / 12:bstr .cbor ([
              / directive-run / 23,2
          ]),
      }),
  }
```

Total size of Envelope without COSE authentication object: 159

Envelope:

```
a2025827815824820258205c097ef64bf3bb9b494e71e1f2418eef8d466c
c902f639a855ec9af3e9eddb99035871a50101020003585fa20281814100
0458568614a40150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492af14
25695e48bf429b2d51f2ab450358248202582000112233445566778899aa
bbccddeeff0123456789abcdeffedcba98765432100e1987d0010f020f0a
4382030f0c43821702
```

Total size of Envelope with COSE authentication object: 272

Envelope with COSE authentication object:

```
a2025898825824820258205c097ef64bf3bb9b494e71e1f2418eef8d466c
c902f639a855ec9af3e9eddb99586fd28443a10126a05824820258205c09
7ef64bf3bb9b494e71e1f2418eef8d466cc902f639a855ec9af3e9eddb99
584060f5c3d03a3aa759bfef2ef0f5f97a93b1f5e741f7463f4385af8851
3a5c2957bea2d6c4cfddd03392a267aab0fc0fd515560ed58e33fad26ac3
2a024c5a7143035871a50101020003585fa202818141000458568614a401
50fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b
2d51f2ab450358248202582000112233445566778899aabbccddeeff0123
456789abcdeffedcba98765432100e1987d0010f020f0a4382030f0c4382
1702
```

B.2. Example 1: Simultaneous Download and Installation of Payload

This example covers the following templates:

- Compatibility Check (Section 7.1)

- Firmware Download (Section 7.3)

Simultaneous download and installation of payload. No secure boot is present in this example to demonstrate a download-only manifest.

```
{
    / authentication-wrapper / 2:bstr .cbor ({        digest: bstr
.cbor ([
            / algorithm-id / 2 / "sha256" /,
            / digest-bytes /
```

```
  h'987eec85fa99fd31d332381b9810f90b05c2e0d4f284a6f4211207ed00fff750'
          ])        signatures: [
              bstr .cbor (18([
                      / protected / bstr .cbor ({
                          / alg / 1:-7 / "ES256" /,
                      }),
                      / unprotected / {
                      },
                      / payload / bstr .cbor ([
                          / algorithm-id / 2 / "sha256" /,
                          / digest-bytes /
  h'987eec85fa99fd31d332381b9810f90b05c2e0d4f284a6f4211207ed00fff750'
                      ]),
                      / signature / h'750141d65b4f20a88dc70c6785a67e0f4f
  085aead83ba2289d6e37271508cc91e0a0592f5c940c2257c9c0b26403c0ba4477f2ce
  37b60089fe02cde7911d1c15'
                  ]))
          ]
      }),
      / manifest / 3:bstr .cbor ({
          / manifest-version / 1:1,
          / manifest-sequence-number / 2:1,
          / common / 3:bstr .cbor ({
              / components / 2:[
                  [h'00']
              ],
              / common-sequence / 4:bstr .cbor ([
                  / directive-override-parameters / 20,{
                      / vendor-id /
  1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
  be9d-e663e4d41ffe /,
                      / class-id / 2:h'1492af1425695e48bf429b2d51f2ab45'
  / 1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
                      / image-digest / 3:bstr .cbor ([
                          / algorithm-id / 2 / "sha256" /,
                          / digest-bytes /
  h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
                      ]),
                      / image-size / 14:34768,
                  } ,
                  / condition-vendor-identifier / 1,15 ,
                  / condition-class-identifier / 2,15
              ]),
          }),
          / install / 9:bstr .cbor ([
              / directive-set-parameters / 19,{
                  / uri / 21:'http://example.com/file.bin',
              } ,
```

```
                / directive-fetch / 21,2 ,
                / condition-image-match / 3,15
            ]),
            / validate / 10:bstr .cbor ([
                / condition-image-match / 3,15
            ]),
        }),
    }
```

Total size of Envelope without COSE authentication object: 194

Envelope:

```
a2025827815824820258209 87eec85fa99fd31d332381b9810f90b05c2e0
d4f284a6f4211207ed00fff750035894a50101020103585fa20281814100
0458568614a40150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492af14
25695e48bf429b2d51f2ab450358248202582000112233445566778899aa
bbccddeeff0123456789abcdeffedcba98765432100e1987d0010f020f09
58258613a115781b687474703a2f2f6578616d706c652e636f6d2f66696c
652e62696e1502030f0a4382030f
```

Total size of Envelope with COSE authentication object: 307

Envelope with COSE authentication object:

```
a2025898258248202582098 7eec85fa99fd31d332381b9810f90b05c2e0
d4f284a6f4211207ed00fff750586fd28443a10126a0582482025820987e
ec85fa99fd31d332381b9810f90b05c2e0d4f284a6f4211207ed00fff750
5840750141d65b4f20a88dc70c6785a67e0f4f085aead83ba2289d6e3727
1508cc91e0a0592f5c940c2257c9c0b26403c0ba4477f2ce37b60089fe02
cde7911d1c15035894a50101020103585fa202818141000458568614a401
50fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b
2d51f2ab450358248202582000112233445566778899aabbccddeeff0123
456789abcdeffedcba98765432100e1987d0010f020f0958258613a11578
1b687474703a2f2f6578616d706c652e636f6d2f66696c652e62696e1502
030f0a4382030f
```

B.3. Example 2: Simultaneous Download, Installation, Secure Boot, Severed Fields

This example covers the following templates:

- Compatibility Check (Section 7.1)
- Secure Boot (Section 7.2)
- Firmware Download (Section 7.3)

This example also demonstrates severable elements (Section 5.5), and text (Section 8.6.4).

```
{
    / authentication-wrapper / 2:bstr .cbor ({        digest: bstr
.cbor ([
            / algorithm-id / 2 / "sha256" /,
            / digest-bytes /
h'75685579a83babd71ec8ef22fa49ac873f78a708a43a674e782ad30b6598d17a'
        ])        signatures: [
            bstr .cbor (18([
                    / protected / bstr .cbor ({
                        / alg / 1:-7 / "ES256" /,
                    }),
                    / unprotected / {
                    },
                    / payload / bstr .cbor ([
                        / algorithm-id / 2 / "sha256" /,
                        / digest-bytes /
h'75685579a83babd71ec8ef22fa49ac873f78a708a43a674e782ad30b6598d17a'
                    ]),
                    / signature / h'861b9bfb449125742baa648bc9d148cba4
5519cca8efecf705c2165ecdecaeba8b6ce2131284e66708788d741e8779d5973fa8e2
5da49eb203c81920719da949'
                ]))
        ]
    }),
    / manifest / 3:bstr .cbor ({
        / manifest-version / 1:1,
        / manifest-sequence-number / 2:2,
        / common / 3:bstr .cbor ({
            / components / 2:[
                [h'00']
            ],
            / common-sequence / 4:bstr .cbor ([
                / directive-override-parameters / 20,{
                    / vendor-id /
1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
                    / class-id / 2:h'1492af1425695e48bf429b2d51f2ab45'
/ 1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
                    / image-digest / 3:bstr .cbor ([
                        / algorithm-id / 2 / "sha256" /,
                        / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
                    ]),
                    / image-size / 14:34768,
                } ,
```

```
                / condition-vendor-identifier / 1,15 ,
                / condition-class-identifier / 2,15
            ]),
        }),
        / install / 9:[
            / algorithm-id / 2 / "sha256" /,
            / digest-bytes /
  h'3ee96dc79641970ae46b929ccf0b72ba9536dd846020dbdc9f949d84ea0e18d2'
        ],
        / validate / 10:bstr .cbor ([
            / condition-image-match / 3,15
        ]),
        / run / 12:bstr .cbor ([
            / directive-run / 23,2
        ]),
        / text / 13:[
            / algorithm-id / 2 / "sha256" /,
            / digest-bytes /
  h'23f48b2e2838650f43c144234aee18401ffe3cce4733b23881c3a8ae2d2b66e8'
        ],
    }),
    / install / 9:bstr .cbor ([
        / directive-set-parameters / 19,{
            / uri /
  21:'http://example.com/very/long/path/to/file/file.bin',
        } ,
        / directive-fetch / 21,2 ,
        / condition-image-match / 3,15
    ]),
    / text / 13:bstr .cbor ({
        [h'00']:{
                / vendor-domain / 3:'arm.com',
                / component-description / 5:'This component is a
  demonstration. The digest is a sample pattern, not a real one.',
            }
    }),
}
```

Total size of the Envelope without COSE authentication object or Severable Elements: 233

Envelope:

```
a2025827815824820258207568 5a83babd71ec8ef22fa49ac873f78a7
08a43a674e782ad30b6598d17a0358bba70101020203585fa20281814100
0458568614a40150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492af14
25695e48bf429b2d51f2ab450358248202582000112233445566778899aa
bbccddeeff0123456789abcdeffedcba98765432100e1987d0010f020f09
820258203ee96dc79641970ae46b929ccf0b72ba9536dd846020dbdc9f94
9d84ea0e18d20a4382030f0c438217020d8202582023f48b2e2838650f43
c144234aee18401ffe3cce4733b23881c3a8ae2d2b66e8
```

Total size of the Envelope with COSE authentication object but without Severable Elements: 346

Envelope:

```
a2025898825824820258207568 5a83babd71ec8ef22fa49ac873f78a7
08a43a674e782ad30b6598d17a586fd28443a10126a05824820258207568
5579a83babd71ec8ef22fa49ac873f78a708a43a674e782ad30b6598d17a
5840861b9bfb449125742baa648bc9d148cba45519cca8efecf705c2165e
cdecaeba8b6ce2131284e66708788d741e8779d5973fa8e25da49eb203c8
1920719da9490358bba70101020203585fa2028181410004585686 14a401
50fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b
2d51f2ab450358248202582000112233445566778899aabbccddeeff0123
456789abcdeffedcba98765432100e1987d0010f020f09820258203ee96d
c79641970ae46b929ccf0b72ba9536dd846020dbdc9f949d84ea0e18d20a
4382030f0c438217020d8202582023f48b2e2838650f43c144234aee1840
1ffe3cce4733b23881c3a8ae2d2b66e8
```

Total size of Envelope with COSE authentication object: 929

Envelope with COSE authentication object:

```
   a402589882582482025820756855 79a83babd71ec8ef22fa49ac873f78a7
   08a43a674e782ad30b6598d17a586fd28443a10126a05824820258207568
   5579a83babd71ec8ef22fa49ac873f78a708a43a674e782ad30b6598d17a
   5840861b9bfb449125742baa648bc9d148cba45519cca8efecf705c2165e
   cdecaeba8b6ce2131284e66708788d741e8779d5973fa8e25da49eb203c8
   1920719da9490358bba70101020203585fa202818141000458568614a401
   50fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b
   2d51f2ab45035824820258200011223344556677889 9aabbccddeeff0123
   456789abcdeffedcba98765432100e1987d0010f020f09820258203ee96d
   c79641970ae46b929ccf0b72ba9536dd846020dbdc9f949d84ea0e18d20a
   4382030f0c438217020d8202582023f48b2e2838650f43c144234aee1840
   1ffe3cce4733b23881c3a8ae2d2b66e809583c8613a1157832687474703a
   2f2f6578616d706c652e636f6d2f766572792f6c6f6e672f706174682f74
   6f2f66696c652f66696c652e62696e1502030f0d590204a20179019d2323
   204578616d706c6520323a2053696d756c74616e656f757320446f776e6c
   6f61642c20496e7374616c6c6174696f6e2c20536563757265 20426f6f74
   2c20536576657265 64204669656c64730a0a2020202054686973 20657861
   6d706c6520636f766572732074686520666f6c6c6f77696e672074656d70
   6c617465733a0a202020200a202020202a20436f6d7061746962696c6974
   7920436865636b20287b7b74656d706c6174652d636f6d7061746962696c
   6974792d636865636b7d7d290a202020202a20536563757265 20426f6f74
   20287b7b74656d706c6174652d7365637572652d626f6f747d7d290a2020
   20202a204669726d7761726520446f776e6c6f616420287b7b6669726d77
   6172652d646f776e6c6f61642d74656d706c6174657d7d290a202020200a
   2020202054686973206578616d706c6520616c736f2064656d6f6e737472
   6174657320736576657261626c6520656c656d656e747320287b7b6f7672
   2d736576657261626c657d7d292c20616e64207465787420287b7b6d616e
   69666573742d6469676573742d746578747d7d292e814100a2036761726d
   2e636f6d0578525468697320636f6d706f6e656e742069732064656d
   6f6e7374726174696f6e2e20546865206469676573742069732061207361
   6d706c65207061747465726e2c206e6f742061207265616c206f6e652e
```

B.4. Example 3: A/B images

This example covers the following templates:

- Compatibility Check (Section 7.1)
- Secure Boot (Section 7.2)
- Firmware Download (Section 7.3)
- A/B Image Template (Section 7.11)

```
  {
     / authentication-wrapper / 2:bstr .cbor ({        digest: bstr
  .cbor ([
              / algorithm-id / 2 / "sha256" /,
```

```
            / digest-bytes /
  h'ae0c1ea689c9800a843550f38796b6fdbd52a0c78be5d26011d8e784da43d47c'
        ])        signatures: [
            bstr .cbor (18([
                    / protected / bstr .cbor ({
                        / alg / 1:-7 / "ES256" /,
                    }),
                    / unprotected / {
                    },
                    / payload / bstr .cbor ([
                        / algorithm-id / 2 / "sha256" /,
                        / digest-bytes /
  h'ae0c1ea689c9800a843550f38796b6fdbd52a0c78be5d26011d8e784da43d47c'
                    ]),
                    / signature / h'359960bae5a7de2457c8f48d3250d96d1a
  f2d36e08764b62d76f8a3f3041774b150b2c835bb1b2d7b1b2e629e1f08cc3b1b48fce
  bb8fb38182c116161e02b33f'
                ]))
        ]
    }),
    / manifest / 3:bstr .cbor ({
        / manifest-version / 1:1,
        / manifest-sequence-number / 2:3,
        / common / 3:bstr .cbor ({
            / components / 2:[
                [h'00']
            ],
            / common-sequence / 4:bstr .cbor ([
                / directive-override-parameters / 20,{
                    / vendor-id /
  1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
  be9d-e663e4d41ffe /,
                    / class-id / 2:h'1492af1425695e48bf429b2d51f2ab45'
  / 1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
                } ,
                / directive-try-each / 15,[
                    bstr .cbor ([
                        / directive-override-parameters / 20,{
                            / offset / 5:33792,
                        } ,
                        / condition-component-offset / 5,5 ,
                        / directive-override-parameters / 20,{
                            / image-digest / 3:bstr .cbor ([
                                / algorithm-id / 2 / "sha256" /,
                                / digest-bytes /
  h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
                            ]),
                            / image-size / 14:34768,
```

```
                        }
                    ]) ,
                    bstr .cbor ([
                        / directive-override-parameters / 20,{
                            / offset / 5:541696,
                        } ,
                        / condition-component-offset / 5,5 ,
                        / directive-override-parameters / 20,{
                            / image-digest / 3:bstr .cbor ([
                                / algorithm-id / 2 / "sha256" /,
                                / digest-bytes /
  h'0123456789abcdeffedcba98765432100011223344556677889aabbccddeeff'
                            ]),
                            / image-size / 14:76834,
                        }
                    ])
                ] ,
                / condition-vendor-identifier / 1,15 ,
                / condition-class-identifier / 2,15
            ]),
        }),
        / install / 9:bstr .cbor ([
            / directive-try-each / 15,[
                bstr .cbor ([
                    / directive-set-parameters / 19,{
                        / offset / 5:33792,
                    } ,
                    / condition-component-offset / 5,5 ,
                    / directive-set-parameters / 19,{
                        / uri / 21:'http://example.com/file1.bin',
                    }
                ]) ,
                bstr .cbor ([
                    / directive-set-parameters / 19,{
                        / offset / 5:541696,
                    } ,
                    / condition-component-offset / 5,5 ,
                    / directive-set-parameters / 19,{
                        / uri / 21:'http://example.com/file2.bin',
                    }
                ])
            ] ,
            / directive-fetch / 21,2 ,
            / condition-image-match / 3,15
        ]),
        / validate / 10:bstr .cbor ([
            / condition-image-match / 3,15
        ]),
```

```
      }),
  }
```

Total size of Envelope without COSE authentication object: 330

Envelope:

```
a2025827815824820258 20ae0c1ea689c9800a843550f38796b6fdbd52a0
c78be5d26011d8e784da43d47c0359011ba5010102030358aaa202818141
000458a18814a20150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492af
1425695e48bf429b2d51f2ab450f8258368614a105198400050514a20358
2482025820001122334455667788 99aabbccddeeff0123456789abcdeffe
dcba98765432100e1987d0583a8614a1051a00084400050514a203582482
0258200123456789abcdeffedcba9876543210001122334455667788 99aa
bbccddeeff0e1a00012c22010f020f095861860f82582a8613a105198400
050513a115781c687474703a2f2f6578616d706c652e636f6d2f66696c65
312e62696e582c8613a1051a00084400050513a115781c687474703a2f2f
6578616d706c652e636f6d2f66696c65322e62696e1502030f0a4382030f
```

Total size of Envelope with COSE authentication object: 443

Envelope with COSE authentication object:

```
a2025898825824820258 20ae0c1ea689c9800a843550f38796b6fdbd52a0
c78be5d26011d8e784da43d47c586fd28443a10126a0582482025820ae0c
1ea689c9800a843550f38796b6fdbd52a0c78be5d26011d8e784da43d47c
5840359960bae5a7de2457c8f48d3250d96d1af2d36e08764b62d76f8a3f
3041774b150b2c835bb1b2d7b1b2e629e1f08cc3b1b48fcebb8fb38182c1
16161e02b33f0359011ba5010102030358aaa202818141000458a18814a2
0150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf42
9b2d51f2ab450f8258368614a105198400050514a2035824820258200011
2233445566778899aabbccddeeff0123456789abcdeffedcba9876543210
0e1987d0583a8614a1051a00084400050514a2035824820258200123456 7
89abcdeffedcba9876543210001122334455667788 99aabbccddeeff0e1a
00012c22010f020f095861860f82582a8613a105198400050513a115781c
687474703a2f2f6578616d706c652e636f6d2f66696c65312e62696e582c
8613a1051a00084400050513a115781c687474703a2f2f6578616d706c65
2e636f6d2f66696c65322e62696e1502030f0a4382030f
```

B.5. Example 4: Load and Decompress from External Storage

This example covers the following templates:

- Compatibility Check (Section 7.1)
- Secure Boot (Section 7.2)
- Firmware Download (Section 7.3)

- Install (Section 7.4)

- Load & Decompress (Section 7.8)

```
{
    / authentication-wrapper / 2:bstr .cbor ({        digest: bstr
.cbor ([
            / algorithm-id / 2 / "sha256" /,
            / digest-bytes /
h'4b4c7c8c0fda76c9c9591a9db160918e2b3c96a58b0a5e4984fd4e8f9359a928'
        ])        signatures: [
            bstr .cbor (18([
                    / protected / bstr .cbor ({
                        / alg / 1:-7 / "ES256" /,
                    }),
                    / unprotected / {
                    },
                    / payload / bstr .cbor ([
                        / algorithm-id / 2 / "sha256" /,
                        / digest-bytes /
h'4b4c7c8c0fda76c9c9591a9db160918e2b3c96a58b0a5e4984fd4e8f9359a928'
                    ]),
                    / signature / h'd721cb3415f27cfeb8ef066bb6312ba758
32b57410a0c700de71cf8004ea23b9dd3c912a99fab111e9b8f2cc55c7dffcc37012de
cf72e44f69b3d3db8cc98cb6'
                ]))
        ]
    }),
    / manifest / 3:bstr .cbor ({
        / manifest-version / 1:1,
        / manifest-sequence-number / 2:4,
        / common / 3:bstr .cbor ({
            / components / 2:[
                [h'00'] ,
                [h'02'] ,
                [h'01']
            ],
            / common-sequence / 4:bstr .cbor ([
                / directive-set-component-index / 12,0 ,
                / directive-override-parameters / 20,{
                    / vendor-id /
1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
                    / class-id / 2:h'1492af1425695e48bf429b2d51f2ab45'
/ 1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
                    / image-digest / 3:bstr .cbor ([
                        / algorithm-id / 2 / "sha256" /,
                        / digest-bytes /
```

```
  h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
                        ]),
                        / image-size / 14:34768,
                    } ,
                    / condition-vendor-identifier / 1,15 ,
                    / condition-class-identifier / 2,15
                ]),
            }),
            / payload-fetch / 8:bstr .cbor ([
                / directive-set-component-index / 12,1 ,
                / directive-set-parameters / 19,{
                    / uri / 21:'http://example.com/file.bin',
                } ,
                / directive-fetch / 21,2 ,
                / condition-image-match / 3,15
            ]),
            / install / 9:bstr .cbor ([
                / directive-set-component-index / 12,0 ,
                / directive-set-parameters / 19,{
                    / source-component / 22:1 / [h'02'] /,
                } ,
                / directive-copy / 22,2 ,
                / condition-image-match / 3,15
            ]),
            / validate / 10:bstr .cbor ([
                / directive-set-component-index / 12,0 ,
                / condition-image-match / 3,15
            ]),
            / load / 11:bstr .cbor ([
                / directive-set-component-index / 12,2 ,
                / directive-set-parameters / 19,{
                    / image-digest / 3:bstr .cbor ([
                        / algorithm-id / 2 / "sha256" /,
                        / digest-bytes /
  h'0123456789abcdeffedcba98765432100011223344556677889aabbccddeeff'
                    ]),
                    / image-size / 14:76834,
                    / source-component / 22:0 / [h'00'] /,
                    / compression-info / 19:1 / "gzip" /,
                } ,
                / directive-copy / 22,2 ,
                / condition-image-match / 3,15
            ]),
            / run / 12:bstr .cbor ([
                / directive-set-component-index / 12,2 ,
                / directive-run / 23,2
            ]),
        }),
```

```
}
```

Total size of Envelope without COSE authentication object: 287

Envelope:

```
a2025827815824820258204b4c7c8c0fda76c9c9591a9db160918e2b3c96
a58b0a5e4984fd4e8f9359a9280358f1a801010204035867a20283814100
8141028141010458588 80c0014a40150fa6b4a53d5ad5fdfbe9de663e4d4
1ffe02501492af1425695e48bf429b2d51f2ab450358248202582000 1122
33445566778899aabbccddeeff0123456789abcdeffedcba98765432100e
1987d0010f020f085827880c0113a115781b687474703a2f2f6578616d70
6c652e636f6d2f66696c652e62696e1502030f094b880c0013a116011602
030f0a45840c00030f0b583a880c0213a4035824820258200123456789ab
cdeffedcba9876543210001122334455 66778899aabbccddeeff0e1a0001
2c22130116001602030f0c45840c021702
```

Total size of Envelope with COSE authentication object: 400

Envelope with COSE authentication object:

```
a2025898825824820258204b4c7c8c0fda76c9c9591a9db160918e2b3c96
a58b0a5e4984fd4e8f9359a928586fd28443a10126a05824820258204b4c
7c8c0fda76c9c9591a9db160918e2b3c96a58b0a5e4984fd4e8f9359a928
5840d721cb3415f27cfeb8ef066bb6312ba75832b57410a0c700de71cf80
04ea23b9dd3c912a99fab111e9b8f2cc55c7dffcc37012decf72e44f69b3
d3db8cc98cb60358f1a801010204035867a202838141008141028141010 4
5858880c0014a40150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492af
1425695e48bf429b2d51f2ab450358248202582000112233445566778899
aabbccddeeff0123456789abcdeffedcba98765432100e1987d0010f020f
085827880c0113a115781b687474703a2f2f6578616d706c652e636f6d2f
66696c652e62696e1502030f094b880c0013a116011602030f0a45840c00
030f0b583a880c0213a4035824820258200123456789abcdeffedcba9876
5432100011223344556677 8899aabbccddeeff0e1a00012c221301160016
02030f0c45840c021702
```

B.6. Example 5: Two Images

This example covers the following templates:

- Compatibility Check (Section 7.1)
- Secure Boot (Section 7.2)
- Firmware Download (Section 7.3)

Furthermore, it shows using these templates with two images.

```
{
    / authentication-wrapper / 2:bstr .cbor ({        digest: bstr
.cbor ([
            / algorithm-id / 2 / "sha256" /,
            / digest-bytes /
h'de7c7927a15bd2eda59cab1512875f17c9f1e9e23885ce1ac6d671eefcefa37a'
        ])        signatures: [
            bstr .cbor (18([
                    / protected / bstr .cbor ({
                        / alg / 1:-7 / "ES256" /,
                    }),
                    / unprotected / {
                    },
                    / payload / bstr .cbor ([
                        / algorithm-id / 2 / "sha256" /,
                        / digest-bytes /
h'de7c7927a15bd2eda59cab1512875f17c9f1e9e23885ce1ac6d671eefcefa37a'
                    ]),
                    / signature / h'e71e332c985fb0479f296685669d05348b
cdba8e186f25a5418f4682ea168df61661f54bf48f964577225ed455b22d277dd94de8
7c57f1baceedd6719f3d56ec'
                ]))
        ]
    }),
    / manifest / 3:bstr .cbor ({
        / manifest-version / 1:1,
        / manifest-sequence-number / 2:5,
        / common / 3:bstr .cbor ({
            / components / 2:[
                [h'00'] ,
                [h'01']
            ],
            / common-sequence / 4:bstr .cbor ([
                / directive-set-component-index / 12,0 ,
                / directive-override-parameters / 20,{
                    / vendor-id /
1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
                    / class-id / 2:h'1492af1425695e48bf429b2d51f2ab45'
/ 1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
                    / image-digest / 3:bstr .cbor ([
                        / algorithm-id / 2 / "sha256" /,
                        / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
                    ]),
                    / image-size / 14:34768,
                } ,
                / condition-vendor-identifier / 1,15 ,
```

```
                / condition-class-identifier / 2,15 ,
                / directive-set-component-index / 12,1 ,
                / directive-override-parameters / 20,{
                    / image-digest / 3:bstr .cbor ([
                        / algorithm-id / 2 / "sha256" /,
                        / digest-bytes /
  h'0123456789abcdeffedcba98765432100011223344556677889 9aabbccddeeff'
                    ]),
                    / image-size / 14:76834,
                }
            ]),
        }),
        / install / 9:bstr .cbor ([
            / directive-set-component-index / 12,0 ,
            / directive-set-parameters / 19,{
                / uri / 21:'http://example.com/file1.bin',
            } ,
            / directive-fetch / 21,2 ,
            / condition-image-match / 3,15 ,
            / directive-set-component-index / 12,1 ,
            / directive-set-parameters / 19,{
                / uri / 21:'http://example.com/file2.bin',
            } ,
            / directive-fetch / 21,2 ,
            / condition-image-match / 3,15
        ]),
        / validate / 10:bstr .cbor ([
            / directive-set-component-index / 12,0 ,
            / condition-image-match / 3,15 ,
            / directive-set-component-index / 12,1 ,
            / condition-image-match / 3,15
        ]),
        / run / 12:bstr .cbor ([
            / directive-set-component-index / 12,0 ,
            / directive-run / 23,2
        ]),
    }),
}
```

Total size of Envelope without COSE authentication object: 304

Envelope:

```
a2025827815824820258 20de7c7927a15bd2eda59cab1512875f17c9f1e9
e23885ce1ac6d671eefcefa37a03590101a601010205035895a202828141
008141010458898c0c0014a40150fa6b4a53d5ad5fdfbe9de663e4d41ffe
02501492af1425695e48bf429b2d51f2ab450358248202582000 11223344
5566778899aabbccddeeff0123456789abcdeffedcba98765432100e1987
d0010f020f0c0114a203582482025820 0123456789abcdeffedcba987654
32100011223344556677 8899aabbccddeeff0e1a00012c2209584f900c00
13a115781c687474703a2f2f6578616d706c652e636f6d2f66696c65312e
62696e1502030f0c0113a115781c687474703a2f2f6578616d706c652e63
6f6d2f66696c65322e62696e1502030f0a49880c00030f0c01030f0c4584
0c001702
```

Total size of Envelope with COSE authentication object: 417

Envelope with COSE authentication object:

```
a2025898825824820258 20de7c7927a15bd2eda59cab1512875f17c9f1e9
e23885ce1ac6d671eefcefa37a586fd28443a10126a0582482025820de7c
7927a15bd2eda59cab1512875f17c9f1e9e23885ce1ac6d671eefcefa37a
5840e71e332c985fb0479f296685669d05348bcdba8e186f25a5418f4682
ea168df61661f54bf48f964577225ed455b22d277dd94de87c57f1baceed
d6719f3d56ec03590101a601010205035895a202828141008141010458 89
8c0c0014a40150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492af1425
695e48bf429b2d51f2ab450358248202582000 11223344556677 8899aabb
ccddeeff0123456789abcdeffedcba98765432100e1987d0010f020f0c01
14a203582482025820 0123456789abcdeffedcba9876543210 0011223344
5566778899aabbccddeeff0e1a00012c2209584f900c0013a115781c6874
74703a2f2f6578616d706c652e636f6d2f66696c65312e62696e1502030f
0c0113a115781c687474703a2f2f6578616d706c652e636f6d2f66696c65
322e62696e1502030f0a49880c00030f0c01030f0c45840c001702
```

Appendix C. C. Design Rational

In order to provide flexible behavior to constrained devices, while still allowing more powerful devices to use their full capabilities, the SUIT manifest encodes the required behavior of a Recipient device. Behavior is encoded as a specialized byte code, contained in a CBOR list. This promotes a flat encoding, which simplifies the parser. The information encoded by this byte code closely matches the operations that a device will perform, which promotes ease of processing. The core operations used by most update and trusted invocation operations are represented in the byte code. The byte code can be extended by registering new operations.

The specialized byte code approach gives benefits equivalent to those provided by a scripting language or conventional byte code, with two substantial differences. First, the language is extremely high level, consisting of only the operations that a device may perform

during update and trusted invocation of a firmware image. Second, the language specifies linear behavior, without reverse branches. Conditional processing is supported, and parallel and out-of-order processing may be performed by sufficiently capable devices.

By structuring the data in this way, the manifest processor becomes a very simple engine that uses a pull parser to interpret the manifest. This pull parser invokes a series of command handlers that evaluate a Condition or execute a Directive. Most data is structured in a highly regular pattern, which simplifies the parser.

The results of this allow a Recipient to implement a very small parser for constrained applications. If needed, such a parser also allows the Recipient to perform complex updates with reduced overhead. Conditional execution of commands allows a simple device to perform important decisions at validation-time.

Dependency handling is vastly simplified as well. Dependencies function like subroutines of the language. When a manifest has a dependency, it can invoke that dependency's commands and modify their behavior by setting parameters. Because some parameters come with security implications, the dependencies also have a mechanism to reject modifications to parameters on a fine-grained level.

Developing a robust permissions system works in this model too. The Recipient can use a simple ACL that is a table of Identities and Component Identifier permissions to ensure that operations on components fail unless they are permitted by the ACL. This table can be further refined with individual parameters and commands.

Capability reporting is similarly simplified. A Recipient can report the Commands, Parameters, Algorithms, and Component Identifiers that it supports. This is sufficiently precise for a manifest author to create a manifest that the Recipient can accept.

The simplicity of design in the Recipient due to all of these benefits allows even a highly constrained platform to use advanced update capabilities.

C.1. C.1 Design Rationale: Envelope

The Envelope is used instead of a COSE structure for several reasons:

1. This enables the use of Severable Elements (Section 8.8)

2. This enables modular processing of manifests, particularly with large signatures.

3. This enables multiple authentication schemes.

4. This allows integrity verification by a dependent to be unaffected by adding or removing authentication structures.

Modular processing is important because it allows a Manifest Processor to iterate forward over an Envelope, processing Delegation Chains and Authentication Blocks, retaining only intermediate values, without any need to seek forward and backwards in a stream until it gets to the Manifest itself. This allows the use of large, Post-Quantum signatures without requiring retention of the signature itself, or seeking forward and back.

Four authentication objects are supported by the Envelope:

- COSE_Sign_Tagged

- COSE_Sign1_Tagged

- COSE_Mac_Tagged

- COSE_Mac0_Tagged

The SUIT Envelope allows an Update Authority or intermediary to mix and match any number of different authentication blocks it wants without any concern for modifying the integrity of another authentication block. This also allows the addition or removal of an authentication blocks without changing the integrity check of the Manifest, which is important for dependency handling. See Section 6.2

C.2. C.2 Byte String Wrappers

Byte string wrappers are used in several places in the suit manifest. The primary reason for wrappers it to limit the parser extent when invoked at different times, with a possible loss of context.

The elements of the suit envelope are wrapped both to set the extents used by the parser and to simplify integrity checks by clearly defining the length of each element.

The common block is re-parsed in order to find components identifiers from their indices, to find dependency prefixes and digests from their identifiers, and to find the common sequence. The common sequence is wrapped so that it matches other sequences, simplifying the code path.

A severed SUIT command sequence will appear in the envelope, so it must be wrapped as with all envelope elements. For consistency, command sequences are also wrapped in the manifest. This also allows the parser to discern the difference between a command sequence and a SUIT_Digest.

Parameters that are structured types (arrays and maps) are also wrapped in a bstr. This is so that parser extents can be set correctly using only a reference to the beginning of the parameter. This enables a parser to store a simple list of references to parameters that can be retrieved when needed.

Appendix D. D. Implementation Conformance Matrix

This section summarizes the functionality a minimal implementation needs to offer to claim conformance to this specification, in the absence of an application profile standard specifying otherwise.

The subsequent table shows the conditions.

Name	Reference	Implementation
Vendor Identifier	Section 8.7.5.2	REQUIRED
Class Identifier	Section 8.7.5.2	REQUIRED
Device Identifier	Section 8.7.5.2	OPTIONAL
Image Match	Section 8.7.6.2	REQUIRED
Image Not Match	Section 8.7.6.3	OPTIONAL
Use Before	Section 8.7.6.4	OPTIONAL
Component Offset	Section 8.7.6.5	OPTIONAL
Abort	Section 8.7.6.9	OPTIONAL
Minimum Battery	Section 8.7.6.6	OPTIONAL
Update Authorized	Section 8.7.6.7	OPTIONAL
Version	Section 8.7.6.8	OPTIONAL
Custom Condition	Section 8.7.6.10	OPTIONAL

The subsequent table shows the directives.

Name	Reference	Implementation
Set Component Index	Section 8.7.7.1	REQUIRED if more than one component
Set Dependency Index	Section 8.7.7.2	REQUIRED if dependencies used
Try Each	Section 8.7.7.3	OPTIONAL
Process Dependency	Section 8.7.7.4	OPTIONAL
Set Parameters	Section 8.7.7.5	OPTIONAL
Override Parameters	Section 8.7.7.6	REQUIRED
Fetch	Section 8.7.7.7	REQUIRED for Updater
Copy	Section 8.7.7.9	OPTIONAL
Run	Section 8.7.7.10	REQUIRED for Bootloader
Wait For Event	Section 8.7.7.11	OPTIONAL
Run Sequence	Section 8.7.7.12	OPTIONAL
Swap	Section 8.7.7.13	OPTIONAL
Fetch URI List	Section 8.7.7.8	OPTIONAL

The subsequent table shows the parameters.

Name	Reference	Implementation
Vendor ID	Section 8.7.5.3	REQUIRED
Class ID	Section 8.7.5.4	REQUIRED
Image Digest	Section 8.7.5.6	REQUIRED
Image Size	Section 8.7.5.7	REQUIRED
Use Before	Section 8.7.5.8	RECOMMENDED
Component Offset	Section 8.7.5.9	OPTIONAL
Encryption Info	Section 8.7.5.10	RECOMMENDED
Compression Info	Section 8.7.5.11	RECOMMENDED
Unpack Info	Section 8.7.5.12	RECOMMENDED
URI	Section 8.7.5.13	REQUIRED for Updater
Source Component	Section 8.7.5.14	OPTIONAL
Run Args	Section 8.7.5.15	OPTIONAL
Device ID	Section 8.7.5.5	OPTIONAL
Minimum Battery	Section 8.7.5.16	OPTIONAL
Update Priority	Section 8.7.5.17	OPTIONAL
Version Match	Section 8.7.5.18	OPTIONAL
Wait Info	Section 8.7.5.19	OPTIONAL
URI List	Section 8.7.5.20	OPTIONAL
Strict Order	Section 8.7.5.22	OPTIONAL
Soft Failure	Section 8.7.5.23	OPTIONAL
Custom	Section 8.7.5.24	OPTIONAL

Authors' Addresses

Brendan Moran
Arm Limited

EMail: Brendan.Moran@arm.com

Hannes Tschofenig
Arm Limited

EMail: hannes.tschofenig@arm.com

Henk Birkholz
Fraunhofer SIT

EMail: henk.birkholz@sit.fraunhofer.de

Koen Zandberg
Inria

EMail: koen.zandberg@inria.fr

SUIT
Internet-Draft
Intended status: Standards Track
Expires: May 6, 2021

B. Moran
H. Tschofenig
Arm Limited

November 02, 2020

Strong Assertions of IoT Network Access Requirements
draft-moran-suit-mud-01

Abstract

The Manufacturer Usage Description (MUD) specification describes the access and network functionality required a device to properly function. The MUD description has to reflect the software running on the device and its configuration. Because of this, the most appropriate entity for describing device network access requirements is the same as the entity developing the software and its configuration.

A network presented with a MUD file by a device allows detection of misbehavior by the device software and configuration of access control.

This document defines a way to link a SUIT manifest to a MUD file offering a stronger binding between the two.

Table of Contents

1. Introduction

Under [RFC8520], devices report a URL to a MUD manager in the network. RFC 8520 envisions different approaches for conveying the information from the device to the network such as:

- DHCP,

- IEEE802.1AB Link Layer Discovery Protocol (LLDP), and

- IEEE 802.1X whereby the URL to the MUD file would be contained in the certificate used in an EAP method.

The MUD manager then uses the the URL to fetch the MUD file, which contains access and network functionality required a device to properly function.

The MUD manager must trust the service from which the URL is fetched and to return an authentic copy of the MUD file. This concern may be mitigated using the optional signature reference in the MUD file. The MUD manager must also trust the device to report a correct URL. In case of DHCP and LLDP the URL is unprotected. When the URL to the MUD file is included in a certificate then it is authenticated and integrity protected. A certificate created for use with network access authentication is typically not signed by the entity that wrote the software and configured the device, which leads to conflation of local network access rights with rights to assert all network access requirements.

There is a need to bind the entity that creates the software and configuration to the MUD file because only that entity can attest the network access requirements of the device. This specification defines an extension to the SUIT manifest to include a MUD file (per reference or by value). When combining a manufacturer usage description with a manifest used for software/firmware updates (potentially augmented with attestation) then a network operator can get more confidence in the description of the access and network functionality required a device to properly function.

2. 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.

3. Architecture

The intended workflow is as follows:

- At the time of onboarding, devices report their manifest in use to the MUD manager.

- If the SUIT_MUD_container has been severed, the suit-reference-uri can be used to retrieve the complete manifest.

- The manifest authenticity is verified by the MUD manager, which enforces that the MUD file presented is also authentic and as intended by the device software vendor.

- Each time a device is updated, rebooted, or otherwise substantially changed, it will execute an attestation.

  - Among other claims in the Entity Attestation Token (EAT) [I-D.ietf-rats-eat], the device will report its software digest(s), configuration digest(s), primary manifest URI, and primary manifest digest to the MUD manager.

  - The MUD manager can then validate these attestation reports in order to check that the device is operating with the expected version of software and configuration.

  - Since the manifest digest is reported, the MUD manager can look up the corresponding manifest.

- If the MUD manager does not already have a full copy of the manifest, it can be acquired using the reference URI.

- Once a full copy of the manifest is provided, the MUD manager can verify the device attestation report and apply any appropriate policy as described by the MUD file.

4. Extensions to SUIT

To enable strong assertions about the network access requirements that a device should have for a particular software/configuration pair, we include the ability to add MUD files to the SUIT manifest. However, there are also circumstances in which a device should allow the MUD to be changed without a firmware update. To enable this, we add a MUD url to SUIT along with a subject-key identifier, according to [RFC7093], mechanism 4 (the keyIdentifier is composed of the hash of the DER encoding of the SubjectPublicKeyInfo value).

The following CDDL describes the extension to the SUIT_Manifest structure:

```
? suit-manifest-mud => SUIT_Digest
```

The SUIT_Envelope is also amended:

```
? suit-manifest-mud => bstr .cbor SUIT_MUD_container

SUIT_MUD_container = {
    ? suit-mud-url => #6.32(tstr),
    ? suit-mud-ski => SUIT_Digest,
    ? suit-mud-file => bstr
}
```

The MUD file is included verbatim within the bstr. No limits are placed on the MUD file: it may be any RFC8520-compliant file.

5. Security Considerations

This specification links MUD files to other IETF technologies, particularly to SUIT manifests, for improving security protection and ease of use. By including MUD files (per reference or by value) in SUIT manifests an extra layer of protection has been created and synchronization risks can be minimized. If the MUD file and the software/firmware loaded onto the device gets out-of-sync a device may be firewalled and, with firewalling by networks in place, the device may stop functioning.

6. IANA Considerations

suit-manifest-mud must be added as an extension point to the SUIT manifest registry.

7. Normative References

[I-D.ietf-rats-eat]
Mandyam, G., Lundblade, L., Ballesteros, M., and J. O'Donoghue, "The Entity Attestation Token (EAT)", draft-ietf-rats-eat-04 (work in progress), August 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-09 (work in progress), July 2020.

[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>.

[RFC7093] Turner, S., Kent, S., and J. Manger, "Additional Methods for Generating Key Identifiers Values", RFC 7093, DOI 10.17487/RFC7093, December 2013, <https://www.rfc-editor.org/info/rfc7093>.

[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>.

[RFC8520] Lear, E., Droms, R., and D. Romascanu, "Manufacturer Usage Description Specification", RFC 8520, DOI 10.17487/RFC8520, March 2019, <https://www.rfc-editor.org/info/rfc8520>.

Authors' Addresses

Brendan Moran
Arm Limited

EMail: Brendan.Moran@arm.com

Hannes Tschofenig
Arm Limited

EMail: hannes.tschofenig@arm.com

SUIT
Internet-Draft
Intended status: Informational
Expires: April 26, 2021

B. Moran
Arm Limited

October 23, 2020

Secure Reporting of Update Status
draft-moran-suit-report-00

Abstract

The Software Update for the Internet of Things (SUIT) manifest provides a way for many different update and boot workflows to be described by a common format. However, this does not provide a feedback mechanism for developers in the event that an update or boot fails.

This specification describes a lightweight feedback mechanism that allows a developer in possession of a manifest to reconstruct the decisions made and actions performed by a manifest processor.

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 April 26, 2021.

Copyright Notice

Table of Contents

1. Introduction

A SUIT manifest processor can fail to install or boot an update for many reasons. Frequently, the error codes generated by such systems fail to provide developers with enough information to find root causes and produce corrective actions, resulting in extra effort to reproduce failures. Logging the results of each SUIT command can simplify this process.

While it is possible to report the results of SUIT commands through existing logging or attestation mechanisms, this comes with several drawbacks:

- data inflation, particularly when designed for text-based logging
- missing information elements
- missing support for multiple components

The CBOR objects defined in this document allow devices to:

- report a trace of how an update was performed
- report expected vs. actual values for critical checks
- describe the installation of complex multi-component architectures

This document provides a definition of a SUIT-specific logging container that may be used in a variety of scenarios.

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.

Terms used in this specification include:

- Boot: initialization of an executable image. Although this specification refers to boot, any boot-specific operations described are equally applicable to starting an executable in an OS context.

3. The SUIT Record

If the developer can be assumed to have a copy of the manifest, then they need little information to reconstruct what the manifest processor has done. They simply need any data that influences the control flow of the manifest. The manifest only supports the following control flow primitives:

- Set Component/Dependency Index
- Set/Override Parameters
- Try-Each
- Run Sequence
- Conditions.

Of these, only conditions change the behavior of the processor from the default, and then only when the condition fails.

Then, to reconstruct the flow of a manifest, all a developer needs is a list of metadata about failed conditions:

- the current manifest
- the current section
- the offset into the current section
- the current component index
- the "reason" for failure

Most conditions compare a parameter to an actual value, so the "reason" is typically simply the actual value.

Since it is possible that a non-condition command may fail in an exceptional circumstance, this must be included as well. To accommodate for a failed command, the list of failed conditions is expanded to be a list of failed commands instead. In the case of a command failure, the failure reason is typically a numeric error code.

Reconstructing what a device has done in this way is compact, however it requires some reconstruction effort. This is an issue that can be solved by tooling.

```
suit-record = {
    suit-record-manifest-id        => [* uint ],
    suit-record-manifest-section   => int,
    suit-record-section-offset     => uint,
    (
        suit-record-component-index  => uint //
        suit-record-dependency-index => uint
    ),
    suit-record-failure-reason     => SUIT_Parameters,
}
```

suit-record-manifest-id is used to identify which manifest contains the command that caused the record to be generated. The manifest id is a list of integers that form a walk of the manifest tree, starting at the root. An empty list indicates that the command was contained in the root manifest. If the list is not empty, the command was contained in one of the root manifest's dependencies, or nested even further below that.

For example, suppose that the root manifest has 3 dependencies and each of those dependencies has 2 dependencies of its own:

- Root
  - Dependency A
    - Dependency A0
    - Dependency A1
  - Dependency B
    - Dependency B0

        * Dependency B1

    o Dependency C

        * Dependency C0

        * Dependency C1

A manifest-id of [1,0] would indicate that the current command was contained within Dependency B0. Similarly, a manifest-id of [2,1] would indicate Dependency C1

suit-record-manifest-section indicates which section of the manifest was active. This is used in addition to an offset so that the developer can index into severable sections in a predictable way. The value of this element is the value of the key that identified the section in the manifest.

suit-record-section-offset is the number of bytes into the current section at which the current command is located.

suit-record-component-index is the index of the component that was specified at the time that the report was generated. This field is necessary due to the availability of set-current-component values of True and a list of components. Both of these values cause the manifest processor to loop over commands using a series of component-ids , so the developer needs to know which was selected when the command executed.

suit-record-dependency-index is similar to suit-record-component-index but is used to identify the dependency that was active.

suit-record-failure-reason contains the reason for the command failure. For example, this could be the actual value of a SUIT_Digest or class identifier. This is encoded in a SUIT_Parameters block as defined in [I-D.ietf-suit-manifest]. If it is not a condition that has failed, but a directive, then the value of this element is an integer that represents an implementation-defined failure code.

4. The SUIT Report

Some metadata is common to all records, such as the root manifest: the manifest that is the entry-point for the manifest processor. This metadata is aggregated with a list of suit-records.

```
suit-report = {
    suit-report-manifest-digest => SUIT_Digest,
    ? suit-report-manifest-uri  => tstr,
    suit-report-records         => [ *suit-record ]
}
```

suit-report-manifest-digest provides a SUIT_Digest (as defined in [I-D.ietf-suit-manifest]) that is the characteristic digest of the Root manifest.

suit-report-manifest-uri provides the reference URI that was provided in the root manifest.

suit-report-records is a list of 0 or more SUIT Records. Because SUIT Records are only generated on failure, in simple cases this can be an empty list.

5. IANA Considerations

IANA is requested to allocate a CBOR tag for the SUIT Report.

6. Security Considerations

The SUIT Report should either be carried over a secure transport, or signed, or both. Ideally, attestation should be used to prove that the report was generated by legitimate hardware.

7. Acknowledgements

8. Normative References

[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-09 (work in progress), July 2020.

[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>.

[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>.

Author's Address

Brendan Moran
Arm Limited

EMail: Brendan.Moran@arm.com