SUIT B. Moran
Internet-Draft M. Meriac
Intended status: Informational H. Tschofenig
Expires: September 6, 2018 Arm Limited
March 05, 2018

A Firmware Update Architecture for Internet of Things Devices
draft-moran-suit-architecture-03

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.

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.

Copyright Notice

Table of Contents

1. Introduction

When developing IoT devices, one of the most difficult problems to solve is how to update the 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 or where manual intervention is cost prohibitive or otherwise difficult. The need for a firmware update may be to fix bugs in software, to add new functionality, or to re-configure the device.

The firmware update process has to ensure that

- The firmware image is authenticated and attempts to flash a malicious firmware image 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 plaintext binary is often one of the first steps for an attack to mount an attack.

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 RFC 2119 [RFC2119].

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 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. The image may consist of a differential update for performance reasons. Firmware is the more universal term. Both terms are used in this document and are interchangeable.

The following entities are used:

- Author: The author is the entity that creates the firmware image, signs and/or encrypts it and attaches a manifest to it. The author is most likely a developer using a set of tools.

- Device: The device is the recipient of the firmware image and the manifest. The goal is to update the firmware of the device.

- Untrusted Storage: Firmware images and manifests are stored on untrusted fileservers or cloud storage infrastructure. Some deployments may require storage of the firmware images/manifests to be stored on various entities before they reach the device.

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

- Uses state-of-the-art security mechanisms

- Rollback attacks must be prevented.

- High reliability

- Operates with a small bootloader

- Small Parsers

- Minimal impact on existing firmware formats

- Robust permissions

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

For an update to be broadcast friendly, it cannot rely on link layer, network layer, or transport layer security. In addition, the same message 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. Uses state-of-the-art security mechanisms

End-to-end security between the author and the device, as shown in Section 5, is used to ensure that the device can verify firmware images and manifests produced by authorized authors.

The use of post-quantum secure signature mechanisms, such as hash-based signatures, should be explored. A mandatory-to-implement set of algorithms has to be defined offering a key length of 112-bit symmetric key or security or more, as outlined in Section 20 of RFC 7925. This corresponds to a 233 bit ECC key or a 2048 bit RSA key.

If the firmware image is to be encrypted, 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 be an absolute minimum.

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 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. Operates with a small bootloader

The bootloader must be minimal, containing only flash support, cryptographic primitives and optionally a recovery mechanism. The recovery mechanism is used in case the update process failed and may include support for firmware updates over serial, USB or even a limited version of wireless connectivity standard like a limited Bluetooth Smart. Such a recovery mechanism must provide security at least at the same level as the full featured firmware update functionalities.

The bootloader needs to verify the received manifest and to install the bootable firmware image. The bootloader should not require updating since a failed update poses a risk in reliability. If more functionality is required in the bootloader, it must use a two-stage bootloader, with the first stage comprising the functionality defined above.

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.

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 which are required to validate at least one signature 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

A device may have many modules that require updating individually. It may also need to trust several actors in order to authorize an update. For example, a firmware author may not have the authority to install firmware on a device in critical infrastructure without the authorization of a device operator. In this case, the device should reject firmware updates unless they are signed both by the firmware author and by the device operator. To facilitate complex use-cases such as this, updates require several permissions.

4. Claims

When a simple set of permissions fails to encapsulate the rules required for a device make decisions about firmware, claims can be used instead. Claims represent a form of policy. Several claims can be used together, when multiple actors should have the rights to set policies.

Some example claims are:

- Trust the actor identified by the referenced public key.

- Three actors are trusted identified by their public keys. Signatures from at least two of these actors are required to trust a manifest.

- The actor identified by the referenced public key is authorized to create secondary policies

The baseline claims for all manifests are described in Appendix A. In summary, they 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 install firmware with metadata signed by a trusted actor.

- Only allow an actor to exercise rights on the device via a manifest if that actor has signed the manifest.

- Only allow a firmware installation if all required rights have been met through signatures (one or more) or manifest dependencies (one or more).

- Use the instructions provided by the manifest to install the firmware.

- Any authorized actor may redirect any URI.

- Install any and all firmware images that are linked together with manifest dependencies.

- Choose the mechanism to install the firmware, based on the type of firmware it is.

5. Architecture

We start the architectural description with the security model. It is based on end-to-end security. Figure 1 illustrates the security model where a firmware image and the corresponding manifest are created by an author and verified by the device. The firmware image is integrity protected and may be encrypted. The manifest is integrity protected and authenticated. When the author is ready to distribute the firmware image it is conveyed using some communication channel to the device, which will typically involve the use of untrusted storage. Examples of untrusted storage are FTP servers, Web servers or USB sticks.

```
                              +-----------+
 +--------+  Firmware Image  |           |   Firmware Image  +--------+
 |        |  + Manifest      | Untrusted |   + Manifest      |        |
 | Device |<-----------------| Storage   |<------------------| Author |
 |        |                  |           |                   |        |
 +--------+                  +-----------+                   +--------+
      ^                                                          *
      *                                                          *
      ************************************************************
                          End-to-End Security
```

Figure 1: End-to-End Security.

Whether the firmware image and the manifest is pushed to the device or fetched by the device is outside the scope of this work and existing device management protocols can be used for efficiently distributing this information.

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

- To accept an update, a device needs to decide whether the author signing the firmware image and the manifest is authorized to make the updates. We use public key cryptography to accomplish this. The device verifies the signature covering the manifest using a digital signature algorithm. The device is provisioned with a trust anchor that is used to validate the digital signature produced by the author. This trust anchor is potentially different from the trust anchor used to validate the digital signature produced for other protocols (such as device management protocols). This trust anchor may be provisioned to the device during manufacturing or during commissioning.

- 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.

There are different types of delivery modes, which are illustrates 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 server for download of the firmware. It is also applicable when the firmware update happens via a USB stick or via Bluetooth Smart. Figure 2 shows this delivery mode graphically.

```
              /------------\                 /------------\
             /Manifest with \               /Manifest with \
             |attached      |               |attached      |
             \firmware image/               \firmware image/
              \------------/  +-----------+  \------------/
 +--------+                   |           |                 +--------+
 |        |<..................| Untrusted |<................|        |
 | Device |                   | Storage   |                 | Author |
 |        |                   |           |                 |        |
 +--------+                   +-----------+                 +--------+
```

Figure 2: Manifest with attached firmware.

Figure 3 shows an option for remotely updating a device where the device fetches the firmware image from some file server. The manifest itself is delivery independently and provides information about the firmware image(s) to download.

```
                                /------------\
                               /              \
                               |   Manifest   |
                               \              /
    +--------+                  \------------/                +--------+
    |        |<..............................................>|        |
    | Device |                                             -- | Author |
    |        |<-                                         ---  |        |
    +--------+  --                                     ---    +--------+
                  --                                 ---
                    ---                           ---
                       --       +-----------+    --
                         --     |           |  --
          /------------\   --   | Untrusted |<-    /------------\
         /              \    -- | Storage   |     /              \
         |   Firmware   |       |           |     |   Firmware   |
         \              /       +-----------+     \              /
          \------------/                           \------------/
```

Figure 3: 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 intented to be applied to,

- information about when the firmware update has to be applied,

- information about when the manifest was created,

- dependencies to 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.

The manifest format is described in a companion document.

7. Example Flow

The following example message flow illustrates the interaction for distributing a firmware image to a device starting with an author uploading the new firmware to untrusted storage and creating a manifest.

```
+--------+    +-----------------+      +------+
| Author |    |Untrusted Storage|      |Device|
+--------+    +-----------------+      +------+
  |                   |                     |
  | Create Firmware   |                     |
  |---------------    |                     |
  |              |    |                     |
  |<--------------    |                     |
  |                   |                     |
  | Upload Firmware   |                     |
  |------------------>|                     |
  |                   |                     |
  | Create Manifest   |                     |
  |----------------   |                     |
  |               |   |                     |
  |<---------------   |                     |
  |                   |                     |
  | Sign Manifest     |                     |
  |--------------     |                     |
  |             |     |                     |
```

```
     |<-------------     |                     |
     |                   |                     |
     | Upload Manifest   |                     |
     |------------------>|                     |
     |                   |                     |
     |                   |   Query Manifest    |
     |                   |<--------------------|
     |                   |                     |
     |                   |   Send Manifest     |
     |                   |-------------------->|
     |                   |                     |
     |                   |                     | Validate Manifest
     |                   |                     |------------------
     |                   |                     |                 |
     |                   |                     |<-----------------
     |                   |                     |
     |                   |  Request Firmware   |
     |                   |<--------------------|
     |                   |                     |
     |                   | Send Firmware       |
     |                   |-------------------->|
     |                   |                     |
     |                   |                     | Verify Firmware
     |                   |                     |---------------
     |                   |                     |              |
     |                   |                     |<--------------
     |                   |                     |
     |                   |                     | Store Firmware
     |                   |                     |--------------
     |                   |                     |             |
     |                   |                     |<-------------
     |                   |                     |
     |                   |                     | Reboot
     |                   |                     |-------
     |                   |                     |      |
     |                   |                     |<------
     |                   |                     |
     |                   |                     | Bootloader validates
     |                   |                     | Firmware
     |                   |                     |----------------------
     |                   |                     |                     |
     |                   |                     |<---------------------
     |                   |                     |
     |                   |                     | Bootloader activates
     |                   |                     | Firmware
     |                   |                     |----------------------
     |                   |                     |                     |
     |                   |                     |<---------------------
```

```
     |                   |                     |
     |                   |                     | Bootloader transfers
     |                   |                     | control to new Firmware
     |                   |                     |----------------------
     |                   |                     |                     |
     |                   |                     |<---------------------
     |                   |                     |
```

Figure 4: Example Flow for a Firmware Upate.

8. IANA Considerations

This document does not require any actions by IANA.

9. 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]. This document (and associated specifications) offer a standardized firmware manifest format providing end-to-end security from the author to the device.

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's 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 digitial signature protecting the manifest.

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

10. Mailing List Information

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

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

11. Acknowledgements

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

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

- Marti Bolivar

- Andrzej Puzdrowski

- Markus Gueller

We would also like to thank the WG chairs, Russ Housley, David Waltermire, Dave Thaler and the responsible security area director, Kathleen Moriarty, for their support and their reviews.

12. References

12.1. 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>.

12.2. Informative References

[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>.

[STRIDE] Microsoft, "The STRIDE Threat Model", January 2018.

12.3. URIs

[1] mailto:suit@ietf.org

Appendix A. Threat Model, User Stories, Security Requirements, and Usability Requirements

A.1. Threat Model

This appendix aims 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.

A.2. Threat Descriptions

A.2.1. Threat MFT1: Old Firmware

Classification: Escalation 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: MFSR1

A.2.2. Threat MFT2: 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: MFSR2

A.2.3. Threat MFT3: Offline device + Old Firmware

Classification: Escalation 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.

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

Mitigated by: MFSR3

A.2.4. Threat MFT4: The target device misinterprets the type of payload

Classification: Denial of Service

If a device misinterprets the type 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 escalation of privilege and potentially expand this to all types of threat.

Mitigated by: MFSR4

A.2.5. Threat MFT5: 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 escalation of privilege and potentially expand this to all types of threat.

Mitigated by: MFSR4

A.2.6. Threat MFT6: Redirection

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: MFSR4

A.2.7. Threat MFT7: Payload Verification on Boot

Classification: All Types

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.

Mitigated by: MFSR4

A.2.8. Threat MFT8: Unauthenticated Updates

Classification: 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: MFSR5

A.2.9. Threat MFT9: Unexpected Precursor images

Classification: Denial of Service

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.

Threat Escalation: An attacker that can cause a device to install a payload against the wrong precursor image could gain escalation of privilege and potentially expand this to all types of threat.

Mitigated by: MFSR4

A.2.10. Threat MFT10: Unqualified Firmware

Classification: Denial of Service, Escalation of Privilege

This threat can appear in several ways, however it is ultimately about interoperability of devices with other systems. The owner or operator of a network needs to approve firmware for their network in order to ensure interoperability with other devices on the network, or the network itself. If the firmware is not qualified, it may not work. Therefore, if a device installs firmware without the approval of the network owner or operator, this is a threat to devices and the network.

Example 1: We assume that OEMs expect the rights to create firmware, but that Operators expect the rights to qualify firmware as fit-for-purpose on their networks.

An attacker obtains a manifest for a device on Network A. They send that manifest to a device on Network B. Because Network A and Network B are different, and the firmware has not been qualified for Network B, the target device is disabled by this unqualified, but signed firmware.

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

Example 2: Multiple devices that interoperate are used on the same network. Some devices are manufactured by OEM A and other devices by OEM B. These devices communicate with each other. A new firmware is released by OEM A that breaks compatibility with OEM B devices. An attacker sends the new firmware to the OEM A devices 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.

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

Mitigated by: MFSR6

A.2.11. Threat MFT11: 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: MFSR7

A.3. Security Requirements

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

A.3.1. Security Requirement MFSR1: 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. Devices MUST reject manifests with sequence numbers smaller than any onboard sequence number.

N.B. 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 MFT1 Implemented by: Manifest Field: Timestamp

A.3.2. Security Requirement MFSR2: 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 MFT2 Implemented by: Manifest Fields: Vendor ID Condition, Class ID Condition

A.3.3. Security Requirement MFSR3: Best-Before Timestamps

Firmware 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 a best-before timestamp, the device MUST reject the manifest if current time is larger than the best-before time.

Mitigates: Threat MFT3 Implemented by: Manifest Field: Best-Before timestamp condition

A.3.4. Security Requirement MFSR4: Signed Payload Descriptor

All descriptive information about the payload MUST be signed. This MUST include:

- The type of payload (which may be independent of format)
- The location to store the payload
- The payload digest, in each state of installation (encrypted, plaintext, installed, etc.)
- The payload size
- The payload format
- Where to obtain the payload
- All instructions or parameters for applying the payload
- Any rules that identify whether or not the payload can be used on this device

Mitigates: Threats MFT4, MFT5, MFT6, MFT7, MFT9 Implemented by: Manifest Fields: Vendor ID Condition, Class ID Condition, Precursor Image Digest Condition, Payload Format, Storage Location, URIs, Digests, Size

A.3.5. Security Requirement MFSR5: Cryptographic Authenticity

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

Mitigates: Threat MFT8 Implemented by: Signature

A.3.6. Security Requirement MFSR6: 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 MFSR5 are acceptable but need to follow the end-to-end security model.

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 an OEM and an Operator, respectively, then the firmware cannot be installed without proof of rights from both the OEM and the Operator.

Mitigates: MFT10 Implemented by: Signature

A.3.7. Security Requirement MFSR7: Firmware encryption

Firmware images must be encrypted to prevent third parties, including attackers, from reading the content of the firmware image and to reverse engineer the code.

Mitigates: MFT11 Implemented by: Manifest Field: Content Key Distribution Method

A.4. User Stories

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

A.4.1. Use Case MFUC1: Installation Instructions

As an OEM for IoT devices, 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:

- Specify a package handler

- Use a table of hashes to ensure that each block of the payload is validate before writing.

- Run post-processing script after the update is installed

- 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: MFUR1

A.4.2. Use Case MFUC2: Reuse Local Infrastructure

As an Operator of IoT devices, I would like to tell my devices to look at my own infrastructure for payloads so that I can manage the traffic generated by firmware updates on my network and my peers' networks.

Satisfied by: MFUR2, MFUR3

A.4.3. Use Case MFUC3: Modular Update

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

Satisfied by: MFUR3

A.4.4. Use Case MFUC4: Multiple Authorisations

As an Operator, I want to ensure the quality of a firmware update before installing it, so that I can ensure a high standard of reliability on my network. The OEM may restrict my ability to create firmware, so I cannot be the only authority on the device.

Satisfied by: MFUR4

A.4.5. Use Case MFUC5: Multiple Payload Formats

As a OEM or Operator of devices, 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: MFUR5

A.4.6. Use Case MFUC6: IP Protection

As an OEM or developer for IoT devices, I want to protect the IP contained in the firmware image, such as the utilized algorithms. The need for protecting IP may have also been imposed on my due to the use of some third party code libraries.

Satisfied by: MFSR7

A.5. Usability Requirements

The following usability requirements satisfy the user stories listed above.

A.5.1. Usability Requirement MFUR1

It must be possible to write additional installation instructions into the manifest.

Satisfies: Use-Case MFUC1 Implemented by: Manifest Field: Directives

A.5.2. Usability Requirement MFUR2

It must be possible to redirect payload fetches. This applies where two manifests are used in conjunction. For example, an OEM manifest specifies a payload and signs it, and provides a URI for that payload. An Operator creates a second manifest, with a dependency on the first. They use this second manifest to override the URIs provided by the OEM, directing them into their own infrastructure instead.

Satisfies: Use-Case MFUC2 Implemented by: Manifest Field: Aliases

A.5.3. Usability Requirement MFUR3

It MUST be possible to link multiple manifests together so that a multi-component 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.

Satisfies: Use-Case MFUC2, MFUC3 Implemented by: Manifest Field: Dependencies

A.5.4. Usability Requirement MFUR4

It MUST be possible to sign a manifest multiple times so that signatures from multiple parties with different permissions can be required in order to authorise installation of a manifest.

Satisfies: Use-Case MFUC4 Implemented by: COSE Signature (or similar)

A.5.5. Usability Requirement MFUR5

The manifest format MUST accommodate any payload format that an operator or OEM wishes to use. Some examples of payload format would be:

- Binary

- Elf

- Differential

- Compressed

- Packed configuration

Satisfies: Use-Case MFUC5 Implemented by: Manifest Field: Payload Format

A.6. Manifest Fields

Each manifest field is anchored in a security requirement or a usability requirement. The manifest fields are described below and justified by their requirements.

A.6.1. Manifest Field: Timestamp

A monotonically increasing sequence number. For convenience, a timestamp implements the requirement of a monotonically increasing sequence number. This allows global synchronisation of sequence numbers without any additional management.

Implements: Security Requirement MFSR1.

A.6.2. Manifest Field: Vendor ID Condition

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 type 5 UUIDs with the vendor's domain name and the UUID DNS prefix. Other options include type 1 and type 4 UUIDs.

Implements: Security Requirement MFSR2, MFSR4.

A.6.3. Manifest Field: Class ID Condition

Class Identifiers MUST be unique within a Vendor ID. This is to prevent similarly, or identically named devices colliding in their customer's infrastructure. Recommended practice is to use type 5 UUIDs with the model, hardware revision, etc. and use the Vendor ID as the UUID prefix. Other options include type 1 and type 4 UUIDs. A device "Class" is defined as any device that can run the same firmware without modification. Classes MAY be implemented in a more granular way. Classes MUST NOT be implemented in a less granular way. Class ID can encompass model name, hardware revision, software revision. Devices MAY have multiple Class IDs.

Implements: Security Requirement MFSR2, MFSR4.

A.6.4. Manifest Field: Precursor Image Digest Condition

When a precursor image is required by the payload format, a precursor image digest condition MUST be present in the conditions list.

Implements: Security Requirement MFSR4

A.6.5. Manifest Field: Best-Before timestamp condition

This field tells a device the last application time. This is only usable in conjunction with a secure clock.

Implements: Security Requirement MFSR3

A.6.6. Manifest Field: Payload Format

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

Implements: Security Requirement MFSR4, Usability Requirement MFUR5

A.6.7. Manifest Field: Storage Location

This field tells the device which component is being updated. The device can use this to establish which permissions are necessary and the physical location to use.

Implements: Security Requirement MFSR4

A.6.8. Manifest Field: URIs

This field is a list of weighted URIs that the device uses to select where to obtain a payload.

Implements: Security Requirement MFSR4

A.6.9. Manifest Field: Digests

This field is a map of digests, each for a separate stage of installation. This allows the target device to ensure authenticity of the payload at every step of installation.

Implements: Security Requirement MFSR4

A.6.10. Manifest Field: Size

The size of the payload in bytes.

Implements: Security Requirement MFSR4

A.6.11. Manifest Field: Signature

This is not strictly a manifest field. Instead, the manifest is wrapped by a standardised authentication container, such as a COSE or CMS signature object. The authentication container MUST support multiple actors and multiple authentications.

Implements: Security Requirement MFSR5, MFSR6, MFUR4

A.6.12. Manifest Field: Directives

A list of instructions that the device should execute, in order, when installing the payload.

Implements: Usability Requirement MFUR1

A.6.13. Manifest Field: Aliases

A list of URI/Digest pairs. A device should build an alias table while paring a manifest tree and treat any aliases as top-ranked URIs for the corresponding digest.

Implements: Usability Requirement MFUR2

A.6.14. Manifest Field: Dependencies

A list of URI/Digest pairs that refer to other manifests by digest. The manifests that are linked in this way must be acquired and installed simultaneously in order to form a complete update.

Implements: Usability Requirement MFUR3

A.6.15. Manifest Field: Content Key Distribution Method

Encrypting firmware images requires symmetric content encryption keys. Since there are several methods to protect or distribute the symmetric content encryption keys, the manifest contains a field for the Content Key Distribution Method. One examples for such a Content Key Distribution Method is the usage of Key Tables, pointing to content encryption keys, which themselves are encrypted using the public keys of devices.

Implements: Security Requirement MFSR7.

Authors' Addresses

Brendan Moran
Arm Limited

EMail: Brendan.Moran@arm.com

Milosch Meriac
Arm Limited

EMail: Milosch.Meriac@arm.com

Hannes Tschofenig
Arm Limited

EMail: hannes.tschofenig@gmx.net

SUIT B. Moran
Internet-Draft H. Tschofenig
Intended status: Informational Arm Limited
Expires: January 9, 2020 H. Birkholz
Fraunhofer SIT
July 08, 2019

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

Abstract

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

Status of This Memo

Copyright Notice

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, such as the Lightweight Machine-to-Machine (LwM2M) protocol offering device management of IoT devices. Equally important is the inclusion of meta-data about the conveyed firmware image (in the form of a manifest) and the use of end-to-end security protection to detect modifications and (optionally) to make reverse engineering more difficult. End-to-end security allows the author, who builds the firmware image, to be sure that no other party (including potential adversaries) can install firmware updates on IoT devices without adequate privileges. This authorization process is ensured by the use of dedicated symmetric or asymmetric keys installed on the IoT device: for use cases where only integrity protection is required it is sufficient to install a trust anchor on the IoT device. For confidentiality protected firmware images it is additionally required to install either one or multiple symmetric or asymmetric keys on the IoT device. Starting security protection at the author is a risk mitigation technique so firmware images and manifests can be stored on untrusted respositories; it also reduces the scope of a compromise of any repository or intermediate system to be no worse than a denial of service.

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

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

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

2. Conventions and Terminology

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

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

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

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

- Manifest: A piece of information that describes one or more payloads, one or more resources, and the processors needed to transform resources into payloads.

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

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

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

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

- Directive: An action for the Recipient to perform.

- Command: A Condition or a Directive.

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

3. Distributing firmware

Distributing firmware in a multi-party environment is a difficult operation. Each party requires a different subset of data. Some data may not be accessible to all parties. Multiple signatures may be required from parties with different authorities. This topic is covered in more depth in [Architecture].

4. Workflow of a device applying a firmware update

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

1. Verify the signature of the manifest

2. Verify the applicability of the manifest

3. Resolve dependencies

4. Fetch payload(s)

5. Install payload(s)

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

1. Verify image(s)

2. Load image(s)

3. Run image(s)

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

5. SUIT manifest goals

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

1. Meet the requirements defined in [Information].

2. Simple to parse on a constrained node

3. Simple to process on a constrained node

4. Compact encoding

5. Comprehensible by an intermediate system

6. Expressive enough to enable advanced use cases on advanced nodes

7. Extensible

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

1. the Firmware Author to reason about releasing a firmware.

2. the Network Operator to reason about compatibility of a firmware.

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

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

5. the Plant Manager to reason about timing and acceptance of firmware updates.

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

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

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

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

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

6. SUIT manifest design overview

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

The specialised byte code approach gives benefits equivalent to those provided by a scripting language or conventional byte code, with two substantial differences. First, the language is extremely high level, consisting of only the operations that a device may perform during update and trusted execution of a firmware image. Second, the language specifies behaviours in a linearised form, without reverse branches. Conditional processing is supported, and parallel and out-of-order processing may be performed by sufficiently capable devices.

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

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

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

Developing a robust permissions system works in this model too. The Recipient can use a simple ACL that is a table of Identities and Component Identifier permissions to ensure that only manifests authenticated by the appropriate identity have access to operate on a component.

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

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

6.1. Manifest Design Evaluation

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

Goal evaluation:

1. Each command and condition is anchored to a manifest information element in [Information]

2. The use of a byte code encourages flat encoding and reduces nesting depth. This promotes a simple encoding.

3. The encoded information closely matches the operations that a device will perform, making the format easy to process.

4. Encoding efficiency exceeds 50% when compared to raw data.

5. Tooling will be required to reason about the manifest.

6. The core operations used by most update and trusted execution operations are represented in the byte code. The use cases listed in [Information] are enabled.

7. Registration of new standard byte code identifiers enables extension in a comprehensible way.

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

6.2. Severable Elements

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

Elements are made severable by removing them from the manifest, encoding them in a bstr, and placing a SUIT_Digest of the bstr in the manifest so that they can still be authenticated. The SUIT_Digest typically consumes 4 bytes more than the size of the raw digest, therefore elements smaller than (Digest Bits)/8 + 4 SHOULD never 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.

6.3. Conventions

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

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

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

6.4. Payloads

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

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

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

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

1. Payloads that need a specialised installer must indicate this in suit-payload-info-unpack.

2. Payloads that need specialised verification must indicate this in the SUIT_Parameter_Image_Digest by indicating a SUIT_Digest algorithm that correctly validates their information.

7. Manifest Structure

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

1. The outer wrapper

    1. The authentication wrapper

    2. The manifest

        1. Critical Information

        2. Information shared by all command sequences

            1. List of dependencies

            2. List of payloads

            3. List of payloads in dependencies

            4. Common list of conditions, directives

        3. Dependency resolution Reference or list of conditions, directives

        4. Payload fetch Reference or list of conditions, directives

        5. Installation Reference or list of conditions, directives

        6. Verification conditions/directives

        7. Load conditions/directives

        8. Run conditions/directives

        9. Text / Reference

        10. COSWID / Reference

    3. Dependency resolution conditions/directives

    4. Payload fetch conditions/directives

    5. Installation conditions/directives

    6. Text

    7. COSWID / Reference

    8. Intermediate Certificate(s) / CWTs

    9. Inline Payload(s)

7.1. Outer wrapper

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

The CDDL that describes the wrapper is below

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

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

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

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

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

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

The outer wrapper MUST contain only one of

- a plaintext manifest: SUIT_Manifest

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

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

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

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

suit-manifest-encrypted contains a ciphertext manifest.

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

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

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

7.2. Manifest

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

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

The following CDDL fragment defines the manifest.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.3. SUIT_Dependency

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

The following CDDL describes the SUIT_Dependency structure.

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

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

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

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

7.4. SUIT_Component_Reference

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

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

7.5. Manifest Parameters

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

1. Integers
2. Byte strings
3. Booleans

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

The defined manifest parameters are described below.

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

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

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

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

7.5.1. SUIT_Parameter_Strict_Order

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

7.5.2. SUIT_Parameter_Coerce_Condition_Failure

When executing a command sequence inside SUIT_Run_Sequence and a condition failure occurs, the manifest processor aborts the sequence. If Coerce Condition Failure is True, it returns Success. Otherwise, it returns the original condition failure. SUIT_Parameter_Coerce_Condition_Failure is scoped to the enclosing SUIT_Directive_Run_Sequence. Its value is discarded when SUIT_Directive_Run_Sequence terminates.

7.6. SUIT_Parameter_Encryption_Info

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

7.7. SUIT_Parameter_Compression_Info

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

SUIT_Parameter_Compression_Info is defined by the following CDDL:

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


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

7.8. SUIT_Parameter_Unpack_Info

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

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

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

7.9. SUIT_Parameters CDDL

The following CDDL describes all SUIT_Parameters.

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

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

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

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

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

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

7.10. SUIT_Command_Sequence

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

1. Dependency Resolution

2. Payload Fetch

3. Payload Installation

4. Image Validation

5. Image Loading

6. Run or Boot

Each of these follows exactly the same structure to ensure that the parser is as simple as possible.

Lists of commands are constructed from two kinds of element:

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

2. Directives that MUST be executed.

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

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

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

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

Each condition and directive is composed of:

1. A command code identifier

2. An argument block

Argument blocks are defined for each type of command.

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

To facilitate optional conditions, a special directive is provided. It runs several new lists of conditions/directives, one after another, that are contained as an argument to the directive. By default, it assumes that a failure of a condition should not indicate a failure of the update/boot, but a parameter is provided to override this behaviour.

7.11. SUIT_Condition

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

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

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

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

7.11.1. Identifier Conditions

There are three identifier-based conditions: suit-condition-vendor-identifier, suit-condition-class-identifier, and suit-condition-device-identifier. Each of these conditions match a RFC 4122 [RFC4122] UUID that MUST have already been set as a parameter. The installing device MUST match the specified UUID in order to consider the manifest valid. These identifiers MAY be scoped by component.

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

7.11.2. suit-condition-image-match

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

7.11.3. suit-condition-image-not-match

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

7.11.4. suit-condition-use-before

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

7.11.5. suit-condition-minimum-battery

suit-condition-minimum-battery provides a mechanism to test a device's battery level before installing an update. This condition is for use in primary-cell applications, where the battery is only ever discharged. For batteries that are charged, suit-directive-wait is more appropriate, since it defines a "wait" until the battery level is sufficient to install the update. suit-condition-minimum-battery is specified in mWh. suit-condition-minimum-battery is OPTIONAL to implement.

7.11.6. suit-condition-update-authorised

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

7.11.7. suit-condition-version

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

- Greater

- Greater or Equal

- Equal

- Lesser or Equal

- Lesser

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

The following CDDL describes SUIT_Condition_Version_Argument

```
SUIT_Condition_Version_Argument = [
    suit-condition-version-comparison: SUIT_Condition_Version_Comparison_Types,
    suit-condition-version-comparison: SUIT_Condition_Version_Comparison_Value
]
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Gre
ater
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Gre
ater_Equal
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Equ
al
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Les
ser_Equal
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Les
ser
SUIT_Condition_Version_Comparison_Greater = 1
SUIT_Condition_Version_Comparison_Greater_Equal = 2
SUIT_Condition_Version_Comparison_Equal = 3
SUIT_Condition_Version_Comparison_Lesser_Equal = 4
SUIT_Condition_Version_Comparison_Lesser = 5

SUIT_Condition_Version_Comparison_Value = [+int]
```

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

- 1.2.3 = [1,2,3]

- 1.2-rc3 = [1,2,-1,3]

- 1.2-beta = [1,2,-2]

- 1.2-alpha = [1,2,-3]

- 1.2-alpha4 = [1,2,-3,4]

suit-condition-version is OPTIONAL to implement.

7.11.8. SUIT_Condition_Custom

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

7.11.9. Identifiers

Many conditions use identifiers to determine whether a manifest matches a given recipient or not. These identifiers are defined to be RFC 4122 [RFC4122] UUIDs. These UUIDs are explicitly NOT human-readable. They are for machine-based matching only.

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

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

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

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

1. hardware model/revision

2. OS

3. WiFi module model/revision

4. Application

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

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

7.11.9.1. Creating UUIDs:

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

The RECOMMENDED method to create a vendor ID is: Vendor ID = UUID5(DNS_PREFIX, vendor domain name)

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

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

- Model number

- Hardware revision

- Bootloader version (for immutable bootloaders)

7.11.10. SUIT_Condition CDDL

The following CDDL describes SUIT_Condition:

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

SUIT_Condition_Version_Argument = [
    suit-condition-version-comparison: SUIT_Condition_Version_Comparison_Types,
    suit-condition-version-comparison: SUIT_Condition_Version_Comparison_Value
]
SUIT_Condition_Version_Comparison_Types /= suit-condition-version-comparison-gre
ater
SUIT_Condition_Version_Comparison_Types /= suit-condition-version-comparison-gre
ater-equal
SUIT_Condition_Version_Comparison_Types /= suit-condition-version-comparison-equ
al
SUIT_Condition_Version_Comparison_Types /= suit-condition-version-comparison-les
ser-equal
SUIT_Condition_Version_Comparison_Types /= suit-condition-version-comparison-les
ser

SUIT_Condition_Version_Comparison_Value = [+int]
```

7.12. SUIT_Directive

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

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

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

7.12.1. suit-directive-set-component-index

Set Component Index defines the component to which successive directives and conditions will apply. The supplied argument MUST be either a boolean or an unsigned integer index into the concatenation of suit-components and suit-dependency-components. If the following directives apply to ALL components, then the boolean value "True" is used instead of an index. True does not apply to dependency components. If the following directives apply to NO components, then the boolean value "False" is used. When suit-directive-set-dependency-index is used, suit-directive-set-component-index = False is implied. When suit-directive-set-component-index is used, suit-directive-set-dependency-index = False is implied.

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

```
SUIT_Directive_Set_Component_Index_Argument = uint/bool
```

7.12.2. suit-directive-set-dependency-index

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

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

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

```
SUIT_Directive_Set_Manifest_Index_Argument = uint/bool
```

7.12.3. suit-directive-abort

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

7.12.4. suit-directive-run-sequence

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

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

The following CDDL describes the SUIT_Run_Sequence argument.

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

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

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

7.12.5. suit-directive-try-each

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

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

The following CDDL describes the SUIT_Try_Each argument.

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

7.12.6. suit-directive-process-dependency

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

If the current dependency is False, this directive has no effect. If the current dependency is True, then this directive applies to all dependencies. If the current section is "common," this directive MUST have no effect.

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

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

```
SUIT_Directive_Process_Dependency_Argument = nil
```

7.12.7. suit-directive-set-parameters

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

Available parameters are defined in Section 7.5.

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

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

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

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

7.12.8. suit-directive-override-parameters

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

Available parameters are defined in Section 7.5.

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

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

7.12.9. suit-directive-fetch

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

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

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

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

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

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

```
SUIT_Directive_Fetch_Argument = nil/bstr
```

7.12.10. suit-directive-copy

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

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

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

suit-directive-copy reads its source from SUIT_Parameter_Source_Component.

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

```
SUIT_Directive_Copy_Argument = nil
```

7.12.11. suit-directive-swap

suit-directive-swap instructs the manifest processor to move the source to the destination and the destination to the source simultaneously. Swap has nearly identical semantics to suit-directive-copy except that suit-directive-swap replaces the source with the current contents of the destination in an application-defined way. If SUIT_Parameter_Compression_Info or SUIT_Parameter_Encryption_Info are present, they must be handled in a symmetric way, so that the source is decompressed into the destination and the destination is compressed into the source. The source is decrypted into the destination and the destination is encrypted into the source. suit-directive-swap is OPTIONAL to implement.

7.12.12. suit-directive-run

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

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

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

```
SUIT_Directive_Run_Argument = nil/bstr
```

7.12.13. suit-directive-wait

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

1. Authorisation

2. External Power

3. Network availability

4. Other Device Firmware Version

5. Time

6. Time of Day

7. Day of Week

The following CDDL defines the encoding of these events.

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


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

7.12.14. SUIT_Directive CDDL

The following CDDL describes SUIT_Directive:

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

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

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


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

8. Dependency processing

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

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

Dependencies MAY be addressed in one of three ways:

1. Iterate by component

2. Iterate by manifest

3. Out-of-order

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

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

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

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

9. Access Control Lists

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

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

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

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

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

10. SUIT digest container

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

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

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

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

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

11. Creating conditional sequences

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

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


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

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

```
          },
          "fetch" : null
      ]
  }
```

12. Full CDDL

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

```
SUIT_Outer_Wrapper = {
    suit-authentication-wrapper => bstr .cbor SUIT_Authentication_Wrapper / nil,
    suit-manifest               => bstr .cbor SUIT_Manifest,
    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-authentication-wrapper = 1
suit-manifest = 2
suit-dependency-resolution = 7
suit-payload-fetch = 8
suit-install = 9
suit-text = 13
suit-coswid = 14

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


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

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

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

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

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

concise-software-identity = any
```

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

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

SUIT_Component_Identifier =  [* bstr]

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

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

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

suit-component-dependency-index = 2

SUIT_Command_Sequence = [ + { SUIT_Condition // SUIT_Directive // SUIT_Command_C
ustom} ]

SUIT_Command_Custom = (nint => bstr)

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

SUIT_Condition_Vendor_Identifier = 1
RFC4122_UUID = bstr .size 16

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

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

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

SUIT_Condition_Version_Comparison_Value = [+int]

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

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


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

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

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

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

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

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

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

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

SUIT_Unpack_Algorithm_Delta = 1
SUIT_Unpack_Algorithm_Hex = 2
```

```
SUIT_Unpack_Algorithm_Elf = 3

SUIT_Text_Map = {int => tstr}
```

13. Examples

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

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

13.1. Example 0:

Secure boot only.

The following JSON shows the intended behaviour of the manifest.

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

Converted into the SUIT manifest, this produces:

```
{
    / auth object / 1 : None
    / manifest / 3 : h'a40101020103583ca2024c818245466c6173684300340104'
                     h'582a8213a20b58200011223344556677889 9aabbccddeeff'
                     h'0123456789abcdeffedcba98765432100c1987d00c47860c'
                     h'0003f617f6' \
    {
        / structure-version / 1 : 1
        / sequence-number / 2 : 1
        / common / 3 : h'a2024c818245466c6173684300340104582a8213a20b58'
                       h'2000112233445566778899aabbccddeeff0123456789ab'
                       h'cdeffedcba98765432100c1987d0' \ {
            / components / 2 : h'818245466c61736843003401' \
            [
                [h'466c617368', h'003401'],
            ],
            / common / 4 : h'8213a20b58200011223344556677889 9aabbccddee'
                           h'ff0123456789abcdeffedcba98765432100c1987d0'
            \ [
                / set-vars / 19, {
                    / digest / 11 :h'00112233445566778899aabbccddeeff01'
                                    h'23456789abcdeffedcba9876543210',
                    / size / 12 : 34768
                },
            ],
        },
        / run-image / 12 : h'860c0003f617f6' \ [
            / set-component-index / 12, 0,
            / condition-image / 3, None,
            / run / 23, None,
        ],
    }
}
```

Total size of outer wrapper without COSE authentication object: 83

Outer:

a201f603584da40101020103583ca2024c818245466c6173684300340104582a8213a20b
58200011223344556677889 9aabbccddeeff0123456789abcdeffedcba98765432100c19
87d00c47860c0003f617f6

13.2. Example 1:

Simultaneous download and installation of payload.

The following JSON shows the intended behaviour of the manifest.

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


Converted into the SUIT manifest, this produces:

```
{
    / auth object / 1 : None
    / manifest / 3 : h'a40101020203583ca2024c818245466c6173684300340104'
                     h'582a8213a20b58200011223344556677889 9aabbccddeeff'
                     h'0123456789abcdeffedcba98765432100c1987d009582586'
                     h'0c0013a106781b687474703a2f2f6578616d706c652e636f'
                     h'6d2f66696c652e62696e15f6' \
    {
        / structure-version / 1 : 1
        / sequence-number / 2 : 2
        / common / 3 : h'a2024c818245466c6173684300340104582a8213a20b58'
                       h'200011223344556677889 9aabbccddeeff0123456789ab'
                       h'cdeffedcba98765432100c1987d0' \ {
            / components / 2 : h'818245466c61736843003401' \
            [
                [h'466c617368', h'003401'],
            ],
            / common / 4 : h'8213a20b58200011223344556677889 9aabbccddee'
                           h'ff0123456789abcdeffedcba98765432100c1987d0'
            \ [
                / set-vars / 19, {
                    / digest / 11 :h'0011223344556677889 9aabbccddeeff01'
                                    h'23456789abcdeffedcba9876543210',
                    / size / 12 : 34768
                },
            ],
        },
        / apply-image / 9 : h'860c0013a106781b687474703a2f2f6578616d70'
                            h'6c652e636f6d2f66696c652e62696e15f6' \ [
            / set-component-index / 12, 0,
            / set-vars / 19, {
                / uri / 6 : http://example.com/file.bin
            },
            / fetch / 21, None,
        ],
    }
}
```

Total size of outer wrapper without COSE authentication object: 114

Outer:

a201f603586ca40101020203583ca2024c818245466c6173684300340104582a8213a20b
5820001122334455667788 99aabbccddeeff0123456789abcdeffedcba98765432100c19
87d0095825860c0013a106781b687474703a2f2f6578616d706c652e636f6d2f66696c65
2e62696e15f6

13.3. Example 2:

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

The following JSON shows the intended behaviour of the manifest.


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


Converted into the SUIT manifest, this produces:

```
{
    / auth object / 1 : None
    / manifest / 3 : h'a5010102030358 64a2024c818245466c6173684300340104'
                     h'58528613a40350fa6b4a53d5ad5fdfbe9de663e4d41ffe04'
                     h'501492af1425695e48bf429b2d51f2ab450b582000112233'
                     h'445566778899aabbccddeeff0123456789abcdeffedcba98'
                     h'765432100c1987d001f602f6095825860c0013a106781b68'
                     h'7474703a2f2f6578616d706c652e636f6d2f66696c652e62'
                     h'696e15f60c47860c0003f617f6' \
    {
        / structure-version / 1 : 1
        / sequence-number / 2 : 3
        / common / 3 : h'a2024c818245466c6173684300340104 58528613a40350'
                       h'fa6b4a53d5ad5fdfbe9de663e4d41ffe04501492af1425'
                       h'695e48bf429b2d51f2ab450b5820001122334455667788'
                       h'99aabbccddeeff0123456789abcdeffedcba9876543210'
                       h'0c1987d001f602f6' \ {
            / components / 2 : h'818245466c61736843003401' \
            [
                [h'466c617368', h'003401'],
            ],
            / common / 4 : h'8613a40350fa6b4a53d5ad5fdfbe9de663e4d41ffe'
                           h'04501492af1425695e48bf429b2d51f2ab450b5820'
                           h'00112233445566778899aabbccddeeff0123456789'
                           h'abcdeffedcba98765432100c1987d001f602f6' \ [
                / set-vars / 19, {
                    / vendor-id / 3 : h'fa6b4a53d5ad5fdfbe9de663e4d41f'
                                      h'fe'
                    / class-id / 4 : h'1492af1425695e48bf429b2d51f2ab45'
                    / digest / 11 :h'00112233445566778899aabbccddeeff01'
                                    h'23456789abcdeffedcba9876543210',
                    / size / 12 : 34768
                },
                / condition-vendor-id / 1, None,
                / condition-class-id / 2, None,
            ],
        },
        / apply-image / 9 : h'860c0013a106781b687474703a2f2f6578616d70'
                            h'6c652e636f6d2f66696c652e62696e15f6' \ [
            / set-component-index / 12, 0,
            / set-vars / 19, {
                / uri / 6 : http://example.com/file.bin
            },
            / fetch / 21, None,
        ],
        / run-image / 12 : h'860c0003f617f6' \ [
            / set-component-index / 12, 0,
            / condition-image / 3, None,
```

```
            / run / 23, None,
        ],
    }
}
```

Total size of outer wrapper without COSE authentication object: 163

Outer:

a201f603589da501010203035864a2024c818245466c6173684300340104585286l3a403
50fa6b4a53d5ad5fdfbe9de663e4d41ffe04501492af1425695e48bf429b2d51f2ab450b
5820001122334455667788999aabbccddeeff0123456789abcdeffedcba98765432100c19
87d001f602f6095825860c0013a106781b687474703a2f2f6578616d706c652e636f6d2f
66696c652e62696e15f60c47860c0003f617f6

13.4. Example 3:

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

The following JSON shows the intended behaviour of the manifest.


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

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

Converted into the SUIT manifest, this produces:

```
{
    / auth object / 1 : None
    / manifest / 3 : h'a50101020403589ba20254828245466c6173684300340182'
                     h'4352414d4200040458818e13a20350fa6b4a53d5ad5fdfbe'
                     h'9de663e4d41ffe04501492af1425695e48bf429b2d51f2ab'
                     h'450c0013a20b5820001122334455667788 99aabbccddeeff'
                     h'0123456789abcdeffedcba98765432100c1987d00c0113a2'
```

```
                     h'0b5820001122334455667788 99aabbccddeeff0123456789'
```

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

Total size of outer wrapper without COSE authentication object: 228

Outer:

a201f60358dea50101020403589ba20254828245466c6173684300340182435241 4d4200
040458818e13a20350fa6b4a53d5ad5fdfbe9de663e4d41ffe04501492af1425695e48bf
429b2d51f2ab450c0013a20b58200011223344556677889 9aabbccddeeff0123456789ab
cdeffedcba98765432100c1987d00c0113a20b58200011223344556677889 9aabbccddee
ff0123456789abcdeffedcba98765432100c1987d001f602f6095825860c0013a106781b
687474703a2f2f6578616d706c652e636f6d2f66696c652e62696e15f60c518e0c0003f6
0c0113a10a0015f603f617f6

13.5. Example 4:

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

The following JSON shows the intended behaviour of the manifest.

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

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

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

Converted into the SUIT manifest, this produces:

```
{
    / auth object / 1 : None
    / manifest / 3 : h'a60101020503589ba20254828245466c6173684300340182'
                     h'4352414d4200040458818e13a20350fa6b4a53d5ad5fdfbe'
                     h'9de663e4d41ffe04501492af1425695e48bf429b2d51f2ab'
                     h'450c0013a20b5820001122334455667788899aabbccddeeff'
                     h'0123456789abcdeffedcba98765432100c1987d00c0113a2'
                     h'0b58200123456789abcdeffedcba98765432100011223344'
                     h'5566778899aabbccddeeff0c1987d001f602f6095825860c'
                     h'0013a106781b687474703a2f2f6578616d706c652e636f6d'
                     h'2f66696c652e62696e15f60b508a0c0003f60c0113a20841'
                     h'f60a0016f60c458403f617f6' \
    {
        / structure-version / 1 : 1
        / sequence-number / 2 : 5
        / common / 3 : h'a20254828245466c617368430034018243524144d420004'
                       h'0458818e13a20350fa6b4a53d5ad5fdfbe9de663e4d41f'
                       h'fe04501492af1425695e48bf429b2d51f2ab450c0013a2'
                       h'0b58200011223344556677889 9aabbccddeeff01234567'
                       h'89abcdeffedcba98765432100c1987d00c0113a20b5820'
                       h'0123456789abcdeffedcba987654321000112233445566'
                       h'778899aabbccddeeff0c1987d001f602f6' \ {
            / components / 2 : h'828245466c617368430034018243524144d4200'
                               h'04' \
            [
                [h'466c617368', h'003401'],
                [h'52414d', h'0004'],
            ],
            / common / 4 : h'8e13a20350fa6b4a53d5ad5fdfbe9de663e4d41ffe'
                           h'04501492af1425695e48bf429b2d51f2ab450c0013'
```

```
                          h'a20b58200011223344556677889 9aabbccddeeff01'
                          h'23456789abcdeffedcba98765432100c1987d00c01'
                          h'13a20b58200123456789abcdeffedcba9876543210'
                          h'00112233445566778899aabbccddeeff0c1987d001'
                          h'f602f6' \ [
                / set-vars / 19, {
                    / vendor-id / 3 : h'fa6b4a53d5ad5fdfbe9de663e4d41f'
                                      h'fe'
                    / class-id / 4 : h'1492af1425695e48bf429b2d51f2ab45'
                },
                / set-component-index / 12, 0,
                / set-vars / 19, {
                    / digest / 11 :h'00112233445566778899aabbccddeeff01'
                                   h'23456789abcdeffedcba9876543210',
                    / size / 12 : 34768
                },
                / set-component-index / 12, 1,
                / set-vars / 19, {
                    / digest / 11 :h'0123456789abcdeffedcba987654321000'
                                   h'112233445566778899aabbccddeeff',
                    / size / 12 : 34768
                },
                / condition-vendor-id / 1, None,
                / condition-class-id / 2, None,
            ],
        },
        / apply-image / 9 : h'860c0013a106781b687474703a2f2f6578616d70'
                            h'6c652e636f6d2f66696c652e62696e15f6' \ [
            / set-component-index / 12, 0,
            / set-vars / 19, {
                / uri / 6 : http://example.com/file.bin
            },
            / fetch / 21, None,
        ],
        / load-image / 11 : h'8a0c0003f60c0113a20841f60a0016f6' \ [
            / set-component-index / 12, 0,
            / condition-image / 3, None,
            / set-component-index / 12, 1,
            / set-vars / 19, {
                / unknown / 8 : h'f6'
                / source-component / 10 : 0
            },
            / copy / 22, None,
        ],
        / run-image / 12 : h'8403f617f6' \ [
            / condition-image / 3, None,
            / run / 23, None,
        ],
```

```
    }
}
```

Total size of outer wrapper without COSE authentication object: 234

Outer:

a201f60358e4a60101020503589ba20254828245466c61736843003401824352414d4200
040458818e13a20350fa6b4a53d5ad5fdfbe9de663e4d41ffe04501492af1425695e48bf
429b2d51f2ab450c0013a20b582000112233445566778899aabbccddeeff0123456789ab
cdeffedcba98765432100c1987d00c0113a20b58200123456789abcdeffedcba98765432
1000112233445566778899aabbccddeeff0c1987d001f602f6095825860c0013a106781b
687474703a2f2f6578616d706c652e636f6d2f66696c652e62696e15f60b508a0c0003f6
0c0113a20841f60a0016f60c458403f617f6

13.6. Example 5:

Compatibility test, download, installation, and secure boot.

The following JSON shows the intended behaviour of the manifest.


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

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

Converted into the SUIT manifest, this produces:

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

```
                     h'8245466c6173684200040458818e13a20350fa6b4a53d5ad'
                     h'5fdfbe9de663e4d41ffe04501492af1425695e48bf429b2d'
                     h'51f2ab450c0013a20b5820001122334455667788 99aabbcc'
                     h'ddeeff0123456789abcdeffedcba98765432100c1987d00c'
                     h'0113a20b58200123456789abcdeffedcba98765432100011'
                     h'2233445566778899aabbccddeeff0c1987d001f602f60958'
                     h'25860c0013a106581b687474703a2f2f6578616d706c652e'
                     h'636f6d2f66696c652e62696e15f60b528e0c011819f60c00'
                     h'03f60c0113a10a0015f60c47860c0103f617f6' \
    {
        / structure-version / 1 : 1
        / sequence-number / 2 : 6
        / common / 3 : h'a202578282467b1b4595ab2143003401824546 6c617368'
                       h'4200040458818e13a20350fa6b4a53d5ad5fdfbe9de663'
                       h'e4d41ffe04501492af1425695e48bf429b2d51f2ab450c'
                       h'0013a20b58200011223344556677889 9aabbccddeeff01'
                       h'23456789abcdeffedcba98765432100c1987d00c0113a2'
                       h'0b58200123456789abcdeffedcba98765432100011 2233'
                       h'445566778899aabbccddeeff0c1987d001f602f6' \ {
            / components / 2 : h'8282467b1b4595ab21430034018245466c6173'
                                 h'68420004' \
            [
                [h'7b1b4595ab21', h'003401'],
                [h'466c617368', h'0004'],
            ],
            / common / 4 : h'8e13a20350fa6b4a53d5ad5fdfbe9de663e4d41ffe'
                           h'04501492af1425695e48bf429b2d51f2ab450c0013'
                           h'a20b582000112233445566778899aabbccddeeff01'
                           h'23456789abcdeffedcba98765432100c1987d00c01'
                           h'13a20b58200123456789abcdeffedcba9876543210'
                           h'00112233445566778899aabbccddeeff0c1987d001'
                           h'f602f6' \ [
                / set-vars / 19, {
                    / vendor-id / 3 : h'fa6b4a53d5ad5fdfbe9de663e4d41f'
                                      h'fe'
                    / class-id / 4 : h'1492af1425695e48bf429b2d51f2ab45'
                },
                / set-component-index / 12, 0,
                / set-vars / 19, {
                    / digest / 11 :h'00112233445566778899aabbccddeeff01'
                                    h'23456789abcdeffedcba9876543210',
                    / size / 12 : 34768
                },
                / set-component-index / 12, 1,
                / set-vars / 19, {
                    / digest / 11 :h'0123456789abcdeffedcba987654321000'
                                    h'112233445566778899aabbccddeeff',
                    / size / 12 : 34768
```

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

Total size of outer wrapper without COSE authentication object: 241

Outer:

a201f60358eba60101020603589ea202578282467b1b4595ab21430034018245466c6173
684200040458818e13a20350fa6b4a53d5ad5fdfbe9de663e4d41ffe04501492af142569
5e48bf429b2d51f2ab450c0013a20b582000112233445566778899aabbccddeeff012345
6789abcdeffedcba98765432100c1987d00c0113a20b58200123456789abcdeffedcba98
7654321000112233445566778899aabbccddeeff0c1987d001f602f6095825860c0013a1
06581b687474703a2f2f6578616d706c652e636f6d2f66696c652e62696e15f60b528e0c
011819f60c0003f60c0113a10a0015f60c47860c0103f617f6

13.7. Example 6:

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

The following JSON shows the intended behaviour of the manifest.


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

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

Converted into the SUIT manifest, this produces:

```
{
    / auth object / 1 : None
    / manifest / 3 : h'a5010102070358a0a20257828245466c6173684300340182'
                     h'45466c617368430004020458838e13a20350fa6b4a53d5ad'
                     h'5fdfbe9de663e4d41ffe04501492af1425695e48bf429b2d'
                     h'51f2ab450c0013a20b58200011223344556677889 9aabbcc'
                     h'ddeeff0123456789abcdeffedcba98765432100c1987d00c'
                     h'0113a20b58200123456789abcdeffedcba98765432100011'
                     h'2233445566778899aabbccddeeff0c1a00012c2201f602f6'
                     h'09584b8c0c0013a106781c687474703a2f2f6578616d706c'
                     h'652e636f6d2f66696c65312e62696e0c0113a106781c6874'
                     h'74703a2f2f6578616d706c652e636f6d2f66696c65322e62'
                     h'696e0cf515f60c49880cf503f60c0017f6' \
    {
        / structure-version / 1 : 1
        / sequence-number / 2 : 7
        / common / 3 : h'a20257828245466c6173684300340182 45466c61736843'
                       h'0004020458838e13a20350fa6b4a53d5ad5fdfbe9de663'
                       h'e4d41ffe04501492af1425695e48bf429b2d51f2ab450c'
                       h'0013a20b5820001122334455667788 99aabbccddeeff01'
                       h'23456789abcdeffedcba98765432100c1987d00c0113a2'
                       h'0b58200123456789abcdeffedcba98765432100 0112233'
```

```
                  h'445566778899aabbccddeeff0c1a00012c2201f602f6'
        \ {
            / components / 2 : h'828245466c6173684300340182454 66c617368'
                                h'43000402' \
            [
                [h'466c617368', h'003401'],
                [h'466c617368', h'000402'],
            ],
            / common / 4 : h'8e13a20350fa6b4a53d5ad5fdfbe9de663e4d41ffe'
                           h'04501492af1425695e48bf429b2d51f2ab450c0013'
                           h'a20b58200011223344556677889 9aabbccddeeff01'
                           h'23456789abcdeffedcba98765432100c1987d00c01'
                           h'13a20b58200123456789abcdeffedcba9876543210'
                           h'00112233445566778899aabbccddeeff0c1a00012c'
                           h'2201f602f6' \ [
                / set-vars / 19, {
                    / vendor-id / 3 : h'fa6b4a53d5ad5fdfbe9de663e4d41f'
                                      h'fe'
                    / class-id / 4 : h'1492af1425695e48bf429b2d51f2ab45'
                },
                / set-component-index / 12, 0,
                / set-vars / 19, {
                    / digest / 11 :h'00112233445566778899aabbccddeeff01'
                                   h'23456789abcdeffedcba9876543210',
                    / size / 12 : 34768
                },
                / set-component-index / 12, 1,
                / set-vars / 19, {
                    / digest / 11 :h'0123456789abcdeffedcba987654321000'
                                   h'112233445566778899aabbccddeeff',
                    / size / 12 : 76834
                },
                / condition-vendor-id / 1, None,
                / condition-class-id / 2, None,
            ],
        },
        / apply-image / 9 : h'8c0c0013a106781c687474703a2f2f6578616d70'
                            h'6c652e636f6d2f66696c65312e62696e0c0113a1'
                            h'06781c687474703a2f2f6578616d706c652e636f'
                            h'6d2f66696c65322e62696e0cf515f6' \ [
            / set-component-index / 12, 0,
            / set-vars / 19, {
                / uri / 6 : http://example.com/file1.bin
            },
            / set-component-index / 12, 1,
            / set-vars / 19, {
                / uri / 6 : http://example.com/file2.bin
            },
```

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

Total size of outer wrapper without COSE authentication object: 264

Outer:

a201f603590101a5010102070358a0a20257828245466c6173684300340182454 66c6173

14. IANA Considerations

Several registries will be required for:

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

15. Security Considerations

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

16. Mailing List Information

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

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

17. Acknowledgements

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

- Milosch Meriac

- Geraint Luff

- Dan Ros

- John-Paul Stanford

- Hugo Vincent

- Carsten Bormann

- Oeyvind Roenningstad

- Frank Audun Kvamtroe

- Krzysztof Chruściński

- Andrzej Puzdrowski

- Michael Richardson

- David Brown

- Emmanuel Baccelli

18. References

18.1. Normative References

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

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

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

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

18.2. Informative References

[Architecture] Moran, B., "A Firmware Update Architecture for Internet of Things Devices", January 2019, <https://tools.ietf.org/html/draft-ietf-suit-architecture-02>.

[Information] Moran, B., "Firmware Updates for Internet of Things Devices - An Information Model for Manifests", January 2019, <https://tools.ietf.org/html/draft-ietf-suit-information-model-02>.

[RFC6920] Farrell, S., Kutscher, D., Dannewitz, C., Ohlman, B., Keranen, A., and P. Hallam-Baker, "Naming Things with Hashes", RFC 6920, DOI 10.17487/RFC6920, April 2013, <https://www.rfc-editor.org/info/rfc6920>.

18.3. URIs

[1] mailto:suit@ietf.org

[2] https://www1.ietf.org/mailman/listinfo/suit

[3] https://www.ietf.org/mail-archive/web/suit/current/index.html

Authors' Addresses

Brendan Moran
Arm Limited

EMail: Brendan.Moran@arm.com

Hannes Tschofenig
Arm Limited

EMail: hannes.tschofenig@arm.com

Henk Birkholz
Fraunhofer SIT

EMail: henk.birkholz@sit.fraunhofer.de

Network Working Group S. Nandakumar
Internet-Draft C. Jennings
Intended status: Standards Track S. Cooley
Expires: May 3, 2018 Cisco
October 30, 2017

Solution Requirements - Secure Firmware Upgrade (SecFU)
draft-nandakumar-suit-secfu-requirements-00

Abstract

The IETF SUIT effort has been forming to define a secure firmware upgrade solution for Internet of Things (IOT). Recent vulnerabilities and the need to upgrade firmware on the IoT devices for security updates in a standardized, secure, and automated fashion has been the driving force behind this work.

This specification is a requirements document to aid in developing a solution for Secure Firmware upgrade of the IoT devices.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This draft outlines a set of requirements around firmware download for IoT devices. A sketch of a proposed solution can be found in .

2. Solution Requirements

Informally, a secure firmware upgrade solution might need to address following components:

- o Secure firmware description container format, in the form of Manifest
- o Locating a server to download the firmware from
- o Downloading the manifest and the firmware image(s)
- o Cryptographic validation of the manifest and signed code images
- o Complete the installation

Given above tasks, this specification breaks down the secure firmware upgrade solution into following requirements:

1. Solution must allow devices that delete the old firmware before installing the new firmware. Thus implying a solution that can easily be implementable on a minimal boot-loader

2. Solution must enable devices that have enough memory to have the new firmware image of the firmware simultaneously loaded with the existing image.

3. The manifest format should be self describing.

4. Allow a given device to decide which manifest format is appropriate for it choosing from JSON, CBOR, or perhaps ASN.1 if there is a a device vendor that plans to use this

5. Manifest must allow metadata about the firmware sourced by a single manufacturer

6. Optionally, the solution may allow the manifest to describe metadata about firmwares from different providers

7. The solution should enable firmware that is delivered as a single image

8. Optionally, the solution may enable firmware to be split into multiple images.

9. The charter should recommend a solution agnostic to the format of the firmware image and inter dependencies. Dependency management is complicated and is by nature proprietary and should not be in the initial scope.

10. The proposed solution must provide mechanism to discover where to download the firmware where that mechanism includes the ability for a local cache.

11. The proposed solution should allow flexibility to choose the underlying transport protocol as defined by the deployment scenarios. The WG should define a MTI set of protocols that firmware servers need to implement and clients can choose which one to use

12. The proposed solution must require a device to validate signatures on the manifest and firmware image(s)

13. Optionally, the solution might want to support encrypted manifest and firmware

14. The proposed solution should enable crypto agility and prevent roll-back attacks.

15. Solution should allow for secure transition between the generations of the keying material

16. Charter should not invent new crypto or transports and use existing techniques

3. IANA Consideration

Not Applicable

4. Security Considerations

Not Applicable

5. Acknowledgements

Thanks IOTSU workshop.

Authors' Addresses

Suhas Nandakumar
Cisco

Email: snandaku@cisco.com

Cullen Jennings
Cisco

Email: fluffy@iii.ca

Shaun Cooley
Cisco

Email: scooley@cisco.com

Network Working Group S. Nandakumar
Internet-Draft C. Jennings
Intended status: Standards Track S. Cooley
Expires: May 3, 2018 Cisco
October 30, 2017

Solution Architecture - Secure Firmware Upgrade (SecFU)
draft-nandakumar-suit-secfu-solution-arch-00

Abstract

This specification defines a solution architecture for performing secure firmware upgrade for Internet of Things (IoT). The ulterior motive is to have a framework that is simple, secure, and that uses most common formats and standards in the industry and that works over Internet.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Internet of Things (IOT) represents a plethora of devices that come in varying flavors of constrained sizes, computing power, and operating considerations. These devices usually need minimal or no management for their operation.

Vulnerabilities within IOT devices have raised serious concerns. There needs to be a way to install or update the firmware on these devices in an automated and secure fashion. A common challenge with the existing firmware update mechanism is they do not work in an automated manner in many environments where IoT devices are deployed. Hence, there is a need to define a firmware update solution that is light weight, secure, can operate in variety of deployment environments, and is built on well established standards.

2. Terminology

In this document, the key words "MUST", "MUST NOT", "SHOULD", "SHOULD NOT", "MAY", and "OPTIONAL" are to be interpreted as described in RFC 2119 [RFC2119] and indicate requirement levels for compliant implementations.

3. Device Considerations

This draft targets devices that have a boot loader that run in less than 100K bytes of flash and less than 32K bytes of RAM.

There are certain types of devices that delete the firmware image, except for the boot-loader, before proceeding with the upgrade. Alternatively, many devices have sufficient storage to completely download a new firmware image before updating. This solution should be naturally applicable to both.

4. Solution Overview

draft-nandakumar-suit-secfu-requirements captures various requirements that drives the solution defined in this specification.

Below is a high-level solution flow for a successful firmware update on a IoT device.

```
                    Successful Firmware Update Flow

                          on-ready to update
                               firmware
                                  |
                                  |
                       Can access firmware sever
                         in the local domain ?
                                  |
                                  |
                          ------------------
                     Yes  |                | No
                          |                |
                 Download signed       Download signed
                 manifest from          manifest from
                 local server          manufacturer's
                 well-known URL        pre-configured URL
                          |                  |
                          |                  |
                         Validate manifest via
                         pre-installed public key
                                   |
                                   |
                       Download the firmware image
                       from the location in manifest
                                   |
                                   |
                          verify commit hashes
                          on the firmware image
                                   |
                                   |
                          complete installation
```

5. Solution Components

Following several sub-sections define various components that makes up the proposed solution architecture

5.1. Manifest

A firmware manifest serves as information representation for metadata about the firmware. A manifest file identifies information about the actual firmware image, its location, applicable device(s), and so on. It is cryptographically signed by the provider (usually the manufacturer) of the firmware.

Minimal Manifest in JSON format


```
{
  "manifestVersion" : "1.0",
  "timestamp": "2017-12-10T15:12:15Z",
  "manufacturer": "manufacturer.com",
  "model": "c7960",
  "firmwareVersion": "10.4.12",
  "firmwareLocation": "well-known location",
  "firmwareCryptoInfo": {
    {
      "commitHash": [
        {
          "digestAlgo": "sha256",
          "hash": "..................."
        },
        {
          "digestAlgo": "sha512",
          "hash": "..................."
        }
      ]
    }
  }
  "key-info": <most recent public key info>
}
```


Above shows example of a minimally defined manifest that identifies the mandatory attributes as explained below

- o manifestVersion: Version of the manifest
- o timestamp: Time when the manifest was created.
- o manufacturer: An identifier of the manufacturer providing the firmware image, represented as String
- o model: Device Model, a String
- o firmwareVersion: Firmware Version in the format "major.minor.revision"
- o firmwareLocation: Location of the firmware images. This can be an absolute URI or a relative URI that is relative to where the manifest was downloaded from.
- o firmwareCryptoInfo: Commit Hash information

5.2. Manifest Format

JSON representation is recommended as the default format for describing the manifest. Optionally, formats such as CBOR (see example section) can be used for the same. If more than one format is used, the IoT device can pick one based on its implementation. The firmware download protocol identifies the right format supported by the IoT device.

5.3. Manifest Security

The Manifest file MUST be cryptographically signed by the private key of the manufacturer or the provider of the firmware. This is to ensure source authenticity and to protect integrity of the manifest and the firmware itself.

JWS is the format recommended to store the signed manifest.

```
signed_manifest := JWS(manifest.json)
```

If CBOR is used for describing the manifest, COSE is recommended for signing.

Optionally, the proposed solution also recommends hash based signatures (hash-sigs) to sign the manifest.

```
signed_manifest := hash-sigs(manifest.json, private-key)
```

5.4. Manifest Optional Extensions

There may be scenarios where the minimalistic manifest defined above may not capture all the requirements for a given deployment setting. In those circumstances, the manifest can be optionally extended to meet the requirements in a extensional specifications.

5.5. Firmware Server Discovery

When it is time for an IoT device to perform a firmware upgrade, the device performs couple of steps to decide the location to download the needed firmware. A device might need to download the new firmware when it is either booting for the first time after deployment or there is a need to upgrade to a newer firmware.

The server discovery procedure starts with the boot-loader attempting to access a server that is local to the domain in which the device operates. The URL to look for a local server is automatically generated using the DHCP domain name.

For example, if the domain name was example.com, and the device was a Cisco 7960, the HTTP URL might be

http://_firmware.example.com/.wellknown/firmware/cisco.com/c7960/manifest.json

In situations where the IoT device cannot access the Internet (factory/enterprise settings, for example), the aforementioned approach might be the only way for the device to perform any kind of firmware or security updates.

However, if the local server cannot be reached or not deployed (say home environments), the device proceeds to download the manifest and firmware from the firmware server URL pre-configured in the boot-loader by the manufacturer of the device. For example

http://_firmware.cisco.com/.wellknown/firmware/cisco.com/c7960/manifest.json

If either of the procedures doesn’t work, the IoT device is either unusable or might end up running an old version of the firmware.

5.6. Firmware Download protocol

One can envision two possibilities while downloading the firmware:

- Scenarios where the IoT device downloads firmware directly. This is done in order to minimize number of connections. In this scenario, the firmware image must have a digital signature included within the downloaded firmware. The exact placement of this digital signature (prepended, appended, etc) is up to the device manufacturer, but it MUST provided source and integrity guarantees on the entirety of the firmware image and must be verified by the device prior to upgrade.

- Scenarios where a manifest is retrieved and followed by downloading the actual firmware image.

5.6.1. Validation Procedures

The downloaded manifest and firmware is validated before being used:

- Manifest file signature is validated for source and integrity verification. If encrypted, the manifest is decrypted before proceeding with the firmware download.

- On successful validation of the manifest, the device verifies the commit hashes for component(s) of the firmware downloaded against the ones provided in the "firmwareCryptoInfo" section of the manifest.

5.6.2. Manifest Download

Firmware download protocol enables choosing the approach appropriate to the IoT device for downloading the manifest file.

For example, on performing the "Firmware Server Discovery", if a local server is chosen, the device forms a query URL by constructing an endpoint at ".well-known/manifest/<manufacturer>/<model-no>/manifest.json"

Then a HTTP GET request is sent to that URL. For example

```
http://_firmware.example.com/.wellknown/manifest/cisco.com/c7960/manifest.json
```

The response would be a JSON result of the manifest file. Similarly, the end-point supporting CBOR parsing can request for the CBOR version of the mannifest.

5.6.3. Firmware Download

Once the manifest is downloaded and validated, the device proceeds to download the firmware image from the location identified in the firmware manifest. There might be situations where a firmware image is split into multiple files to imply a functional division of the components. This type of firmware can be used by devices that are memory constrained and thus loading the complete image might not be possible. The manifest file may contain the information to indicate the same.

Above example shows use of HTTP as the communication protocol to talk to the firmware server. If the end-point is capable of doing COAP or other protocols, a similar process as above can be applied to retrieve the manifest and the firmware from a well-known place on the local server.

6. IANA Consideration

TODO

7. Security Considerations

TODO - Talk about roaming IoT Device

8. Acknowledgements

9. 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>.

Authors' Addresses

Suhas Nandakumar
Cisco

Email: snandaku@cisco.com

Cullen Jennings
Cisco

Email: fluffy@iii.ca

Shaun Cooley
Cisco

Email: scooley@cisco.com