A Firmware Update Architecture for Internet of Things Devices
draft-ietf-suit-architecture-05

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.

Status of This Memo

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

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

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

This Internet-Draft will expire on October 11, 2019.
Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

Table of Contents

1. Introduction ................................. 3
2. Conventions and Terminology .................. 3
3. Requirements ................................ 6
   3.1. Agnostic to how firmware images are distributed ... 7
   3.2. Friendly to broadcast delivery ................. 7
   3.3. Use state-of-the-art security mechanisms .......... 7
   3.4. Rollback attacks must be prevented .............. 8
   3.5. High reliability ................................ 8
   3.6. Operate with a small bootloader ................. 8
   3.7. Small Parsers ................................ 9
   3.8. Minimal impact on existing firmware formats .... 9
   3.9. Robust permissions ............................ 9
   3.10. Operating modes ............................ 9
4. Claims ....................................... 11
5. Communication Architecture .................... 12
6. Manifest ..................................... 16
7. Device Firmware Update Examples ............... 17
   7.1. Single CPU SoC ............................. 17
   7.2. Single CPU with Secure - Normal Mode Partitioning 17

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

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

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

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

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

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.

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

- Microcontroller (MCU for microcontroller unit): An MCU is a compact integrated circuit designed for use in embedded systems. A typical microcontroller includes a processor, memory (RAM and flash), input/output (I/O) ports and other features connected via some bus on a single chip. The term 'system on chip (SoC)' is often used for these types of devices.

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

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

The following entities are used:

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

- Firmware Consumer: The firmware consumer is the recipient of the firmware image and the manifest.

- Device: A device refers to the entire IoT product, which consists of one or many MCUs, sensors and/or actuators. Many IoT devices sold today contain multiple MCUs and therefore a single device may need to obtain more than one firmware image and manifest to successfully perform an update. The terms device and firmware consumer are used interchangeably since the firmware consumer is one software component running on an MCU on the device.

- Status Tracker: The status tracker offers device management functionality to monitor the firmware update process. A status tracker may, for example, want to know what state of the firmware update cycle the device is currently in.

- Firmware Server: The firmware server stores firmware images and manifests and distributes them to IoT devices. Some deployments may require a store-and-forward concept, which requires storing the firmware images/manifests on more than one entity before they reach the device.

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

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

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

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

- "A trust anchor represents an authoritative entity via a public key and associated data. The public key is used to verify digital signatures, and the associated data is used to constrain the types of information for which the trust anchor is authoritative."

- "A trust anchor store is a set of one or more trust anchors stored in a device. A device may have more than one trust anchor store, each of which may be used by one or more applications." A trust anchor store must resist modification against unauthorized insertion, deletion, and modification.

3. Requirements

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

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

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

3.2. Friendly to broadcast delivery

This architecture does not specify any specific broadcast protocol. However, given that broadcast may be desirable for some networks, updates must cause the least disruption possible both in metadata and payload transmission.

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. Use 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 migration to post-quantum secure signatures would require significant effort, therefore, mandatory-to-implement support for post-quantum secure signatures is a goal.

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 [RFC7925]. 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, for example AES Key Wrap [RFC5649], in order to maintain friendliness to Content Distribution Networks, bulk storage, and broadcast protocols.
3.4. Rollback attacks must be prevented

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

3.5. High reliability

A power failure at any time must not cause a failure of the device. A failure to validate any part of an update must not cause a failure of the device. One way to achieve this functionality is to provide a minimum of two storage locations for firmware and one bootable location for firmware. An alternative approach is to use a 2nd stage bootloader with build-in full featured firmware update functionality such that it is possible to return to the update process after power down.

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

3.6. Operate with a small bootloader

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 that are required to validate at least one signature or MAC with minimal exposure.

3.8. Minimal impact on existing firmware formats

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

3.9. Robust permissions

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

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

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

3.10. Operating modes

There are three broad classifications of update operating modes.

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

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

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

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

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

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

- Notification
- Pre-authorisation
- Dependency resolution
- Download
- Installation

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

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

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

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

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

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

Each of these steps may require different permissions.

4. Claims

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

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

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

Figure 1 shows the communication architecture where a firmware image is created by an author, and uploaded to a firmware server. The firmware image/manifest is distributed to the device either in a push or pull manner using the firmware consumer residing on the device. The device operator keeps track of the process using the status tracker. This allows the device operator to know and control what devices have received an update and which of them are still pending an update.
End-to-end security mechanisms are used to protect the firmware image and the manifest although Figure 2 does not show the manifest itself since it may be distributed independently.
Figure 2: End-to-End Security.

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

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

- To accept an update, a device needs to verify the signature covering the manifest. There may be one or multiple manifests that need to be validated, potentially signed by different parties. The device needs to be in possession of the trust anchors to verify those signatures. Installing trust anchors to devices via the Trust Provisioning Authority happens in an out-of-band fashion prior to the firmware update process.

- Not all entities creating and signing manifests have the same permissions. A device needs to determine whether the requested action is indeed covered by the permission of the party that signed the manifest. Informing the device about the permissions of the different parties also happens in an out-of-band fashion and is also a duty of the Trust Provisioning Authority.

- For confidentiality protection of firmware images the author needs to be in possession of the certificate/public key or a pre-shared key of a device. The use of confidentiality protection of firmware images is deployment specific.

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

There is an option for embedding a firmware image into a manifest. This is a useful approach for deployments where devices are not connected to the Internet and cannot contact a dedicated firmware server for the firmware download. It is also applicable when the
firmware update happens via a USB stick or via Bluetooth Smart. Figure 3 shows this delivery mode graphically.

```
+---------+ <----------+<----------+
| Device  | Manifest  | Firmware |
|---------+----------+----------|
```

Figure 3: Manifest with attached firmware.

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

```
+---------+ <----------+<----------+
| Device  | Manifest  | Firmware |
|---------+----------+----------|
```

Figure 4: Independent retrieval of the firmware image.

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

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

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

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

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

The manifest information model is described in [I-D.ietf-suit-information-model].
7. Device Firmware Update Examples

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

7.1. Single CPU SoC

The simplest, and currently most common, architecture consists of a single MCU along with its own peripherals. These SoCs generally contain some amount of flash memory for code and fixed data, as well as RAM for working storage. These systems either have a single firmware image, or an immutable bootloader that runs a single image. A notable characteristic of these SoCs is that the primary code is generally execute in place (XIP). Combined with the non-relocatable nature of the code, firmware updates need to be done in place.

7.2. Single CPU with Secure - Normal Mode Partitioning

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

7.3. Dual CPU, shared memory

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

7.4. Dual CPU, other bus

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

8. Bootloader

More devices today than ever before are being connected to the Internet, which drives the need for firmware updates to be provided over the Internet rather than through traditional interfaces, such as USB or RS232. Updating a device over the Internet requires the device to fetch not only the firmware image but also the manifest. Hence, the following building blocks are necessary for a firmware update solution:

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

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

Once manifests have been processed and firmware images successfully downloaded and verified the device needs to hand control over to the bootloader. In most cases this requires the MCU to restart. Once the MCU has initiated a restart, the bootloader takes over control and determines whether the newly downloaded firmware image should be executed.

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

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

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

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

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

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

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

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

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

9. Example

The following example message flow illustrates a possible interaction for distributing a firmware image to a device starting with an author uploading the new firmware to firmware server and creating a manifest. The firmware and manifest are stored on the same firmware server.
Figure 5: Example Flow for a Firmware Update.

10. IANA Considerations

This document does not require any actions by IANA.
11. Security Considerations

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

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

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

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

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

- energy efficiency and battery lifetime considerations.

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

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

12. 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]
13. Acknowledgements

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

- Geraint Luff
- Amyas Phillips
- Dan Ros
- Thomas Eichinger
- Michael Richardson
- Emmanuel Baccelli
- Ned Smith
- Jim Schaad
- Carsten Bormann
- Cullen Jennings
- Olaf Bergmann
- Suhas Nandakumar
- Phillip Hallam-Baker
- Marti Bolivar
- Andrzej Puzdrowski
- Markus Gueller
- Henk Birkholz
- Jintao Zhu
- Takeshi Takahashi
- Jacob Beningo
We would also like to thank the WG chairs, Russ Housley, David Waltermire, Dave Thaler for their support and their reviews.

14. References

14.1. Normative References


14.2. Informative References


14.3. URIs

[1] mailto:suit@ietf.org


Authors’ Addresses

Brendan Moran
Arm Limited

EMail: Brendan.Moran@arm.com

Milosch Meriac
Consultant

EMail: milosch@meriac.com

Hannes Tschofenig
Arm Limited

EMail: hannes.tschofenig@arm.com

David Brown
Linaro

EMail: david.brown@linaro.org
Firmware Updates for Internet of Things Devices - An Information Model for Manifests
draft-ietf-suit-information-model-02

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.

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

Status of This Memo

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

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

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

This Internet-Draft will expire on July 22, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.
This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

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

Table of Contents

1. Introduction ..................................................... 5
2. Conventions and Terminology .................................. 5
3. Manifest Information Elements ................................. 5
   3.1. Manifest Element: version identifier of the manifest structure ................................... 5
   3.2. Manifest Element: Monotonic Sequence Number ........... 6
   3.3. Manifest Element: Vendor ID Condition .................. 6
       3.3.1. Example: Domain Name-based UUIDs .................. 6
   3.4. Manifest Element: Class ID Condition ................... 6
       3.4.1. Example 1: Different Classes ....................... 7
       3.4.2. Example 2: Upgrading Class ID ..................... 7
       3.4.3. Example 3: Shared Functionality ................... 8
   3.5. Manifest Element: Precursor Image Digest Condition .... 8
   3.7. Manifest Element: Best-Before timestamp condition ...... 9
   3.8. Manifest Element: Payload Format ....................... 9
   3.9. Manifest Element: Processing Steps ..................... 9
   3.10. Manifest Element: Storage Location ................... 9
       3.10.1. Example 1: Two Storage Locations ................. 10
       3.10.2. Example 2: File System .......................... 10
       3.10.3. Example 3: Flash Memory ......................... 10
   3.11. Manifest Element: Component Identifier ............... 10
   3.12. Manifest Element: URIs ................................ 10
   3.13. Manifest Element: Payload Digest ........................ 11
3.15. Manifest Element: Signature ........................... 11
3.16. Manifest Element: Directives .......................... 11
3.17. Manifest Element: Aliases .............................. 12
3.18. Manifest Element: Dependencies ......................... 12
3.19. Manifest Element: Content Key Distribution Method .......... 12
3.20. Manifest Element: XIP Address ......................... 12
4. Motivation for Manifest Fields ........................... 13
4.1. Threat Model ........................................... 13
4.2. Threat Descriptions .................................... 13
  4.2.1. Threat MFT1: Old Firmware .......................... 13
  4.2.2. Threat MFT2: Mismatched Firmware .................... 14
  4.2.3. Threat MFT3: Offline device + Old Firmware ............ 14
  4.2.4. Threat MFT4: The target device misinterprets the type of payload ................................................................. 15
  4.2.5. Threat MFT5: The target device installs the payload to the wrong location ............................................................. 15
  4.2.6. Threat MFT6: Redirection .............................. 15
  4.2.7. Threat MFT7: Payload Verification on Boot .............. 16
  4.2.8. Threat MFT8: Unauthenticated Updates .................. 16
  4.2.9. Threat MFT9: Unexpected Precursor images ............. 16
  4.2.10. Threat MFT10: Unqualified Firmware .................. 17
  4.2.11. Threat MFT11: Reverse Engineering Of Firmware Image for Vulnerability Analysis ........................................ 18
  4.2.12. Threat MFT12: Overriding Critical Manifest Elements . 18
  4.2.13. Threat MFT13: Manifest Element Exposure ............. 18
4.3. Security Requirements ................................. 19
  4.3.1. Security Requirement MFSR1: Monotonic Sequence Numbers ................................................................. 19
  4.3.2. Security Requirement MFSR2: Vendor, Device-type Identifiers ................................................................. 19
  4.3.3. Security Requirement MFSR3: Best-Before Timestamps .... 19
  4.3.4. Security Requirement MFSR5: Cryptographic Authenticity ................................................................. 20
  4.3.5. Security Requirement MFSR4a: Authenticated Payload Type ................................................................. 20
  4.3.6. Security Requirement MFSR4b: Authenticated Storage Location ................................................................. 20
  4.3.7. Security Requirement MFSR4c: Authenticated Remote Resource Location ................................................................. 20
  4.3.8. Security Requirement MFSR4d: Secure Boot ................ 21
  4.3.9. Security Requirement MFSR4e: Authenticated precursor images ................................................................. 21
  4.3.10. Security Requirement MFSR4f: Authenticated Vendor and Class IDs ................................................................. 21
  4.3.11. Security Requirement MFSR4f: Authenticated Vendor and Class IDs ................................................................. 21
4.3.13. Security Requirement MFSR7: Firmware encryption ................................................. 22
4.3.15. Security Requirement MFSR9: Encrypted Manifests .................................................... 22
4.4. User Stories .................................................. 23
4.4.1. Use Case MFUS1: Installation Instructions ................................................................. 23
4.4.2. Use Case MFUS2: Override Non-Critical Manifest Elements .......................................... 23
4.4.3. Use Case MFUS3: Modular Update .................................................................................. 24
4.4.4. Use Case MFUS4: Multiple Authorisations ...................................................................... 24
4.4.5. Use Case MFUS5: Multiple Payload Formats ................................................................. 24
4.4.6. Use Case MFUS6: Prevent Confidential Information Disclosures ..................................... 24
4.4.7. Use Case MFUS7: Prevent Devices from Unpacking Unknown Formats .............................. 24
4.4.8. Use Case MFUS8: Specify Version Numbers of Target Firmware ......................................... 24
4.4.9. Use Case MFUS9: Enable Devices to Choose Between Images ............................................ 25
4.4.10. Use Case MFUS10: Secure Boot Using Manifests ............................................................ 25
4.4.11. Use Case MFUS11: Decompress on Load ........................................................................ 25
4.4.12. Use Case MFUS12: Payload in Manifest ......................................................................... 26
4.4.13. Use Case MFUS13: Simple Parsing ................................................................................. 26
4.5. Usability Requirements .................................................. 26
4.5.1. Usability Requirement MFUR1 ....................................................................................... 26
4.5.2. Usability Requirement MFUR2 ....................................................................................... 26
4.5.3. Usability Requirement MFUR3 ....................................................................................... 27
4.5.4. Usability Requirement MFUR4 ....................................................................................... 28
4.5.5. Usability Requirement MFUR5 ....................................................................................... 28
4.5.6. Usability Requirement MFUR6 ....................................................................................... 28
4.5.7. Usability Requirement MFUR7 ....................................................................................... 28
4.5.8. Usability Requirement MFUR8 ....................................................................................... 29
4.5.9. Usability Requirement MFUR9: Bootable Manifest ........................................................ 29
4.5.10. Usability Requirement MFUR10: Load-Time Information ............................................... 29
4.5.11. Usability Requirement MFUR11: Payload in Manifest Superstructure .............................. 29
4.5.12. Usability Requirement MFUR12: Simple Parsing ............................................................ 30
5. Security Considerations .................................................. 30
6. IANA Considerations .................................................. 30
7. Acknowledgements .................................................. 30
8. References .................................................. 30
8.1. Normative References .................................................. 30
8.2. Informative References .................................................. 31
Appendix A. Mailing List Information .................................................. 32
Authors’ Addresses .................................................. 32
1. Introduction

The information model describes all the information elements required to secure firmware updates of IoT devices from the threats described in Section 4.1 and enable the user stories captured in Section 4.4. These threats and user stories are not intended to be an exhaustive list of the threats against IoT devices, nor of the possible use cases of firmware update; instead they are intended to describe the threats against firmware update in isolation and provide sufficient motivation to provide information elements that cover a wide range of use cases. The information model does not define the encoding, ordering, or structure of information elements, only their semantics.

Because the information model covers a wide range of user stories and a wide range of threats, not all information elements apply to all scenarios. As a result, many information elements could be considered optional to implement and optional to use, depending on which threats exist in a particular system and which use cases are required. Elements marked as mandatory provide baseline security and usability properties that are expected to be required for most applications. Those elements are mandatory to implement and mandatory to use. Elements marked as recommended provide important security or usability properties that are needed on most devices. Elements marked as optional enable security or usability properties that are useful in some applications.

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

3. Manifest Information Elements

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

3.1. Manifest Element: version identifier of the manifest structure

An identifier that describes which iteration of the manifest format is contained in the structure.
This element is MANDATORY and must be present in order to allow devices to identify the version of the manifest data model that is in use.

3.2. Manifest Element: Monotonic Sequence Number

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

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


3.3. Manifest Element: 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.

This ID is RECOMMENDED and helps to distinguish between identically named products from different vendors.

Implements: Security Requirement MFSR2, MFSR4f.

3.3.1. Example: Domain Name-based UUIDs

Vendor A creates a UUID based on their domain name:

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

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

3.4. Manifest Element: Class ID Condition

A device "Class" is defined as any device that can accept the same firmware update without modification. Class Identifiers MUST be unique within a Vendor ID. This is to prevent similarly, or identically named devices colliding in their customer’s
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. 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.

Note Well: Class ID is not a human-readable element. It is intended for match/mismatch use only.

This ID is RECOMMENDED and allows devices to determine applicability of a firmware in an unambiguous way.

Implements: Security Requirement MFSR2, MFSR4f.

3.4.1. Example 1: Different Classes

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

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

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

3.4.2. Example 2: Upgrading Class ID

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

Vendor A creates UUIDs as follows:

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

When product X receives the firmware update necessary to be compatible with product X v2, part of the firmware update changes the class ID to Xv2classId.
3.4.3. Example 3: Shared Functionality

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

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

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

3.5. Manifest Element: 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. The precursor image may be installed or stored as a candidate.

This element is MANDATORY for differential updates. Otherwise, it is not needed.

Implements: Security Requirement MFSR4e

3.6. Manifest Element: Required Image Version List

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

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

This element is OPTIONAL.

Implements: MFUR7
3.7. Manifest Element: Best-Before timestamp condition

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

This element is OPTIONAL and MAY enable use cases where a secure clock is provided and firmware is intended to expire regularly.

Implements: Security Requirement MFSR3

3.8. Manifest Element: Payload Format

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

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

Implements: Security Requirement MFSR4a, Usability Requirement MFUR5

3.9. Manifest Element: Processing Steps

A list of all payload processors necessary to process a nested format and any parameters needed by those payload processors. Each Processing Step SHOULD indicate the expected digest of the payload after the processing is complete. Processing steps are distinct from Directives in that Directives apply to the manifest as a whole, whereas Processing Steps apply to an individual payload and provide instructions on how to unpack it.

Implements: Usability Requirement MFUR6

3.10. Manifest Element: Storage Location

This element 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.

This element is MANDATORY and MUST be present to enable devices to store payloads to the correct location.

Implements: Security Requirement MFSR4b
3.10.1. Example 1: Two Storage Locations

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

- OS
- APP

3.10.2. Example 2: File System

A device supports a full filesystem. The firmware authority chooses to make the storage identifier the path at which to install the payload. The payload may be a tarball, in which case, it unpacks the tarball into the specified path.

3.10.3. Example 3: Flash Memory

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

3.11. Manifest Element: Component Identifier

In a heterogeneous storage architecture, a storage identifier is insufficient to identify where and how to store a payload. To resolve this, a component identifier indicates which part of the storage architecture is targeted by the payload. In a homogeneous storage architecture, this element is unnecessary.

This element is OPTIONAL and only necessary in heterogeneous storage architecture devices.

Implements: MFUR3

3.12. Manifest Element: URIs

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

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

Note: Devices will typically require URIs.

Implements: Security Requirement MFSR4c
3.13. Manifest Element: Payload Digest

This element contains the digest of the payload. This allows the target device to ensure authenticity of the payload. It MUST be possible to specify more than one payload digest, indexed by Manifest Element: XIP Address.

This element is MANDATORY and fundamentally necessary to ensure the authenticity and integrity of the payload.

Implements: Security Requirement MFSR4d, Usability Requirement MFUR8


The size of the payload in bytes.

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

Implements: Security Requirement MFSR4d

3.15. Manifest Element: Signature

This is not strictly a manifest element. 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.

This element is MANDATORY and represents the foundation of all security properties of the manifest.

Implements: Security Requirement MFSR5, MFSR6, MFUR4

3.16. Manifest Element: Directives

A list of instructions that the device should execute, in order, when processing the manifest. This information is distinct from the information necessary to process a payload (Processing Steps) and applies to the whole manifest including all payloads that it references. Directives include information such as update timing (for example, install only on Sunday, at 0200), procedural considerations (for example, shut down the equipment under control before executing the update), pre and post-installation steps (for example, run a script).

This element is OPTIONAL and enables some use cases.
3.17. Manifest Element: Aliases

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

This element is OPTIONAL and enables some use cases.

Implements: Usability Requirement MFUR2

3.18. Manifest Element: Dependencies

A list of Digest/URI 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.

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

Implements: Usability Requirement MFUR3

3.19. Manifest Element: 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 an element for the Content Key Distribution Method. One example 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. This MAY be included in a decryption step contained in Processing Steps.

This element is MANDATORY to use for encrypted payloads,


3.20. Manifest Element: XIP Address

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

For example a microcontroller may have a simple bootloader that chooses one of two images to boot. That microcontroller then needs to choose one of two firmware images to install, based on which of its two images is older.
3.21. Manifest Element: Load-time metadata

## Manifest Element: Boot-time metadata ##
## Manifest Element: Payload

### 4. Motivation for Manifest Fields

The following sub-sections describe the threat model, user stories, security requirements, and usability requirements.

#### 4.1. Threat Model

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

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

This threat model only covers elements related to the transport of firmware updates. It explicitly does not cover threats outside of the transport of firmware updates. For example, threats to an IoT device due to physical access are out of scope.

#### 4.2. Threat Descriptions

##### 4.2.1. Threat MFT1: Old Firmware

Classification: Elevation of Privilege

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

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

Mitigated by: MFSR1

4.2.2. 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

Example:

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

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

4.2.3. Threat MFT3: Offline device + Old Firmware

Classification: Elevation of Privilege

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

Mitigated by: MFSR3

4.2.4. 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 elevation of privilege and potentially expand this to all types of threat.

Mitigated by: MFSR4a

4.2.5. 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 elevation of privilege and potentially expand this to all types of threat.

Mitigated by: MFSR4b

4.2.6. 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: MFSR4c
4.2.7. Threat MFT7: Payload Verification on Boot

Classification: Elevation of Privilege

An attacker replaces a newly downloaded firmware after a device finishes verifying a manifest. This could cause the device to execute the attacker’s code. This attack likely requires physical access to the device. However, it is possible that this attack is carried out in combination with another threat that allows remote execution.

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

Mitigated by: MFSR4d

4.2.8. Threat MFT8: Unauthenticated Updates

Classification: Elevation of Privilege

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

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

Mitigated by: MFSR5

4.2.9. 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 elevation of privilege and potentially expand this to all types of threat.

Mitigated by: MFSR4e
4.2.10. Threat MFT10: Unqualified Firmware

Classification: Denial of Service, Elevation 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.

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

Mitigated by: MFSR6, MFSR8

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

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

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

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

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

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

4.2.11. 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

4.2.12. Threat MFT12: Overriding Critical Manifest Elements

Classification: Elevation of Privilege

An authorised actor, but not the firmware authority, uses an override mechanism (MFUS2) to change an information element in a manifest signed by the firmware authority. For example, if the authorised actor overrides the digest and URI of the payload, the actor can replace the entire payload with a payload of their choice.

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

Mitigated by: MFSR8

4.2.13. Threat MFT13: Manifest Element Exposure

Classification: Information Disclosure

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

Mitigated by: MFSR9
4.3. Security Requirements

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

4.3.1. 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. If UTC epoch timestamps are used, they MUST NOT be treated as times, they MUST be treated only as sequence numbers. Devices MUST reject manifests with sequence numbers smaller than any onboard sequence number.

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

Mitigates: Threat MFT1

Implemented by: Manifest Element: Monotonic Sequence Number

4.3.2. 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 Elements: Vendor ID Condition, Class ID Condition

4.3.3. 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 Element: Best-Before timestamp condition
4.3.4. Security Requirement MF5S: Cryptographic Authenticity

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

Mitigates: Threat MFT8

Implemented by: Signature, Payload Digest

4.3.5. Security Requirement MF5Sr4a: Authenticated Payload Type

The type of payload (which may be independent of format) MUST be authenticated. For example, the target must know whether the payload is XIP firmware, a loadable module, or serialized configuration data.

Mitigates: MFT4

Implemented by: Manifest Elements: Payload Format, Storage Location

4.3.6. Security Requirement MF5Sr4b: Authenticated Storage Location

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

Mitigates: MFT5

Implemented by: Manifest Elements: Storage Location

4.3.7. Security Requirement MF5Sr4c: Authenticated Remote Resource Location

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

Mitigates: MFT6

Implemented by: Manifest Elements: URIs
4.3.8. Security Requirement MFSR4d: Secure Boot

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

Mitigates: MFT7

Implemented by: Manifest Elements: Payload Digest, Size

4.3.9. Security Requirement MFSR4e: Authenticated precursor images

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

Mitigates: MFT9

Implemented by: Manifest Elements: Precursor Image Digest Condition

4.3.10. Security Requirement MFSR4f: Authenticated Vendor and Class IDs

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

Mitigates: MFT2

Implemented By: Manifest Elements: Vendor ID Condition, Class ID Condition

4.3.11. Security Requirement MFSR4f: Authenticated Vendor and Class IDs

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

Mitigates: MFT2

Implemented By: Manifest Elements: Vendor ID Condition, Class ID Condition


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 a Device Operator and a Network Operator, respectively, then the firmware cannot be installed without proof of rights from both the Device and the Network Operator.

Mitigates: MFT10

Implemented by: Signature

4.3.13. Security Requirement MFSR7: Firmware encryption

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

Mitigates: MFT11

Implemented by: Manifest Element: Content Key Distribution Method


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

1. An ACL decides which elements of the manifest may be overridden and by which actors.

2. An ACL decides which component identifier/storage identifier pairs can be written by which actors.

Mitigates: MFT12, MFT10

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

4.3.15. Security Requirement MFSR9: Encrypted Manifests

It must be possible to encrypt part or all of the manifest. This may be accomplished with either transport encryption or with at-rest encryption, for example COSE_Encrypt.
4.4. User Stories

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

4.4.1. Use Case MFUS1: Installation Instructions

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

Some installation instructions might be:

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

Satisfied by: MFUR1

4.4.2. Use Case MFUS2: Override Non-Critical Manifest Elements

As a Network Operator, I would like to be able to override the non-critical information in the manifest so that I can control my devices more precisely. This assumes that the Device Operator delegated rights about the device to the Network Operator.

Some examples of potentially overridable information:

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

Satisfied by: MFUR2, MFUR3

4.4.3. Use Case MFUS3: Modular Update

As an Operator, 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

4.4.4. Use Case MFUS4: Multiple Authorisations

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

Satisfied by: MFUR4, MFSR8

4.4.5. Use Case MFUS5: Multiple Payload Formats

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

Satisfied by: MFUR5

4.4.6. Use Case MFUS6: Prevent Confidential Information Disclosures

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

Satisfied by: MFSR7

4.4.7. Use Case MFUS7: Prevent Devices from Unpacking Unknown Formats

As a Device Operator, I want devices to determine whether they can process a payload prior to downloading it.
In some cases, it may be desirable for a third party to perform some processing on behalf of a target. For this to occur, the third party MUST indicate what processing occurred and how to verify it against the Trust Provisioning Authority’s intent.

This amounts to overriding Processing Steps and URIs.

Satisfied by: MFUR6, MFUR2

4.4.8. Use Case MFUS8: Specify Version Numbers of Target Firmware

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

Satisfied by: MFUR7

4.4.9. Use Case MFUS9: Enable Devices to Choose Between Images

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

Satisfied by: MFUR8

4.4.10. Use Case MFUS10: Secure Boot Using Manifests

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

Satisfied by: MFUR9

4.4.11. Use Case MFUS11: Decompress on Load

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

Satisfied by: MFUR10
4.4.12. Use Case MFUS12: Payload in Manifest

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

Satisfied by: MFUR11

4.4.13. Use Case MFUS13: Simple Parsing

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

Satisfied by: MFUR12

4.5. Usability Requirements

The following usability requirements satisfy the user stories listed
above.

4.5.1. Usability Requirement MFUR1

It must be possible to provide all information necessary for the
processing of a manifest into the manifest.

Satisfies: User story MFUS1

Implemented by: Manifest Element: Directives

4.5.2. Usability Requirement MFUR2

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

Satisfies: User story MFUS2

Implemented by: Manifest Element: Aliases
4.5.3. Usability Requirement MFUR3

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

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

Because devices can be either HeSA or HoSA both the storage system and the storage location within that storage system must be possible to specify. In a HoSA device, the payload location may be as simple as an address, or a file path. In a HeSA device, the payload location may be scoped by a component identifier. It is expedient to consider that all HoSA devices are HeSA devices with a single component.

4.5.3.1. Example 1: Multiple Microcontrollers

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

4.5.3.2. Example 2: Code and Configuration

A firmware image can be divided into two payloads: code and configuration. These payloads may require authorizations from different actors in order to install (see MFUR6 and MFUR8). This structure means that multiple manifests may be required, with a dependency structure between them.

4.5.3.3. Example 3: Multiple Chunks

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

Satisfies: User story MFUS2, MFUS3

Implemented by Manifest Element: Dependencies, StorageIdentifier, ComponentIdentifier
4.5.4. 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: User story MFUS4

Implemented by: COSE Signature (or similar)

4.5.5. Usability Requirement MFUR5

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

- Binary
- Elf
- Differential
- Compressed
- Packed configuration
- Intel HEX
- S-Record

Satisfies: User story MFUS5

Implemented by: Manifest Element: Payload Format

4.5.6. Usability Requirement MFUR6

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

Satisfies: User story MFUS6

Implemented by: Manifest Element: Processing Steps

4.5.7. Usability Requirement MFUR7

The manifest format must provide a method to specify multiple version numbers of firmware to which the manifest applies, either with a list or with range matching.
4.5.8. Usability Requirement MFUR8

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

Satisfies: User story MFUS9

Implemented by: Manifest Element: XIP Address

4.5.9. Usability Requirement MFUR9: Bootable Manifest

It must be possible to describe a bootable system with a manifest on both Execute-In-Place microcontrollers and on complex operating systems. This requires the manifest to specify the digest of each statically linked storage location. In addition, the manifest must be able to express metadata used by the bootloader, such as a kernel command-line.

Satisfies: User story MFUS10

Implemented by: Manifest Element: Boot-time Metadata

4.5.10. Usability Requirement MFUR10: Load-Time Information

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

N.B. load comes before boot.

Satisfies: User Story MFUS11

Implemented by: Manifest Element: Load-time Metadata

4.5.11. Usability Requirement MFUR11: Payload in Manifest Superstructure

It must be possible to place a payload in the same structure as the manifest. This typically places the payload in the same packet as the manifest.

Satisfies: User Story MFUS12
4.5.12. Usability Requirement MFUR12: Simple Parsing

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

Satisfies: User Story MFUS13

5. Security Considerations

Security considerations for this document are covered in Section 4.

6. IANA Considerations

This document does not require any actions by IANA.

7. Acknowledgements

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

We would like to thank the participants of the 2018 Berlin SUIT Hackathon and the June 2018 virtual design team meetings for their discussion input. In particular, we would like to thank Koen Zandberg, Emmanuel Baccelli, Carsten Bornmann, David Brown, Markus Gueller, Frank Audun Kvamtro, Oyvind Ronningstad, Michael Richardson, Jan-Frederik Rieckers Francisco Acosta, Anton Gerasimov, Matthias Waehlisch, Max Groening, Daniel Petry, Gaetan Harter, Ralph Hamm, Steve Patrick, Fabio Utzig, Paul Lambert, Benjamin Kaduk, Said Gharout, and Milen Stoychev.

8. References

8.1. Normative References

[I-D.ietf-suit-architecture]
8.2. Informative References


8.3. URIs

[1] mailto:suit@ietf.org


Appendix A. 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]

Authors’ Addresses

Brendan Moran
Arm Limited

EMail: Brendan.Moran@arm.com

Hannes Tschofenig
Arm Limited

EMail: hannes.tschofenig@gmx.net

Henk Birkholz
Fraunhofer SIT

EMail: henk.birkholz@sit.fraunhofer.de
An Information Model for Behavioural Description of Firmware Update and Related Operations
draft-moran-suit-behavioural-manifest-01

Abstract

This specification describes an approach to formally defining the behaviour of a system under firmware update and secure boot conditions. The behavioural documents described here can be used with [Information] to construct a firmware update manifest.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 12, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

Table of Contents

1. Introduction ........................................... 3
2. Conventions and Terminology ............................ 3
3. Design Principles of the Behavioural Manifest ............ 4
4. Structure of a behavioural manifest ..................... 5
   4.1. Processing Steps .................................. 7
5. Commands ................................................ 7
   5.1. Verify Recipient Identity .......................... 8
   5.2. Verify Image Presence .............................. 9
   5.3. Verify Component Properties ....................... 9
   5.4. Verify System Properties ........................... 9
   5.5. Verify 3rd-party Authorisation .................... 9
   5.6. Process sub-behaviours ............................. 9
   5.7. Process Dependencies ............................... 9
   5.8. Set Parameters .................................... 10
   5.9. Move an Image ..................................... 10
   5.10. Invoke an Image ................................... 10
   5.11. Wait for an Event ................................ 10
6. Parameters ............................................... 11
   6.1. Strict Order ....................................... 11
   6.2. Soft Failure ....................................... 11
   6.3. Source List ....................................... 11
   6.4. Processing Step Configurations ..................... 12
   6.5. Image Identifier ................................. 12
7. ACLs/permissions ......................................... 12
8. Workflows ............................................... 13
9. Examples ................................................. 15
   9.1. Example 1: Boot an image on an XIP processor ...... 16
   9.2. Example 2: Download an image ....................... 16
   9.3. Example 3: Check compatibility, download, and boot ... 17
   9.4. Example 4: Check compatibility, download, load from external, and boot .................. 17

1.  Introduction

Conventional hierarchical, descriptive documents, such as draft-moran-manifest-03 imply the behaviour of the recipient without specifying that behaviour. This creates a situation where recipients must construct the assumed behaviour in accordance with a specification, handling many edge cases and introducing significant complexity. Capabilities are difficult to specify because they imply behaviours, rather than data, but the descriptive document only specifies data, not capabilities. This leaves the document author to interpret capabilities (supported behaviours) into allowable combinations of data. This disconnect demonstrates that devices require both an information model and a behavioural model.

This creates a situation where the behaviour of a system is imprecisely specified by the documents that it uses to perform secure boot and secure firmware update. In high security applications, precise specification of behaviour is beneficial, and can even be used for formal verification.

By specifying the behaviour of a device in a document rather than just the information, the gap between specified information and specified behaviour can be closed.

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].

- SUIT: Software Update for the Internet of Things, the IETF working group for this standard.
- Image: A piece of information to be delivered. Typically Firmware for the purposes of SUIT.

- Document, Behavioural Document: The data that defines the behaviour of a recipient.

- Component: A target for storage of the Image

- Dependency: Another Behavioural Document upon which the current Document relies.

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

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


- Command: A Condition or a Directive.

3. Design Principles of the Behavioural Manifest

In order to provide flexible behaviour to constrained devices, while still allowing more powerful devices to use their full capabilities, the SUIT manifest takes a new approach, encoding the required behaviour of a Recipient device, instead of just presenting the information used to determine that behaviour. This gives benefits equivalent to those provided by a scripting language or 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 secure boot of a firmware image. The language specifies behaviours in a linearised form, without branches or loops. 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 consists 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 with minimal functionality to perform complex updates with reduced overhead. Conditional execution of commands allows a simple device to perform important decisions at validation-time, such as which differential update to download for a given current version, or which hash to check, based on the installation address.
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 define a component.

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

Because the behavioural description is precise, and the machine definition upon which it relies is very simple it can be augmented with a proof that the effects of an update fall within a specified policy, in the same way as Proof Carrying Code. By combining this capability with formal verification of the document processor, it is possible to prove the result of a firmware update, prior to application, either on the target or on an intermediate system. The proof can be discarded before distribution to constrained nodes, creating no additional overhead.

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

4. Structure of a behavioural manifest

Behavioural manifests are divided into sections based on the behaviours of the Recipient. There are 8 conceptual sections of a behavioural manifest, listed below.

1. Document-global data
2. Common behaviour
3. Dependency Resolution behaviour
4. Image Acquisition behaviour
5. Image Application behaviour
6. System Validation behaviour

7. Image Loading behaviour

8. Image Invocation behaviour

Document-global data contains the information that is required to enable most behaviours along with a security parameter. The information contained is listed below.

1. Document Structure Version

2. Document Sequence Number

3. List of Dependencies
   1. List of Components affected by each Dependency

4. List of Components affected by this Document

Common behaviour is executed prior to each other behaviour. It is used to make common decisions for all other behaviours.

Dependency Resolution is used to ensure that all required documents have been collected prior to attempting to acquire any image. Where a document has no dependencies, this section is not required.

Image Acquisition is used to obtain images from local or remote sources and stage them for use by the Recipient. If a Document lists no affected components, then Image Acquisition is not required. If a device operates in a simultaneous Acquisition & Application mode (for example, streaming installation), then Image Acquisition should be discarded in favour of Image Application. Image Acquisition can be used in combination with several processing steps defined in Section 4.1.

Image Application is used to place an image into its long-term storage. An image can be moved either from a staging area or from another source (including a remote) into its long-term storage. This can be done in combination with several processing steps defined in Section 4.1.

System validation is used to ensure that all required dependencies are present and that all required images are present. This process is equivalent to that used in the validation portion of Secure Boot workflows.
Image Loading is used to ensure that all required images are moved from long-term storage to active use storage. This can include steps like copying from external Flash to RAM, as defined by Component information. This can be done in combination with several of the processing steps defined in Section 4.1.

Image Invocation is used to finalise the manifest processor’s behaviour and forward execution to a designated component. This is equivalent to Bootloader behaviour.

Some behaviours need only a single successful invocation. These behaviours can then be discarded, provided that the Document serialisation provides a mechanism to do so. Typically discarded behaviours are Dependency Resolution, Image Acquisition, and Image Application.

4.1. Processing Steps

Processing steps are the translation that is performed on an image prior to its execution. These steps typically include, in order, symmetric cryptographic operations, decompression operations, unpacking operations.

Each of these operation may need additional information, such as which algorithm is in use or arguments to that algorithm, such as key identifiers for cryptographic operations. This information can be encoded in Processing Step parameters, as described in Section 6.4.

5. Commands

Behaviours are constructed as lists of commands, each of which may have arguments. The behaviours listed in any of the specified sections derives from a short list of commands. These commands are divided into two types, Conditions (verification operations) and Directives (action operations).

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:

```
Behaviour = [ + { conditions => [ * Condition], directives => [ * Directive] } ]
```
The conditions form preconditions that MUST be true for the following sequence of directives to be executed.

However, this organisation could introduce significant complexity in a parser, so the structure MAY be flattened into the following:

\[
\text{Behaviour} = [ \ast (\text{Condition/Directive}) ]
\]

This does not alter the logical organisation of sequences of preconditions that precede sequences of directives, but it simplifies the consumption of commands in behaviours.

The Conditions are, broadly, those listed below.

1. Verify device identity
2. Verify image presence (correctness) or absence
3. Verify component properties
4. Verify system properties
5. Verify 3rd-party authorisation

The Directives are those listed below.

1. Process sub-behaviours
2. Process dependencies
3. Set parameters
4. Move an Image or Document
5. Invoke an Image
6. Wait for an event

5.1. Verify Recipient Identity

This is used to ensure that the document is being processed by the appropriate device and to eliminate incompatibility failures. Identity can include what sort of device is targeted, what software it uses, or who made it. Identity can also include the particular device that is targeted.
5.2. Verify Image Presence

This is used to ensure that a required image is present. This often includes the use of cryptographic checksums to validate the contents of an image contained in a component.

5.3. Verify Component Properties

This can be used to verify several properties of a targeted component, such as the current nominal version of its APIs, or the base address or offset that it will use.

5.4. Verify System Properties

This can be used to verify several properties of the system including, the current power state, such as battery level or presence of external power, the current time reported by the device, or the current state of a controlled piece of equipment.

5.5. Verify 3rd-party Authorisation

This can be used to ensure that some third-party has approved an action, in a system specific way. Options include checking a remote system for authorisation, looking for a cryptographic token, or invoking a user-interface.

5.6. Process sub-behaviours

In some use-cases, a decision must be made as to which of several behaviours must be invoked. To enable this use-case, sub-behaviours provide a mechanism to permit soft-failure of a Condition. A parameter of the sub-behaviour controls its response to a Condition check failure, allowing the command following the sub-behaviour to be the next to execute, or causing failure of the whole behaviour, depending on its value.

This allows the construction of a conditional behaviour. A sub-behaviour is invoked allowing condition checks to soft-fail. Once the conditions that inform the conditional behaviour have succeeded, the soft-failure parameter is switched to hard-failure, so that further condition failures will be detected.

5.7. Process Dependencies

Dependencies are processed by invoking two behaviours within the dependency; first the common behaviour is invoked, then the behaviour matching the current behaviour of the current document is invoked. So, if "Image Application" is active when "Process Dependencies" is
invoked, then each dependency’s "Common" behaviour will be invoked, followed by its "Image Application" behaviour.

It can be advantageous to process dependencies in a particular order, so dependencies are processed in the order specified.

A failure of processing of any dependency results in a failure of the Process Dependencies behaviour.

5.8. Set Parameters

Many Commands are partially governed by configuration present in the form of parameters. Parameters control the source used for Image Acquisition, the processing steps applied to those images, the order in which commands are processed and more. See Section 6 for more information.

Parameters can be set in one of three ways. They can be set-if-unset (the default), append-if-set (typically used for source lists), set-always (used for critical parameters). Parameters are either global or scoped by Component/Component Group.

5.9. Move an Image

This Command directs the document processor to acquire an Image or Document and store it to a specified Component or Document storage, respectively. The source can be local or remote, or a prioritised list of local and remote sources. The source or source list is specified by the source parameter. The Image or document can optionally be modified in transit by a sequence of processing steps, as defined in Section 6.

5.10. Invoke an Image

This command forwards execution to the specified image in much the same way as a bootloader. As with bootloaders, the semantics of forwarding execution are application defined. An argument may be provided to the Image. The semantics of the argument are application-defined.

5.11. Wait for an Event

Frequently, a behaviour needs to wait for a property of the system to change. This may be a message from a remote, a time, a power state, a user-interaction, or some other system parameter.
6. Parameters

Available parameters may vary by implementation, but some core parameters are usually present.

Typical parameters are listed below.

1. Strict Order
2. Soft-Failure
3. Source List
4. Processing Step Configuration
5. Image Identifier

In some use-cases, device identity may also be configured in a parameter.

6.1. Strict Order

Some advanced devices may have particular requirements regarding command ordering within a behaviour. Others may enable parallel execution of commands. When the Strict Order parameter is set to False, these extended capabilities are enabled. An advanced device may then aggregate all successive commands up until the behaviour ends or the Strict Order parameter is returned to True and process those commands in parallel or reorder them as it requires. Strict Order defaults to True. If a device does not support command reordering or parallel processing, Strict Order = False has no effect.

6.2. Soft Failure

When a device invokes a sub-behaviour, any condition check failure and any directive failure causes the behaviour to immediately abort. However, if the Soft Failure parameter is True, then an abort due to a condition failure does not cause the sub-behaviour to report failure. If the Soft Failure parameter is True, indicating hard failure, then any abort causes the sub-behaviour to report failure as well.

6.3. Source List

The source list is scoped to an individual component or dependency. It is a prioritised search path for the Move command to use in order
6.4. Processing Step Configurations

Many processing steps require configuration to operate, or configuration informs whether or not to use them. Common processing steps include symmetric cryptography, compression or decompression operations, and packing or unpacking, for example relocation, differential compression, or hex file interpretation.

Processing step configuration is scoped to an individual component or document.

6.5. Image Identifier

In order to determine whether an image is present, the verify image presence condition requires an identifier for the image. This could be a version number or a cryptographic identity such as a digest.

7. ACLs/permissions

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

First, the simplest model requires that all documents are authenticated by a single identity. This mode has the advantage that only a single document needs to be authenticated, since each document’s dependencies are uniquely identified in that document.

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 Components or Component Groups to the listed identities or identity groups. Any identity may verify that an image is present, but Moving an image into or out of a Component requires approval from the ACL.

A third model allows a Document Processor to provide even more fine-grained controls: The ACL lists the Component or Component Group that an identity may use, and also lists the commands that the identity may use in combination with that Component/Group.
8. Workflows

The two most common workflows are image installation and image invocation. Both of these workflows use a common component: `do_commands`.

`do_commands` uses the following pseudocode:

```plaintext
function do_commands(section, sequence)
    rc = SUCCESS
    foreach (command in $sequence)
        choose $command[Type]:
            case Sub-Behaviour:
                Load commands = $command[Argument][commands]
                Load parameters = System Parameters
                Load soft_failure = $parameters[Soft Failure]
                Call rc = do_commands(commands)
                if (soft_failure AND is_condition_failure(rc))
                    rc = SUCCESS
                endif
            endcase
            case Process Dependency:
                ; Note Dependency selection can be done via argument or parameter. May process multiple dependencies in a list.
                Load Dependency
                Load common = $Dependency[Common Sequence]
                Call rc = do_commands(Common, $common)
                if (rc is SUCCESS?)
                    Load $sequence = $Dependency[$section]
                    Call rc = do_commands($section, $sequence)
                endif
            endcase
            case Set Parameters:
                Load parameter_list = $command[Argument][Parameter List]
                foreach (parameter in parameter_list)
                    if (is_append(parameter)?)
                        Append argument value to $parameter
                    elseif (is_set($parameter[Name]))
                        Set $parameter[Name] = $parameter[Value]
                    endif
                endfor
            endcase
            case Move:
                ; Note Component selection can be done via argument or parameter. May process multiple components in a list.
                ; Note Dependency selection can be done via argument or parameter. May process multiple dependencies in a list.
                ; Source
```
Load component_list = $command[Argument][Component List]
Load dependency_list = $command[Argument][Dependency List]
foreach (target_list in [component_list, dependency_list])
    foreach (target in $target_list)
        Load $target parameters
        Set source = choose_best_source($parameters[Source List])
    )
    Acquire data from $source
    foreach (processing_step in $parameters[Processing Step Configuration])
        rc = Process_Data($processing_step, $data)
    endforeach
    Store $data to $target
endforeach
endcase

case Invoke:
    ; Note Component selection can be done via argument or parameter. May process multiple components in a list.
    Select component
    Load Argument = $command[Argument]
    Transfer execution to $component with $Argument;
endcase

case Wait:
    Load arguments = $command[Argument][Wait Arguments]
    Load type = $command[Argument][Wait Type]
    Wait ($type, $arguments)
endcase

case Device Identity:
    ; Note Device Identity selection can be done via argument or parameter. May process multiple Device Identities in a list.
    Load device_identity = $parameters[Device Identity]
    if ($device_identity is nil)
        Load device_identity = $command[Argument][Device Identity]
    endif
    rc = Compare $device_identity to $parameters
endcase

case Image Present/not Present:
    ; Note Component selection can be done via argument or parameter. May process multiple components in a list.
    ; Note Dependency selection can be done via argument or parameter. May process multiple dependencies in a list.
    Load component_list = $command[Argument][Component List]
    Load dependency_list = $command[Argument][Dependency List]
    foreach (target_list in [component_list, dependency_list])
        foreach (target in $target_list)
            Load $target parameters
            Set image_identifier = $parameters[Image Identifier]
            rc = Compare $component to $image_identifier;
        endforeach
    endforeach
endcase
endfor
endcase

case Verify Authorisation
    Request authorisation
    Wait for authorisation
    rc = Check Response
endcase
endchoose
endwhile (no)
endfunction

Installation, is represented by the following pseudocode.

function install(Document)
    Load $Document[Common Data] into parameters
    foreach (sequence in [Common, Dependency Resolution, Image Acquisition, Image Application])
        rc = do_commands($sequence, $Document[$sequence]);
        if (rc is not SUCCESS)
            Abort
        endif
    endfor
endfunction

Image invocation is represented by the following pseudocode.

function invoke(Document)
    Load $Document[Common Data] into parameters
    foreach (sequence in [Common, System Validation, Image Loading, Image Invocation])
        rc = do_commands($sequence, $Document[$sequence]);
        if (rc is not SUCCESS)
            Abort
        endif
    endfor
endfunction

Each operation represented here is already present in a device capable of firmware update or secure boot. This approach simply defines the mechanism by which these operations are orchestrated, and enforces that the behaviour of the system is defined by the Behavioural Document, rather than implied by it.

9. Examples

These examples demonstrate the serialisation of the behaviours of an update. They are serialised in JSON for readability, but JSON is not recommended for use on constrained devices.
9.1. Example 1: Boot an image on an XIP processor

```
{
    "structure-version": 1,
    "sequence-number": 1,
    "components": [
        {
            "id": "<Component Identifier>",
            "digest": "<SHA256 of Image>",
            "size": "<Size of Image>
        }
    ],
    "validate": [
        {
            "condition-validate-image": {"component": 0}
        }
    ],
    "image-invocation": [
        {
            "directive-run-component": {"component": 0}
        }
    ]
}
```

9.2. Example 2: Download an image

```
{
    "structure-version": 1,
    "sequence-number": 2,
    "components": [
        {
            "id": "<Component Identifier>",
            "digest": "<SHA256 of Image>",
            "size": "<Size of Image>
        }
    ],
    "image-acquisition": [
        {
            "directive-move": {
                "source": "http://example.com/file.bin",
                "destination": 0
            }
        }
    ]
}
```
9.3. Example 3: Check compatibility, download, and boot

```json
{
   "structure-version": 1,
   "sequence-number": 3,
   "components": [
      {
         "id": <Component Identifier>,
         "digest": "<SHA256 of Image>",
         "size": <Size of Image>
      }
   ],
   "common": [
      {"condition-vendor-id": "fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe"},
      {"condition-class-id": "1492af14-2569-5e48-bf42-9b2d51f2ab45"}
   ],
   "image-application": [
      {"directive-move": {
         "source": "http://example.com/file.bin",
         "destination": 0
      }}
   ],
   "validate": [
      {"condition-validate-image": {"component": 0}}
   ],
   "image-invocation": [
      {"directive-run-component": {"component": 0}}
   ]
}
```

9.4. Example 4: Check compatibility, download, load from external, and boot

```json
{
   "structure-version": 1,
   "sequence-number": 4,
   "components": [
      {
         "id": <Flash Component Identifier>,
         "digest": "<SHA256 of Image>",
         "size": <Size of Image>
      }
   ]
}
```
9.5. Example 5: Check compatibility, download, load with decompress, and boot

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

  ]
}
{ "id" : <Flash Component Identifier>,
   "digest" : "<SHA256 of Compressed Image>",
   "size" : <Size of Compressed Image>
 },
{ "id" : <RAM Component Identifier>,
   "digest" : "<SHA256 of Image>",
   "size" : <Size of Image>
 }
],
"common" : [
   { "condition-vendor-id" : "fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe"},
   { "condition-class-id" : "1492af14-2569-5e48-bf42-9b2d51f0ab45"}
 ],
"image-application" : [
   { "directive-move" : {
      "source" : "http://example.com/file.bin",
      "destination" : 0
   }
  }
],
"validate" : [
   { "condition-validate-image" : {"component" : 0}
    }
],
"load-image" : [
   { "directive-move" : {
      "source" : 0,
      "destination" : 1,
      "processing-step-compression-algorithm" : "gzip"
   }
  }
],
"image-invocation" : [,  
   { "condition-validate-image" : {"component" : 1}
   },  
   { "directive-run-component" : {"component" : 1}
   }
]}

9.6. Example 6: Check compatibility, download, install-from-external and boot

```json
{
    "structure-version" : 1,
    "sequence-number" : 6,
    "components" : [
        {
            "id" : <External Flash Component Identifier>,
            "digest": "<SHA256 of Image>",
            "size" : <Size of Image>
        },
        {
            "id" : <Internal Flash Component Identifier>,
            "digest": "<SHA256 of Image>",
            "size" : <Size of Image>
        }
    ],
    "common" : [
        {
            "condition-vendor-id" : "fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe",
            "condition-class-id" : "1492af14-2569-5e48-bf42-9b2d51f2ab45"
        }
    ],
    "image-acquisition" : [
        {
            "directive-move" : {
                "source": "http://example.com/file.bin",
                "destination" : 0
            }
        }
    ],
    "validate" : [
        {
            "sub-behaviour" : [
                "soft-failure" : True,
                "condition-validate-not-image" : {"component" : 1}
            ],
            "condition-validate-image" : {"component" : 0}
        }
    ],
    "load-image" : [
        {
            "sub-behaviour" : [
                "soft-failure" : True,
                "condition-validate-not-image" : {"component" : 1}
            ],
            "directive-move" : {
                "source": 0,
            }
        }
    ]
}
```
9.7. Example 7: Download and boot an image with a dependency

[
  {
    "structure-version" : 1,
    "sequence-number" : 7,
    "components": [
      {
        "id" : <Component Identifier 0>,
        "digest" : "<SHA256 of Image>",
        "size" : <Size of Image>
      }
    ],
    "common" : [
      {"condition-vendor-id" : "fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe"},
      {"condition-class-id" : "1492af14-2569-5e48-bf42-9b2d51f2ab45"}
    ],
    "image-application" : [
      {
        "directive-move" : {
          "source" : "http://example.com/file.bin",
          "destination" : 0
        }
      }
    ],
    "validate" : [
      {
        "condition-validate-image" : {"component" : 0}
      }
    ],
    "image-invocation" : [
      {
        "directive-run-component" : {"component" : 0}
      }
    ]
  }
]
"structure-version" : 1,
"sequence-number" : 8,
"dependencies" : [
  {
    "digest" : "<SHA256 of Document 0>",
    "components" : [ <Component Identifier 0> ]
  },
  "components" : [
    {
      "id" : <Component Identifier 1>,
      "digest" : "<SHA256 of Image>",
      "size" : <Size of Image>
    }
  ],
  "dependency-resolution" : [
    {
      "directive-move" : {
        "source" : "http://example.com/document0.bin",
        "destination" : <Document 0 ID>
      }
    },
    {
      "condition-validate-image" : { "dependency" : 0 }
    }
  ],
  "image-application" : [
    {
      "directive-move" : {
        "source" : "http://example.com/file1.bin",
        "destination" : 1
      }
    },
    { "process-dependency" : 0 }
  ]
],
"validate" : [
  {
    "condition-validate-image" : { "dependency" : 0 }
  },
  { "process-dependency" : 0 },
  { "condition-validate-image" : { "image" : 1 } }
]
9.8. Example 8: Download and boot an image with a dependency using override.

Override fetch location for dependency:

```json
[
  {
    "structure-version" : 1,
    "sequence-number" : 7,
    "components": [
      {
        "id" : <Component Identifier 0>,
        "digest":"<SHA256 of Image>",
        "size" : <Size of Image>
      }
    ],
    "common" : [
      {"condition-vendor-id" : "fa6b4a53-d5ad-5f6d-be9d-e663e4d41ffe"},
      {"condition-class-id" : "1492af14-2569-5e48-bf42-9b2d51f2ab45"}
    ],
    "image-application" : [
      {
        "set-parameter" : {
          "component" : 0
        },
        "source": "http://example.com/file.bin",
      },
      {
        "directive-move" : {
          "destination" : 0
        }
      }
    ],
    "validate" : [
      {
        "condition-validate-image" : {"component" : 0}
      }
    ],
    "image-invocation" : [
      {
        "directive-run-component":{"component" : 0}
      }
    }
  }
]"
{ "structure-version" : 1,  "sequence-number" : 8,  "dependencies" : [ { "digest" : "<SHA256 of Document 0>"  "components" : [<Component Identifier 0>] } ],  "components" : [ { "id" : <Component Identifier 1>,  "digest" : "<SHA256 of Image>",  "size" : <Size of Image> } ],  "dependency-resolution" : [ { "directive-move" : { "source" : "http://example.com/document0.bin",  "destination" : <Document 0 ID> } } ],  { "condition-validate-image" : {"dependency" : 0} },  "image-application" : [ { "directive-move" : { "source" : "http://example.com/file1.bin",  "destination" : 1 } } ],  { "set-parameter" : { "component" : 0  "source" : "http://other-host.com/file.bin", } },  { "process-dependency" : 0 } ]  "validate" : [ { "condition-validate-image" : {"dependency" : 0} } ]}
10. IANA Considerations

In any given serialisation of this approach, several registries will be required for:
- Standard Commands
- Standard Parameters

This document requires no action from IANA.

11. Security Considerations

This document describes the distribution of firmware updates and the invocation of complex behaviours on a device. As such, the contents of a document following the described approach to updates MUST be authenticated as described in Section 7. A more detailed discussion about security can be found in the architecture document [Architecture].

12. References

12.1. Normative References


12.2. Informative References

12.3. URIs

[1] mailto:suit@ietf.org


Appendix A. 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]

Authors’ Addresses

Brendan Moran
ARM Limited
EMail: Brendan.Moran@arm.com

Tony Ibbs
ARM Limited
EMail: Tony.Ibbs@arm.com

George Psimenos
ARM Limited
SUIT CBOR manifest serialisation format
draft-moran-suit-manifest-04

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on September 12, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the
document authors. All rights reserved.

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

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

Table of Contents

1. Introduction .................................................. 3
2. Conventions and Terminology ................................ 4
3. SUIT digest container ....................................... 5
4. Distributing firmware ....................................... 6
5. Workflow of a device applying a firmware update ........... 6
6. SUIT manifest goals ........................................... 6
7. SUIT manifest design overview ............................... 8
    7.1. Severable Elements ...................................... 9
    7.2. Conventions ............................................. 9
    7.3. Payloads ................................................ 10
8. Manifest Structure .......................................... 10
    8.1. Outer wrapper .......................................... 12
    8.2. Manifest ............................................... 13
    8.3. SUIT_Dependency ....................................... 16
    8.4. SUIT_Component ......................................... 17
    8.5. SUIT_Component_Reference ............................... 17
    8.6. Manifest Parameters .................................... 18
        8.6.1. SUIT_Parameter_Strict_Order .................... 20
        8.6.2. SUIT_Parameter_Coerce_Condition_Failure ........ 20
    8.7. SUIT_Parameter_Encryption_Info ....................... 20
    8.8. SUIT_Parameter_Compression_Info ...................... 20
    8.9. SUIT_Parameter_Unpack_Info ............................ 21
    8.10. SUIT_Parameters CDDL ................................ 21
    8.11. SUIT_Command_Sequence ............................... 23
    8.12. SUIT_Condition ........................................ 24
        8.12.1. ID Conditions ................................... 25
        8.12.2. SUIT_Condition_Image_Match .................... 26
        8.12.3. SUIT_Condition_Image_Not_Match ............... 26
        8.12.4. SUIT_Condition_Use_Before ..................... 26
        8.12.5. SUIT_Condition_Minimum_Battery ............... 26
        8.12.6. SUIT_Condition_Update_Authorised ............ 27
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]. This document is structured as follows: In Section 8 we describe the main building blocks of the manifest and Section 12 contains the description of the CBOR of the manifest.

The SUIT manifest is heavily optimised for consumption by constrained devices. This means that it is not constructed as a conventional descriptive document, as described in [Behaviour]. This means that a user viewing the contents of the document will require tooling to view the contents in a more descriptive way.

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.


- Command: A Condition or a Directive.

3. 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. Algorithm identifiers defined in RFC 6920 [RFC6920] are reused for this digest container. This structure is described by the following CDDL:

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

; Named Information Hash Algorithm Identifiers
digest-algorithm-ids /= algorithm-id-sha256
digest-algorithm-ids /= algorithm-id-sha256-128
digest-algorithm-ids /= algorithm-id-sha256-120
digest-algorithm-ids /= algorithm-id-sha256-96
digest-algorithm-ids /= algorithm-id-sha256-64
digest-algorithm-ids /= algorithm-id-sha256-32
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
4. 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].

5. 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.

6. SUIT manifest goals

The manifest described in this document is intended to simplify the construction of constrained device firmware update solutions. It is also intended to allow update authors to describe complex update processes for complex devices.

Manifests implemented as descriptive documents require changes to the parser and the information model whenever a new feature is added. This is particularly accentuated when the parser is a fixed-function minimal parser (or a pull parser) such as the type that is typically
used in a bootloader or in a constrained client. The issue is not as significant in devices that can use general purpose parsers.

The manifest detailed in this document aims to address these and more problems by changing the processing model from a piece of software that loads a manifest, interprets the data, then performs some actions, into a model in which the software performs exactly the operations stated in the manifest, in order. This allows the manifest to encode data in a way that matches precisely with what the parser expects. It also makes inflexible code, like a bootloader, more flexible in what it can do; because the manifest defines part of the "program," the manifest’s execution defines part of the behaviour of the system. Further detail on this approach is covered in [Behaviour]

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 of a firmware at boot.

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

To verify authenticity at boot time, only the smallest portion of the manifest is required. This core part of the manifest describes only the fully installed firmware and any of its dependencies.
7. 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 takes a new approach, encoding the required behaviour of a Recipient device, instead of just presenting the information used to determine that behaviour. This gives benefits equivalent to those provided by a scripting language or 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 secure boot of a firmware image. The language specifies behaviours in a linearised form, without branches or loops. 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 with minimal functionality to perform complex updates with reduced overhead. Conditional execution of commands allows a simple device to perform important decisions at validation-time, such as which differential update to download for a given current version, or which hash to check, based on the installation address.

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 define a component.

Capability reporting is similarly simplified. A Recipient can report the Commands and Parameters 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.

7.1. 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.

7.2. 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.
7.3. 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, elf, and serialised configuration require installation-time unpacking.

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_Payload section or SUIT_Parameter_Image_Digest by indicating a SUIT_Digest algorithm that correctly validates their information.

8. 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. List of dependencies
   3. List of payloads
   4. List of payloads in dependencies
   5. Common list of conditions, directives
   6. Dependency resolution Reference or list of conditions, directives
   7. Payload fetch Reference or list of conditions, directives
   8. Installation Reference or list of conditions, directives
   9. Verification conditions/directives
  10. Load conditions/directives
  11. Run conditions/directives
  12. Text / Reference
  13. 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. Small Payload(s)
8.1. 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               => bstr .cbor 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-external          => bstr .cbor SUIT_Text_Info,
    suit-coswid-external        => bstr .cbor COSWID
}
```

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 null.

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

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 8.2.

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.

8.2. 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-dependencies => [ + SUIT_Dependency ],
    suit-components => [ + SUIT_Component ],
    suit-dependency-components => [ + SUIT_Component_Reference ],
    suit-common => bstr .cbor SUIT_Command_Sequence,
    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-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

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 has been severed.

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 MANDATORY.

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 MANDATORY.

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.

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 include the digest and size of the result. Components affected by a manifest only include the component identifier and the index of the manifest that fully defines the component.

suit-components is a list of SUIT_Component blocks that specify the vital information about the content a component identifier should contain following the update. These are 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 is a SUIT_Command_Sequence to execute prior to executing any other command sequence. Typical actions in suit-common include setting expected device identity and image digests when they are conditional (see Section 11 for more information on conditional sequences). suit-common is OPTIONAL.

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 MANDATORY 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 MANDATORY. 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.

8.3. 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 ‘body_protected’ and ‘body_signed’ 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.

8.4. SUIT_Component

The SUIT_Component describes an image that is uniquely defined by the current manifest. It consists of three elements: the component identifier that represents a component that will be affected by this manifest. This excludes components that are affected by dependencies. The following CDDL describes the SUIT_Component.

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

Because suit-component-size and suit-component-digest can be dependent on installation offset, they cannot be exclusively contained in SUIT_Component. However, since these are security critical parameters, these parameters are updated to match the contents of suit-components prior to processing suit-common. If absent, these are set to Zero and NULL, respectively. This enforces that the manifest defining a component is the only manifest that can describe its contents.

8.5. 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
}

8.6. 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| 1      | Global           | Strict Order      | Requires that the manifest is processed | Requires that the manifest is processed in a strictly linear fashion. Set to 0 to enable parallel handling of manifest directives. |
| 2                 | boolean| 0      | Global           | 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         |

| 4 | bstr | nil | Component/Glob bal | Class ID | A RFC4122 UUID representing the class of the device or component |
| 5 | bstr | nil | Component/Glob bal | Device ID | A RFC4122 UUID representing the device or component |
| 6 | bstr | nil | Component/Dep endency | URI List | A CBOR encoded list of ranked URIs |
| 7 | bstr | nil | Component/Dep endency | Encryption Info | A COSE object defining the encryption mode of the target |
| 8 | bstr | nil | Component | Compression Info | A SUIT_Compression_Info object |
| 9 | bstr | nil | Component | Unpack Info | A SUIT_Unpack_Info object |
| 10 | int/b str | nil | Component | Source Component | A SUIT_Component_Identifier or Component Index |
| 11 | bstr | nil | Component/Dep endency | Image Digest | A SUIT_Digest |
| 12 | bstr | nil | Component/Dep endency | Image Size | Integer size |
| nint | int/b str | nil | Custom | Custom Parameter | Application-defined parameter |
Each parameter contains a Skip/Append flag. Append is an advanced feature that is not available on highly constrained platforms. The mechanism for setting the Append flag is TBD.

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.

8.6.1. 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.

8.6.2. 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.

8.7. SUIT_Parameter_Encryption_Info

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

8.8. 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:

```cddl
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_deflate
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
```

8.9. 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:

```cddl
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
```

8.10. SUIT_Parameters CDDL

The following CDDL describes all SUIT_Parameters.

```
SUIT_Parameters //= SUIT_Parameter_Strict_Order
SUIT_Parameters //= SUIT_Parameter_Coerce_Condition_Failure
```

SUIT_Parameters //= SUIT_Parameter_Vendor_ID
SUIT_Parameters //= SUIT_Parameter_Class_ID
SUIT_Parameters //= SUIT_Parameter_Device_ID
SUIT_Parameters //= SUIT_Parameter_URI_List
SUIT_Parameters //= SUIT_Parameter_Encryption_Info
SUIT_Parameters //= SUIT_Parameter_Compression_Info
SUIT_Parameters //= SUIT_Parameter_Unpack_Info
SUIT_Parameters //= SUIT_Parameter_Source_Component
SUIT_Parameters //= SUIT_Parameter_Image_Digest
SUIT_Parameters //= SUIT_Parameter_Image_Size
SUIT_Parameters //= SUIT_Parameter_Custom

SUIT_Parameter_Strict_Order = (1 => bool)
SUIT_Parameter_Coerce_Condition_Failure = (2 => bool)
SUIT_Parameter_Vendor_ID = (3 => bstr)
SUIT_Parameter_Class_ID = (4 => bstr)
SUIT_Parameter_Device_ID = (5 => bstr)
SUIT_Parameter_URI_List = (6 => bstr .cbor SUIT_URI_List)
SUIT_Parameter_Encryption_Info = (7 => bstr .cbor SUIT_Encryption_Info)
SUIT_Parameter_Compression_Info = (8 => bstr .cbor SUIT_Compression_Info)
SUIT_Parameter_Unpack_Info = (9 => bstr .cbor SUIT_Unpack_Info)
SUIT_Parameter_Source_Component = (10 => bstr .cbor SUIT_Component_Identifier)
SUIT_Parameter_Image_Digest = (11 => bstr .cbor SUIT_Digest)
SUIT_Parameter_Image_Size = (12 => uint)
SUIT_Parameter_Custom = (nint => 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-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_deflate
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

8.11.  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 manifest 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 a new list of conditions/directives that are contained as an argument to the directive. It also contains a flag that indicates whether or not a failure of a condition should indicate a failure of the update/boot.

8.12. SUIT_Condition

Conditions are used to define mandatory properties of a system in order for an update to be applied. They can be pre-conditions or post-conditions of any directive or series of directives, depending on where they are placed in the list. Conditions include:
<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Condition Name</th>
<th>Argument Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Vendor Identifier</td>
<td>RFC4122 UUID wrapped in a bstr</td>
</tr>
<tr>
<td>2</td>
<td>Class Identifier</td>
<td>RFC4122 UUID wrapped in a bstr</td>
</tr>
<tr>
<td>3</td>
<td>Device Identifier</td>
<td>RFC4122 UUID wrapped in a bstr</td>
</tr>
<tr>
<td>4</td>
<td>Image Match</td>
<td>SUIT_Digest</td>
</tr>
<tr>
<td>5</td>
<td>Image Not Match</td>
<td>SUIT_Digest</td>
</tr>
<tr>
<td>6</td>
<td>Use Before</td>
<td>Unsigned Integer timestamp</td>
</tr>
<tr>
<td>7</td>
<td>Minimum Battery</td>
<td>Unsigned Integer</td>
</tr>
<tr>
<td>8</td>
<td>Update Authorised</td>
<td>Integer</td>
</tr>
<tr>
<td>9</td>
<td>Version</td>
<td>List of Integers</td>
</tr>
<tr>
<td>10</td>
<td>Component Offset</td>
<td>Unsigned Integer</td>
</tr>
<tr>
<td>nint</td>
<td>Custom Condition</td>
<td>bstr</td>
</tr>
</tbody>
</table>

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.

8.12.1. ID Conditions

There are three identifier-based conditions:
SUIT_Condition_Vendor_Identifier, SUIT_Condition_Class_Identifier, and SUIT_Condition_Device_Identifier. Each of these conditions present a RFC 4122 [RFC4122] UUID that MUST be matched by the installing device in order to consider the manifest valid.

These conditions MAY be used with or without an argument. If an argument is supplied, then it must be the RFC 4122 [RFC4122] UUID
that must be matched for the condition to succeed. If no argument is supplied, then 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 MANDATORY to implement. SUIT_Condition_Device_Identifier is OPTIONAL to implement.

8.12.2. SUIT_Condition_Image_Match

Verify that the current component matches the supplied digest. If no digest is specified, then the digest is verified 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_Match is MANDATORY to implement.

8.12.3. 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.

8.12.4. 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.

8.12.5. 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_Event 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.
8.12.6. 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.

8.12.7. 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.

The following CDDL describes SUIT_Condition_Version_Argument

SUIT_Condition_Version_Argument = [  
suit-condition-version-comparison: SUIT_Condition_Version_Comparison_Types,  
suit-condition-version-comparison: SUIT_Condition_Version_Comparison_Value  
]

SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Greater
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Greater_Equal
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Equal
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Lesser_Equal
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Lesser

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

SUIT_Condition_Version_Comparison_Value = [+int]

While the exact encoding of versions is application-defined, semantic versions map directly:
- 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.

8.12.8. 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.

8.12.9. 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.

8.12.9.1. 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)

8.12.10. SUIT_Condition CDDL

The following CDDL describes SUIT_Condition:
SUIT_Condition //= (nint => bstr)
SUIT_Condition //= SUIT_Condition_Vendor_Identifier
SUIT_Condition //= SUIT_Condition_Class_Identifier
SUIT_Condition //= SUIT_Condition_Device_Identifier
SUIT_Condition //= SUIT_Condition_Image_Match
SUIT_Condition //= SUIT_Condition_Image_Not_Match
SUIT_Condition //= SUIT_Condition_Use_Before
SUIT_Condition //= SUIT_Condition_Minimum_Battery
SUIT_Condition //= SUIT_Condition_Update_Authorised
SUIT_Condition //= SUIT_Condition_Version
SUIT_Condition //= SUIT_Condition_Component_Offset
SUIT_Condition //= SUIT_Condition_Custom

SUIT_Condition_Vendor_Identifier = (1 => bstr .size 16)
SUIT_Condition_Class_Identifier = (2 => bstr .size 16)
SUIT_Condition_Device_Identifier = (3 => bstr .size 16)
SUIT_Condition_Image_Match = (4 => SUIT_Digest)
SUIT_Condition_Image_Not_Match = (5 => SUIT_Digest)
SUIT_Condition_Use_Before = (6 => uint)
SUIT_Condition_Minimum_Battery = (7 => uint)
SUIT_Condition_Update_Authorised = (8 => int)
SUIT_Condition_Version = (9 => SUIT_Condition_Version_Argument)
SUIT_Condition_Component_Offset = (10 => uint)
SUIT_Condition_Custom = (nint => bstr)

SUIT_Condition_Version_Argument = [
    suit-condition-version-comparison: SUIT_Condition_Version_Comparison_Types,
    suit-condition-version-comparison: SUIT_Condition_Version_Comparison_Value
]
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Greater
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Greater_Equal
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Equal
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Lesser_Equal
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Lesser

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

SUIT_Condition_Version_Comparison_Value = [+int]

8.13. SUIT_Directive

Directives are used to define the behaviour of the recipient.
Directives include:
<table>
<thead>
<tr>
<th>Directive Code</th>
<th>Directive Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Set Component Index</td>
</tr>
<tr>
<td>12</td>
<td>Set Manifest Index</td>
</tr>
<tr>
<td>13</td>
<td>Run Sequence</td>
</tr>
<tr>
<td>14</td>
<td>Run Sequence Conditional</td>
</tr>
<tr>
<td>15</td>
<td>Process Dependency</td>
</tr>
<tr>
<td>16</td>
<td>Set Parameters</td>
</tr>
<tr>
<td>17</td>
<td>Reserved</td>
</tr>
<tr>
<td>18</td>
<td>Reserved</td>
</tr>
<tr>
<td>19</td>
<td>Override Parameters</td>
</tr>
<tr>
<td>20</td>
<td>Fetch</td>
</tr>
<tr>
<td>21</td>
<td>Copy</td>
</tr>
<tr>
<td>22</td>
<td>Run</td>
</tr>
<tr>
<td>23</td>
<td>Wait</td>
</tr>
</tbody>
</table>

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

8.13.1. 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_Manifest_Index is used, SUIT_Directive_Set_Component_Index = False is implied. When SUIT_Directive_Set_Component_Index is used, SUIT_Directive_Set_Manifest_Index = False is implied.
The following CDDL describes the argument to
SUIT_Directive_Set_Component_Index.

SUIT_Directive_Set_Component_Index_Argument = uint/bool

8.13.2.  SUIT_Directive_Set_Manifest_Index

Set Manifest 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_Manifest_Index = False is implied. When
SUIT_Directive_Set_Manifest_Index is used,
SUIT_Directive_Set_Component_Index = False is implied.

Typical operations that require SUIT_Directive_Set_Manifest_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_Manifest_Index.

SUIT_Directive_Set_Manifest_Index_Argument = uint/bool

8.13.3.  SUIT_Directive_Run_Sequence

To enable conditional commands, and to allow several strictly ordered
sequences to be executed out-of-order, SUIT_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.
8.13.4. SUIT_Directive_Run_Sequence_Conditional

This command is exactly the same as SUIT_Directive_Run_Sequence, except that it initialises Coerce on Condition Failure to True.

SUIT_Parameter_Coerce_Condition_Failure defaults to True when SUIT_Directive_Run_Sequence_Conditional begins. Its value is discarded when SUIT_Directive_Run_Sequence_Conditional terminates.

8.13.5. 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

8.13.6. 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 8.6.

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.

SUITE_Directive_Set_Parameters_Argument = {+ SUITE_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.

8.13.7. SUITE_Directive_Set_Parameter_State_Append

This command is reserved for future use. It will provide a mechanism
to override the "set if unset" logic of SUITE_Directive_Set_Parameters
on a per-parameter basis. This will allow certain parameters to be
treated as lists, rather than fixed values. This enables a feature
for an advanced device to fail over from URIs defined in one manifest
to those defined in another.

8.13.8. SUITE_Directive_Override_Parameters

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

Available parameters are defined in Section 8.6.

The argument to SUITE_Directive_Override_Parameters is defined in the
following CDDL.

SUITE_Directive_Override_Parameters_Argument = {+ SUITE_Parameters}

8.13.9. SUITE_Directive_Fetch

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

SUITE_Directive_Fetch can target one or more manifests and one or more
payloads. SUITE_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 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

8.13.10. 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

8.13.11. 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


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_Directive_Wait_Argument = {
    SUIT_Wait_Events
}
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 00:00:00)
SUIT_Wait_Event_Argument_Day_Of_Week = uint ; Days since Sunday

8.13.13. SUIT_Directive CDDL

    The following CDDL describes SUIT_Directive:
SUIT_Directive // SUIT_Directive_Set_Component_Index
SUIT_Directive // SUIT_Directive_Set_Manifest_Index
SUIT_Directive // SUIT_Directive_Run_Sequence
SUIT_Directive // SUIT_Directive_Run_Sequence_Conditional
SUIT_Directive // SUIT_Directive_Run

SUIT_Directive_Set_Component_Index = (11 => uint/bool)
SUIT_Directive_Set_Manifest_Index = (12 => uint/bool)
SUIT_Directive_Run_Sequence = (13 => bstr .cbor SUIT_Command_Sequence)
SUIT_Directive_Run_Sequence_Conditional = (14 => bstr .cbor SUIT_Command_Sequenc e)
SUIT_Directive_Process_Dependency = (15 => nil)
SUIT_Directive.Override_Parameters = (16 => {+ SUIT_Parameters})
SUIT_Directive_Fetch = (20 => nil/bstr)
SUIT_Directive_Copy = (21 => nil/bstr)
SUIT_Directive_Run = (22 => nil/bstr)
SUIT_Directive_Wait = (23 => { + SUIT_Wait_Events })

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
9. 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.
10. 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.

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,  
  "components": [  
    {  
      "component-identifier": [0],  
      "component-size": [32567],  
    },  
    ],  
  "common": [  
    "set-component-index": 0,  
    "do-sequence": [  
      "condition-component-offset": "<offset A>",  
      "set-parameters": {  
        "component-digest": "<SHA256 A>"  
      },  
    ],  
    "do-sequence": [  
      "condition-component-offset": "<offset B>",  
      "set-parameters": {  
        "component-digest": "<SHA256 A>"  
      },  
    ],  
    ],  
  "fetch": [  
    "set-component-index": 0,  
    "do-sequence": [  
      "condition-component-offset": "<offset A>",  
      "set-parameters": {  
        "uri-list": [[0, "<uri-A>"]]  
      },  
    ],  
    "do-sequence": [  
      "condition-component-offset": "<offset B>",  
      "set-parameters": {  
        "uri-list": [[0, "<uri-B>"]]  
      },  
    ],  
    "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_Sequence,
? suit-payload-fetch => SUIT_Digest / bstr .cbor SUIT_Command_Sequence,
? suit-install => SUIT_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 => SUIT_Digest / bstr .cbor SUIT_Text_Map,
? suit-coswid => SUIT_Digest / bstr .cbor concise-software-identity
}

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 //= (1 => RFC4122_UUID) ; SUIT_Condition_Vendor_Identifier
SUIT_Condition //= (2 => RFC4122_UUID) ; SUIT_Condition_Class_Identifier
SUIT_Condition //= (3 => RFC4122_UUID) ; SUIT_Condition_Device_Identifier
SUIT_Condition //= (4 => SUIT_Digest) ; SUIT_Condition_Image_Match
SUIT_Condition //= (5 => SUIT_Digest) ; SUIT_Condition_Image_Not_Match
SUIT_Condition //= (6 => uint) ; SUIT_Condition_Use_Before
SUIT_Condition //= (7 => uint) ; SUIT_Condition_Minimum_Battery
SUIT_Condition //= (8 => int) ; SUIT_Condition_Update_Authorised
SUIT_Condition //= (9 => SUIT_Condition_Version_Argument) ; SUIT_Condition_Versi
SUIT_Condition //= (10 => uint) ; SUIT_Condition_Component_Offset
SUIT_Condition //= (nint => bstr) ; SUIT_Condition_Custom

RFC4122_UUID = bstr .size 16

SUIT_Condition_Version_Argument = [
]
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Gre
ter
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Gre
ter_Equal
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Equ
al
SUIT_Condition_Version_Comparison_Types /= SUIT_Condition_Version_Comparison_Les
er_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
SUIT_Directive //= (14 => bstr .cbor SUIT_Command_Sequence) ; SUIT_Directive_Run
SUIT_Directive //= (16 => [+ SUIT_Parameters]) ; SUIT_Directive_Set_Parameters
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 00:00:00)
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_Encryption_Info
SUIT_Parameters //= (8 => bstr .cbor SUIT_Compression_Info) ; SUIT_Parameter_Compression_Info
SUIT_Parameters //= (9 => bstr .cbor SUIT_Unpack_Info) ; SUIT_Parameter_Unpack_Info
SUIT_Parameters //= (10 => bstr .cbor SUIT_Component_Identifier) ; SUIT_Parameter_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.

```json
{
    "structure-version": 1,
    "sequence-number": 1,
    "components": [
        {
            "id": ["Flash", 78848],
            "digest": "00112233445566778899aabbccddeeff"
            "0123456789abcdefedcba9876543210",
            "size": 34768
        }
    ],
    "run-image": [
        {"directive-set-component": 0},
        {"condition-image": null},
        {"directive-run": null}
    ]
}
```

Converted into the SUIT manifest, this produces:
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,
    "components": [
        {
            "id": ["Flash", 78848],
            "digest": "00112233445566778899aabbccddeeff"
            "0123456789abcdefedcba9876543210",
            "size": 34768
        }
    ],
    "apply-image": [
        {"directive-set-component": 0},
        {"directive-set-var": {
            "uris": [0, "http://example.com/file.bin"]
        }},
        {"directive-fetch": null}
    ]
}

Converted into the SUIT manifest, this produces:

```
Example 1:

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

The following JSON shows the intended behaviour of the manifest.

```
{
  / auth object / 1 : None
  / manifest / 2 : h'a4010102020481a3018245466c61736843003401021987'
    h'd00382015820001122334455667788899aabbccddeeff0123456789abcdef'
    h'fedcba987654321009582d83a1b0b0a110a106582081200781b68747470'
    h'3a2f2f6578616d76c652e63f66f6696c652e62696ea114f6'
  }
  / structure-version / 1 : 1
  / sequence-number / 2 : 2
  / components / 4 : [
    / component-identifier / 1 : [h'466c617368', h'003401'],
    / component-size / 3 : 34768
    / component-digest / 2 : [
      / sha-256 / 1,
      h'0011223344566778899aabbccddeeff'
      h'0123456789abcdef'
    ],
  ],
  / apply-image / 9 : [
    / set-component-index / 11 : 0
    / set-vars / 16 : {
      / uris / 6 : h'818200781b687474703a2f2f6578616d706c'
       h'652e63f66f6696c652e62696ea114f6'
      [[0, 'http://example.com/file.bin']] /
    },
  ],
  / fetch / 20 : None
}
```

Total size of outer wrapper without COSE authentication object: 115

Outer:

```
/3.3. 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,
    "components": [
        {
            "id": [
                "Flash",
                78848
            ],
            "digest": "00112233445566778899aabbccddeeff"
                "0123456789abcdefedcba9876543210",
            "size": 34768
        }
    ],
    "common": [
        {  
            "condition-vendor-id": "fa6b4a53-d5ad-5fd5-be9d-e663e4d41f6e",
            "condition-class-id": "1492af14-2569-5e48-bf42-9b2d5f2ab45"
        },
        "apply-image": [
            {  
                "directive-set-component": 0,
                "directive-set-var": {  
                    "uris": [
                        "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 / 2 : h'a6010102030481a3018245466c61736843003401021987'
    h'd0038201582000112233445566778899aabcccddeeff0123456789abcdeff'
    h'fedcba987654321006582782a10150fa6b4a53d5ad5dfdfbe9de663e4d41f'
    h'fea102501492af1425695e48bf429b2d51f2ab4509582d83a10b00a110a1'
    h'065820818200781b687474703a2f2f6578616d706c652e636f66696c'
    h'652e62696ea114f60c4a83a10b00a104f6ea11f6f6'
  },
  / structure-version / 1 : 1
  / sequence-number / 2 : 3
  / components / 4 : [ 
    {  
      / component-identifier / 1 : [h'466c617368', h'003401'],
      / component-size / 3 : 34768
      / component-digest / 2 : [ 
        / sha-256 / 1,
        h'0011233445566778899aabcccddeeff'
        h'0123456789abcdeffedcba9876543210'
      ],
    },
  ],
  / common / 6 : [ 
    {  
      / vendor-id / 1 : h'fa6b4a53d5ad5dfdfbe9de663e4d41ffe'
        fa6b4a53-d5ad-5dfd-be9d-e663e4d41ffe},
    {  
      / class-id / 2 : h'1492af1425695e48bf429b2d51f2ab45'
        1492af14-2569-5e48-bf42-9b2d51f2ab45}
  ],
  / apply-image / 9 : [  
    {  
      / set-component-index / 11 : 0
      / set-vars / 16 : {  
        / uris / 6 : h'818200781b687474703a2f2f6578616d706c65'
          h'2e636f66696c652e62696e'
          [0, 'http://example.com/file.bin']] /  
    },
    {  
      / fetch / 20 : None}
  ],
  / run-image / 12 : [  
    {  
      / set-component-index / 11 : 0
      / condition-image / 4 : None
      / run / 22 : None}
  ],
}

Total size of outer wrapper without COSE authentication object: 169

Outer:

Moran, et al. Expires September 12, 2019
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.
Internet-Draft          Firmware Manifest Format              March 2019

{
  "structure-version": 1,
  "sequence-number": 4,
  "components": [
    {
      "id": ["Flash", 78848],
      "digest": "00112233445566778899aabbccddeeff",
      "0123456789abcdefecb9876543210",
      "size": 34768
    },
    {
      "id": ["RAM", 1024],
      "digest": "00112233445566778899aabbccddeeff",
      "0123456789abcdefecb9876543210",
      "size": 34768
    }
  ],
  "common": [
    {"condition-vendor-id": "fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe"},
    {"condition-class-id": "1492af14-2569-5e48-bf42-9b2d51f2ab45"}
  ],
  "apply-image": [
    {"directive-set-component": 0},
    {"directive-set-var": {
      "uris": [[0, "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 / 2 : h\'a6010102040482a3018245466c61736843003401021987''
  h\'d003820158200011223445566778899aabbccddeeff0123456789abcdef'
  h\'fedcba9876543210a301824352414d420004021987d00382015820001122'
}

h'33445566778899aabbccddeeff0123456789abcdeffedcba987654321006'
h'582782a10150fa6b4a53d5ad5dfbe9de663e4d41ffe102501492af1425'
h'695e48bf429b2d51f2ab4509582d83a10b00a110a105820818200781b68'
h'7474703a2f2f6578616d706c652e636f6d2f66696c652e62696e114f60c'
h'581887a10b00a104f6a10b01a110a10a00a114f6a104f6a116f6' 
{
  / structure-version / 1 : 1
  / sequence-number / 2 : 4
  / components / 4 : [
    {
      / component-identifier / 1 : [h'466c617368', h'003401'],
      / component-size / 3 : 34768
      / component-digest / 2 : [
        / sha-256 / 1,
        h'00112233445566778899aabbccddeeff'
        h'0123456789abcdeffedcba9876543210'
      ],
    },
    {
      / component-identifier / 1 : [h'52414d', h'0004'],
      / component-size / 3 : 34768
      / component-digest / 2 : [
        / sha-256 / 1,
        h'00112233445566778899aabbccddeeff'
        h'0123456789abcdeffedcba9876543210'
      ],
    }
  ],
  / common / 6 : [
    / vendor-id / 1 : h'fa6b4a53d5ad5dfbe9de663e4d41ffe' \ 
    fa6b4a53-d5ad-5dfb-be9d-e663e4d41ffe)
    / class-id / 2 : h'1492af1425695e48bf429b2d51f2ab45' \ 
    1492af14-2569-5e48-bf42-9b2d51f2ab45)
  ],
  / apply-image / 9 : [
    / set-component-index / 11 : 0
    / set-vars / 16 : {
      / uris / 6 : h'818200781b687474703a2f2f6578616d706c65' \ 
        h'2e636f6d2f66696c652e62696e' /
        [[0, 'http://example.com/file.bin']]
    },
    / fetch / 20 : None
  ],
  / run-image / 12 : [
    / set-component-index / 11 : 0
    / condition-image / 4 : None
    / set-component-index / 11 : 1
    / set-vars / 16 : {

Total size of outer wrapper without COSE authentication object: 235

Outer:

a201f60258e5a6010102040482a3018245466c6173684300340121987d00382015820
00112334455677899aabbcccddeeff0123456789abcdefedcbaf87654310a30182
4352414d42004021987d0038201582000112334556677899aabbcccddeeff012345
6789abcdefedcbaf87654321006582782a10150fa6b4a53d5d5f5f6b99de663e4d41f
feaa02501492af1425695e48bf429b2d51f2ab4509582d83a10b00a110a1065820812
00781b687474703a2f2f6578616d706c652e636f6d2f66696c652e62696ea14f60c58
1487a00a104f6a10b01a110a10a0a114f6a104f6a116f6

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,  
  "components": [  
    {  
      "id": ["Flash", 78848],  
      "digest": "00112233445566778899aabbccddeeff"  
                    "0123456789abcdefdcba9876543210",  
      "size": 34768  
    },  
    {  
      "id": ["RAM", 1024],  
      "digest": "0123456789abcdefdcba9876543210"  
                    "00112233445566778899aabbccddeeff",  
      "size": 34768  
    }  
  ],  
  "common": [  
    {"condition-vendor-id": "fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe"},  
    {"condition-class-id": "1492af14-2569-5e48-bf42-9b2d51f2ab45"}  
  ],  
  "apply-image": [  
    {"directive-set-component": 0},  
    {"directive-set-var": {  
      "uris": [0, "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 / 2 : h’a701012050482a3018245466c61736843003401021987’
  h’d0038201582000112233445566778899aabbccddeeff0123456789abdef’
  h’fedcba9876543210a301824352414d420004021987d00382015820012345’
  h’6789abcedffedcb987654321000112233445566778899aabbccddeeff06’
  h’582782a10150fa6b4a53d5ad5fdfe99de6634d4ffea102501492af1425’
  h’695e48bf429b2d51f2ab4509582d83a10b00a110a1065820818200781b68’
  h’7474703a2f2f6578616d706c652e63f6d2f66696c652e62696ea114f60b’
  h’5585a10b00a104f6a10b01a110a20841f60a00a115f60c4782a104f6a116’
  h’f6’ 
}  

/ structure-version / 1 : 1  
/ sequence-number / 2 : 5  
/ components / 4 : [  
  
/ component-identifier / 1 : [h’466c617368’, h’003401’],  
/ component-size / 3 : 34768  
/ component-digest / 2 : [  
  / sha-256 / 1,  
    h’00112233445566778899aabbccddeeff’  
    h’0123456789abcedffedcb9876543210’  
  ],  
},  

/ component-identifier / 1 : [h’52414d’, h’0004’],  
/ component-size / 3 : 34768  
/ component-digest / 2 : [  
  / sha-256 / 1,  
    h’0123456789abcedffedcb9876543210’  
    h’00112233445566778899aabbccddeeff’  
  ],  
},  

/ common / 6 : [  
  / vendor-id / 1 : h’fa6b4a53d5ad5fdfe9de663e4d41ffe’ \  
    fa6b4a53-d5ad-5fdfe-be9de663e4d41ffe],  
  / class-id / 2 : h’1492af1425695e48bf429b2d51f2ab45’ \  
    1492af14-2569-5e48-bf42-9b2d51f2ab45]  
},  

/ apply-image / 9 : [  
  / set-component-index / 11 : 0  
  / set-vars / 16 : [  
    / uris / 6 : h’818200781b687474703a2f2f6578616d706c65’ \  
      h’e636f6d2f66696c652e62696e’ \  
      [0, ‘http://example.com/file.bin’]  
  ]  
},  

/ fetch / 20 : None}
13.6. Example 5:

Compatibility test, download, installation, and secure boot.

The following JSON shows the intended behaviour of the manifest.

```
],
/ load-image / 11 : [
  {/ set-component-index / 11 : 0}
  {/ condition-image / 4 : None}
  {/ set-component-index / 11 : 1}
  {/ set-vars / 16 : {
    / unknown / 8 : b’\xf6’
    / source-component / 10 : 0
  }},
  {/ copy / 21 : None}
],
/ run-image / 12 : [
  {/ condition-image / 4 : None}
  {/ run / 22 : None}
],
}

Total size of outer wrapper without COSE authentication object: 240

Outer:
a201f60258ea7010102050482a3018245466c61736843003401021987d00382015820
00112233445566778899aabbccddeeff0123456789abcdefedcba9876543210a30182
4352414d420004021987d003820158200123456789abcdefedcba9876543210001122
33445566778899aabbccddeeff06582782a10150fa6b4a53d5ad5fdfbe9de663e4d41f
fea102501492af1425695e48bf429b2d51f2ab4509582d83a10b00a110a10658208182
00781b687474703a2f2f6578616d706c652e636f6d2f6696c652e62696ea114f60b55
85a10b00a104f6a10b01a110a20841f60a00a115f60c4782a104f6a116f6
```
{
  "structure-version": 1,
  "sequence-number": 6,
  "components": [
    {
      "id": ["ext-Flash", 78848],
      "digest": "00112233445566778899aabbccddeeff"
        "0123456789abcdefedcb9876543210",
      "size": 34768
    },
    {
      "id": ["Flash", 1024],
      "digest": "0123456789abcdefedcb9876543210"
        "00112233445566778899aabbccddeeff",
      "size": 34768
    }
  ],
  "common": [
    {"condition-vendor-id": "fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe"},
    {"condition-class-id": "1492af14-2569-5e48-bf42-9b2d51f2ab45"}
  ],
  "apply-image": [
    {"directive-set-component": 0},
    {"directive-set-var": {
      "uris": [[0, "http://example.com/file.bin"]]
    }},
    {"directive-fetch": null}
  ],
  "load-image": [
    {"directive-run-conditional": [
      {"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:

```json
{
    /auth object / 1 : None
    /manifest / 2 : h'a7010102060482a30182496578742d466c617368430034' \
        h'01021987d0038201582004112234455566778899aabccddeeff01234567' \
        h'89abcdeffedcbab976543210a3018245466c617368420004201987d00382' \
        h'0158200123456789abcdeffedcbab98765432100112234455566778899aa' \
        h'bbccddeeff06582072820010150fa6b4a53d5ad5fdbe9de663e4d1ffe0a102' \
        h'501492af1425695e48bf429b2d51f2ab4509582d83a10b00a110a1065820' \
        h'818200781b687474703a2f2f6578616d706c652e636f6d2f6696c652e62' \
        h'696ea114f60b581d81a0e581887a10b01a105f6a10b01a104f6a10b01a1' \
        h'10a10a00a114f60c4a8a10b01a104f6a116f6' \
        
        /structure-version / 1 : 1
        /sequence-number / 2 : 6
        /components / 4 : [
            /component-identifier / 1 : [
                h'6578742d466c617368',
                h'003401'
            ],
            /component-size / 3 : 34768
            /component-digest / 2 : [
                /sha-256 / 1,
                h'00112233445566778899aabccddeeff'
            ],
            h'0123456789abcdeffedcba9876543210'
        ],
        
        /component-identifier / 1 : [h'466c617368', h'0004'],
        /component-size / 3 : 34768
        /component-digest / 2 : [
            /sha-256 / 1,
            h'0123456789abcdeffedcba9876543210'
        ],
        h'00112233445566778899aabccddeeff'
    ],

    /common / 6 : [
        /vendor-id / 1 : h'fa6b4a53d5ad5fdbe9de663e4d1ffe' \
            fa6b4a53-d5ad-5fdbe-9de663e4d1ffe
        /class-id / 2 : h'1492af1425695e48bf429b2d51f2ab45' \
            1492af14-2569-5e48-bf42-9b2d51f2ab45
    ],

    /apply-image / 9 : [
        /set-component-index / 11 : 0
        /set-vars / 16 : [
```
Example 6:
Compatibility test, 2 images, simultaneous download and installation, and secure boot.

The following JSON shows the intended behaviour of the manifest.
Converted into the SUIT manifest, this produces:

```json
{
  "auth object / 1 : None
  "manifest / 2 : h’a6010102070482a3018245466c61736843003401021987’
    h’d0038201580200011233445566778899aabbccddeeff0123456789abcdef’
    h’fedcba9876543210a3018245466c6173684300402021a00012c22038201’
}
```
h'58200123456789abcdeffedcba98765432100112233445566778899aab'
h'ccddeeff06582782a10150fa6b4a53d5ad5fdfbe9de663e4d41ffe10250' h'1492af1425695e48bf429b2d51f2ab4509585b86a10b00a110a106582181' h'8200781c687474703a2f2f6578616d706c652e636f6d2f66696c65312e62' h'696ea10b01a110a1065821818200781c687474703a2f2f6578616d706c65' h'2e636f6d2f66696c6532e62696ea10bf5a114f60c4d8a10bf5a104f6a1' h'0b00a116f6' 
} / structure-version / 1 : 1 
/ sequence-number / 2 : 7 
/ components / 4 : [ 
  / component-identifier / 1 : [h'466c617368', h'003401'], 
  / component-size / 3 : 34768 
  / component-digest / 2 : [ 
    / sha-256 / 1, 
    h'00112233445566778899aabbccddeeff' 
    h'0123456789abcdeffedcba9876543210'
  ], 
}

/ component-identifier / 1 : [h'466c617368', h'000402'], 
/ component-size / 3 : 76834 
/ component-digest / 2 : [ 
  / sha-256 / 1, 
  h'0123456789abcdeffedcba9876543210'
  h'00112233445566778899aabbccddeeff'
],

/ common / 6 : [ 
  / vendor-id / 1 : h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' 
  fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe)
  / class-id / 2 : h'1492af1425695e48bf429b2d51f2ab45' 
  1492af14-2569-5e48-bf42-9b2d51f2ab45)
], 
/ apply-image / 9 : [ 
  / set-component-index / 11 : 0 
  / set-vars / 16 : [ 
    / uris / 6 : h'818200781c687474703a2f2f6578616d706c' 
    h'652e636f6d2f66696c65312e62696ea10bf5a114f60c4d8a10bf5a104f6a1'
    h'0b00a116f6' 
    ]
  ]
},

/ set-component-index / 11 : 1 
/ set-vars / 16 : [ 
  / uris / 6 : h'818200781c687474703a2f2f6578616d706c' 
  h'652e636f6d2f66696c6532e62696ee10bf5a114f60c4d8a10bf5a104f6a1'
  h'0b00a116f6' 
  ]
],

/ set-component-index / 11 : 1 
/ set-vars / 16 : [ 
  / uris / 6 : h'818200781c687474703a2f2f6578616d706c' 
  h'652e636f6d2f66696c6532e62696ae10bf5a114f60c4d8a10bf5a104f6a1'
  h'0b00a116f6' 
  ]
Total size of outer wrapper without COSE authentication object: 275

Outer:
a201f60259010ca6010102070482a3018245466c61736843003401021987d003820158
2000112233445566778899aabcdddeeff0123456789abcdefdcba9876543210a301
8245466c6173684300040202a000012012233445566778899aabcdddeeff0658287201a0150f6b4a53d53ad5fdbe
9de663e4d41f0e102501492af1425695e48bf429b2d51f2ab450958586a10b00a110
a1065821818200781c687474703a2f2f6578616d706c652e63f6d2f6669c65312e62
696ea10b01a110a1065821818200781c687474703a2f2f6578616d706c652e63f6d2f
6696c65322e62696ea10bf5a114f60c4d8a10bf5a10f66a10b00a116f6

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 Chrusciel
- Andrzej Puzdrowski
- Michael Richardson
- David Brown
- Emmanuel Baccelli

18. References

18.1. Normative References
18.2. Informative References

[Architecture]

[Behaviour]

[Information]


18.3. URIs

[1] mailto:suit@ietf.org

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