Workgroup: SUIT
Internet-Draft: draft-ietf-suit-manifest-18
Published: 11 July 2022
Intended Status: Standards Track
Expires: 12 January 2023
Authors: B. Moran, Arm Limited
H. Tschofenig, Arm Limited
H. Birkholz, Fraunhofer SIT
K. Zandberg, Inria

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

Abstract

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

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

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

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

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

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

* Meet the requirements defined in [RFC9124].
* Simple to parse on a constrained node
* Simple to process on a constrained node
* Compact encoding
* Comprehensible by an intermediate system
* Expressive enough to enable advanced use cases on advanced nodes
* Extensible

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

* a Firmware Author to reason about releasing a firmware.
* a Network Operator to reason about compatibility of a firmware.
* a Device Operator to reason about the impact of a firmware.

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

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

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

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

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

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

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

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

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

* Copy some data from one place to another

* Transform some data

* Digest some data and compare to an expected value

* Compare some system parameters to an expected value

* Run some code

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

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

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

2. Conventions and Terminology

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

Additionally, the following terminology is used throughout this document:

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

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

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

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

* Envelope: A container with the manifest, an authentication wrapper with cryptographic information protecting the manifest, authorization information, and severable elements.

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

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

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

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

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

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

*Command: A Condition or a Directive.

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

*Directive: An action for the Recipient to perform.

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

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

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

*Report: A list of Records.

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

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

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

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

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

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

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

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

3. How to use this Document

This specification covers five aspects of firmware update:

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

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

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

*Section 7 describes the process of creating a Manifest.

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

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

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

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

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

*Firmware Encryption is covered in [I-D.ietf-suit-firmware-encryption]

*Update Management is covered in [I-D.ietf-suit-update-management]

*Features, such as dependencies, key delegation, multiple processors, required by the use of multiple trust domains are covered in [I-D.ietf-suit-trust-domains]

*Secure reporting of the update status is covered in [I-D.ietf-suit-report]

* Compression of firmware images

4. Background

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

4.1. IoT Firmware Update Constraints

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

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

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

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

* small internal storage may require external storage support.

* multiple microcontrollers may require coordinated update of all applications.

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

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

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

4.2. SUIT Workflow Model

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

* Compatibility must be checked before any other operation is performed.

* In some applications, payloads must be fetched and validated prior to installation.

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

* Compatibility must be checked before any other operation is performed.

* All payloads must be validated prior to loading.

* All loaded images must be validated prior to execution.

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

1. Verify the signature of the manifest.

2. Verify the applicability of the manifest.

3. Fetch payload(s).

4. Install payload(s).

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

1. Verify image(s).

2. Load image(s).

3. Run image(s).

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

5. Metadata Structure Overview

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

The manifest is structured from several key components:

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

2. The Authentication Block (see Section 5.2) contains a list of signatures or MACs of the manifest..

3. The Manifest (see Section 5.3) contains all critical, non-severable metadata that the Recipient requires. It is further broken down into:

    1. Critical metadata, such as sequence number.

    2. Common metadata, such as affected components.

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

    4. Integrity check values for severable elements.

4. Severable elements (see Section 5.4).

5. Integrated payloads (see Section 5.5).

The diagram below illustrates the hierarchy of the Envelope.

```
+-------------------------+
| Envelope                |
+-------------------------+
| Authentication Block    |
| Manifest           --------------> +------------------------------+
| Severable Elements      |          | Manifest                     |
| Human-Readable Text     |          +------------------------------+
| Integrated Payloads     |          | Structure Version            |
+-------------------------+          | Sequence Number              |
                                     | Reference to Full Manifest   |
                               +------ Common Structure             |
                               | +---- Command Sequences            |
+-------------------------+    | |   | Digests of Envelope Elements |
| Common Structure        | <--+ |   +------------------------------+
+-------------------------+      |
| Components IDs          |      +-> +-----------------------+
| Common Command Sequence ---------> | Command Sequence      |
+-------------------------+          +-----------------------+
                                     | List of ( pairs of (  |
                                     |   * command code      |
                                     |   * argument /        |
                                     |      reporting policy |
                                     | ))                    |
                                     +-----------------------+
```

5.1. Envelope

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

See Section 8.2 for more detail.

5.2. Authentication Block

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

*COSE_Sign_Tagged

*COSE_Sign1_Tagged

*COSE_Mac_Tagged

*COSE_Mac0_Tagged

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

See Section 8.3 for more detail.

5.3. Manifest

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

See Section 8.4 for more detail.

5.3.1. Critical Metadata

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

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

5.3.2. Common

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

See Section 8.4.5 for more detail.

5.3.3. Command Sequences

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

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

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

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

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

See Section 8.4.6 for more detail.

5.3.4. Integrity Check Values

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

See Section 8.4.11 for more detail.

5.3.5. Human-Readable Text

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

See Section 8.4.4 for more detail.

5.4. Severable Elements

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

Because of this organisation, these elements can be discarded or "Severed" from the Envelope without changing the signature of the

Manifest. This allows savings based on the size of the Envelope in several scenarios, for example:

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

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

See Section 8.5 for more detail.

5.5. Integrated Payloads

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

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

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

See Section 7.5 for more detail.

6. Manifest Processor Behavior

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

6.1. Manifest Processor Setup

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

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

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

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

* Test failed (e.g. Vendor ID/Class ID).

* Unsupported command encountered.

* Unsupported parameter encountered.

* Unsupported Component Identifier encountered.

* Payload not available.

* Application crashed when executed.

* Watchdog timeout occurred.

* Payload verification failed.

* Missing required component from a Component Set.

* Required parameter not supplied.

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

Selecting an older manifest in the event of failure of the latest valid manifest is a robustness mechanism that is necessary for supporting the requirements in [RFC9019], section 3.5. It may not be appropriate for all applications. In particular Trusted Execution Environments MAY require a failure to invoke a new installation, rather than a rollback approach. See [RFC9124], Section 4.2.1 for more discussion on the security considerations that apply to rollback.

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

6.2. Required Checks

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

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

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

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

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

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

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

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

1. has sufficient permissions imparted by its signatures

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

6.2.1. Minimizing Signature Verifications

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

* The Authentication Block contains a COSE_Sign_Tagged or COSE_Sign1_Tagged

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

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

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

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

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

6.3. Interpreter Fundamental Properties

The interpreter has a small set of design goals:

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

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

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

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

6.3.1. Resilience to Disruption

As required in Section 3 of RFC9019 and as an extension of design goal 1, devices must remain operable after a disruption, such as a power failure or network interruption, interrupts the update process.

The manifest processor must be resilient to these faults. In order to enable this resilience, systems implementing the manifest processor MUST make the following guarantees:

Either: 1. A fallback/recovery image is provided so that a disrupted system can apply the SUIT Manifest again. 2. Manifests are constructed so that repeated partial invocations of any manifest sequence always results in a correct system configuration. 3. A journal of manifest operations is stored in nonvolatile memory so that a repeated invocation does not alter nonvolatile memory up until the point of the previous failure. The journal enables the parser to recreate the processor state just prior to the disruption. This journal can be, for example, a SUIT Report. This report can be used to resume processing of the manifest from the point of failure.

AND

1. Where a command is not repeatable because of the way in which it alters system state (e.g. swapping images or in-place delta) it MUST be resumable or revertible. This applies to commands that modify at least one source component as well as the destination component.

6.4. Abstract Machine Description

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

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

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

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

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

current := components\[component-index\]

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

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

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

Table 1

6.5. Special Cases of Component Index

Component Index can take on one of three types:

1. Integer

2. Array of integers

3. True

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

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

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

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

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

6.6. Serialized Processing Interpreter

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

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

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

6.7. Parallel Processing Interpreter

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

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

*Set Parameters.

*Override Parameters.

*Set Strict Order = True.

*Set Component Index.

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

7. Creating Manifests

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

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

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

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

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

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

7.1. Compatibility Check Template

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

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

* Set Component Index directive (see Section 8.4.10.1)

* Override Parameters directive (see Section 8.4.10.3) for Vendor ID and Class ID (see Section 8.4.8)

* Check Vendor Identifier condition (see Section 8.4.8.2)

* Check Class Identifier condition (see Section 8.4.8.2)

7.2. Trusted Invocation Template

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

The following commands are placed into the common sequence:

* Set Component Index directive (see Section 8.4.10.1)

*Override Parameters directive (see Section 8.4.10.3) for Image Digest and Image Size (see Section 8.4.8)

The system validation sequence contains the following commands:

*Set Component Index directive (see Section 8.4.10.1)

*Check Image Match condition (see Section 8.4.9.2)

Then, the run sequence contains the following commands:

*Set Component Index directive (see Section 8.4.10.1)

*Run directive (see Section 8.4.10.7)

7.3. Component Download Template

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

The following commands are placed into the common sequence:

*Set Component Index directive (see Section 8.4.10.1)

*Override Parameters directive (see Section 8.4.10.3) for Image Digest and Image Size (see Section 8.4.8)

Then, the install sequence contains the following commands:

*Set Component Index directive (see Section 8.4.10.1)

*Override Parameters directive (see Section 8.4.10.3) for URI (see Section 8.4.8.9)

*Fetch directive (see Section 8.4.10.4)

*Check Image Match condition (see Section 8.4.9.2)

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

7.4. Install Template

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

This template is typically used with the Component Download template, however a modification to that template is required: the

Component Download operations are moved from the Payload Install sequence to the Payload Fetch sequence.

Then, the install sequence contains the following commands:

* Set Component Index directive (see Section 8.4.10.1)

* Override Parameters directive (see Section 8.4.10.3) for Source Component (see Section 8.4.8.10)

* Copy directive (see Section 8.4.10.5)

* Check Image Match condition (see Section 8.4.9.2)

7.5. Integrated Payload Template

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

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

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

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

7.6. Load from Nonvolatile Storage Template

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

The following commands are placed into the load sequence:

* Set Component Index directive (see Section 8.4.10.1)

* Override Parameters directive (see Section 8.4.10.3) for Source Component (see Section 8.4.8)

* Copy directive (see Section 8.4.10.5)

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

7.7. A/B Image Template

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

The following commands are placed in the common block:

* Set Component Index directive (see Section 8.4.10.1)

* Try Each

  - First Sequence:

    o Override Parameters directive (see Section 8.4.10.3, Section 8.4.8) for Slot A

    o Check Slot Condition (see Section 8.4.9.3)

    o Override Parameters directive (see Section 8.4.10.3) for Image Digest A and Image Size A (see Section 8.4.8)

  - Second Sequence:

    o Override Parameters directive (see Section 8.4.10.3, Section 8.4.8) for Slot B

    o Check Slot Condition (see Section 8.4.9.3)

    o Override Parameters directive (see Section 8.4.10.3) for Image Digest B and Image Size B (see Section 8.4.8)

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

* Set Component Index directive (see Section 8.4.10.1)

* Try Each

  - First Sequence:

    o Override Parameters directive (see Section 8.4.10.3, Section 8.4.8) for Slot A

    o Check Slot Condition (see Section 8.4.9.3)

    o Set Parameters directive (see Section 8.4.10.3) for URI A (see Section 8.4.8)

-Second Sequence:

oOverride Parameters directive (see Section 8.4.10.3, Section 8.4.8) for Slot B

oCheck Slot Condition (see Section 8.4.9.3)

oSet Parameters directive (see Section 8.4.10.3) for URI B (see Section 8.4.8)

*Fetch

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

*Set Component Index directive (see Section 8.4.10.1)

*Try Each

-First Sequence:

oOverride Parameters directive (see Section 8.4.10.3, Section 8.4.8) for Slot A

oCheck Slot Condition (see Section 8.4.9.3)

-Second Sequence:

oOverride Parameters directive (see Section 8.4.10.3, Section 8.4.8) for Slot B

oCheck Slot Condition (see Section 8.4.9.3)

*Run

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

8. Metadata Structure

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

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

8.1. Encoding Considerations

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

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

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

All CBOR maps in the Manifest and manifest envelope MUST be encoded with the canonical CBOR ordering as defined in [RFC8949].

8.2. Envelope

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

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

8.3. Authenticated Manifests

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

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

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

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

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

The algorithms used in SUIT_Authentication are defined by the profiles declared in [I-D.moran-suit-mti].

8.4. Manifest

The manifest contains:

* a version number (see Section 8.4.1)
* a sequence number (see Section 8.4.2)
* a reference URI (see Section 8.4.3)
* a common structure with information that is shared between command sequences (see Section 8.4.5)
* one or more lists of commands that the Recipient should perform (see Section 8.4.6)
* a reference to the full manifest (see Section 8.4.3)
* human-readable text describing the manifest found in the SUIT_Envelope (see Section 8.4.4)

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

8.4.1. suit-manifest-version

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

8.4.2. suit-manifest-sequence-number

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

8.4.3. suit-reference-uri

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

8.4.4. suit-text

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

*integer => text

*SUIT_Component_Identifier => map

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

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

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

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

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

Table 2

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

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

Table 3

suit-text is OPTIONAL to implement.

8.4.5. suit-common

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

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

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

8.4.5.1. SUIT_Component_Identifier

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

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

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

8.4.6. SUIT_Command_Sequence

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

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

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

3. Image Validation: suit-validate is a SUIT_Command_Sequence to execute in order to validate that the result of applying the update is correct. Typical actions involve image validation. suit-validate is REQUIRED to implement.

4. Image Loading: suit-load is a SUIT_Command_Sequence to execute in order to prepare a payload for execution. Typical actions

include copying an image from permanent storage into RAM, optionally including actions such as decryption or decompression. suit-load is OPTIONAL to implement.

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

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

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

Lists of commands are constructed from two kinds of element:

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

2. Directives that MUST be executed.

Each condition is composed of:

1. A command code identifier

2. A SUIT_Reporting_Policy (Section 8.4.7)

Each directive is composed of:

1. A command code identifier

2. An argument block or a SUIT_Reporting_Policy (Section 8.4.7)

Argument blocks are consumed only by flow-control directives:

* Set Component Index

* Set/Override Parameters

* Try Each

* Run Sequence

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

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

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

8.4.7. Reporting Policy

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

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

Table 4

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

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

The information elements provided to the Reporting Engine are:

* The reporting policy

* The result of the command

* The values of parameters consumed by the command

* The system information consumed by the command

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

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

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

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

8.4.8. SUIT_Parameters

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

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

The defined manifest parameters are described below.

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

Table 5

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

8.4.8.1. CBOR PEN UUID Namespace Identifier

The CBOR PEN UUID Namespace Identifier is constructed as follows:

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

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

Computing a type 5 UUID from these produces:

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

8.4.8.2. Constructing UUIDs

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

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

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

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

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

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

1. Hardware model/revision

2. OS

3. WiFi module model/revision

4. Application

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

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

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

The RECOMMENDED method to create a vendor ID is:

```
Vendor ID = UUID5(DNS_PREFIX, vendor domain name)
```

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

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

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

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

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

* Model number.

* Hardware revision.

* Bootloader version (for immutable bootloaders).

8.4.8.3. suit-parameter-vendor-identifier

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

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

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

8.4.8.4. suit-parameter-class-identifier

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

8.4.8.5. suit-parameter-device-identifier

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

8.4.8.6. suit-parameter-image-digest

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

8.4.8.7. suit-parameter-image-size

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

8.4.8.8. suit-parameter-component-slot

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

8.4.8.9. suit-parameter-uri

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

8.4.8.10. suit-parameter-source-component

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

8.4.8.11. suit-parameter-run-args

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

8.4.8.12. suit-parameter-fetch-arguments

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

8.4.8.13. suit-parameter-strict-order

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

See Section 6.7 for behavioral description of Strict Order.

8.4.8.14. suit-parameter-soft-failure

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

suit-parameter-soft-failure is scoped to the enclosing SUIT_Command_Sequence. Its value is discarded when

SUIT_Command_Sequence terminates. It MUST NOT be set outside of suit-directive-try-each or suit-directive-run-sequence.

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

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

8.4.8.15. suit-parameter-custom

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

8.4.9. SUIT_Condition

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

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

Table 6

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

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

Each condition MUST report a result code on completion. If a condition reports failure, then the current sequence of commands MUST terminate. A subsequent command or command sequence MAY continue executing if suit-parameter-soft-failure (Section 8.4.8.14) is set. If a condition requires additional information, this MUST be specified in one or more parameters before the condition is executed. If a Recipient attempts to process a condition that expects additional information and that information has not been set, it MUST report a failure. If a Recipient encounters an unknown condition, it MUST report a failure.

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

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

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

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

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

8.4.9.2. suit-condition-image-match

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

8.4.9.3. suit-condition-component-slot

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

8.4.9.4. suit-condition-abort

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

8.4.9.5. suit-condition-custom

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

8.4.10. SUIT_Directive

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

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

Table 7

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

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

8.4.10.1. suit-directive-set-component-index

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

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

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

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

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

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

8.4.10.2. suit-directive-try-each

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

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

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

8.4.10.3. suit-directive-override-parameters

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

Available parameters are defined in Section 8.4.8.

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

8.4.10.4. suit-directive-fetch

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

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

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

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

8.4.10.5. suit-directive-copy

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

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

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

8.4.10.6. suit-directive-run

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

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

8.4.10.7. suit-directive-run-sequence

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

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

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

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

8.4.10.8. suit-directive-swap

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

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

8.4.11. Integrity Check Values

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

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

8.5. Severable Elements

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

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

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

9. Access Control Lists

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

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

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

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

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

10. SUIT Digest Container

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

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

* SHA-256 (-16)

The following algorithms MAY be implemented in a Manifest Processor:

* SHAKE128 (-18)
* SHA-384 (-43)
* SHA-512 (-44)
* SHAKE256 (-45)

11. IANA Considerations

IANA is requested to:

* allocate CBOR tag 107 in the CBOR Tags registry for the SUIT Envelope.

* allocate CBOR tag 1070 in the CBOR Tags registry for the SUIT Manifest.

* allocate media type application/suit-envelope in the Media Types registry.

* setup several registries as described below.

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

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

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

11.1. SUIT Commands

Label	Name	Reference
1	Vendor Identifier	Section 8.4.9.1
2	Class Identifier	Section 8.4.9.1
3	Image Match	Section 8.4.9.2
4	Reserved	
5	Component Slot	Section 8.4.9.3
12	Set Component Index	Section 8.4.10.1
13	Reserved	
14	Abort	
15	Try Each	Section 8.4.10.2
16	Reserved	
17	Reserved	
18	Reserved	
19	Reserved	
20	Override Parameters	Section 8.4.10.3
21	Fetch	Section 8.4.10.4
22	Copy	Section 8.4.10.5
23	Run	Section 8.4.10.6
24	Device Identifier	Section 8.4.9.1
25	Reserved	
26	Reserved	

Label	Name	Reference
27	Reserved	
28	Reserved	
29	Reserved	
30	Reserved	
31	Swap	Section 8.4.10.8
32	Run Sequence	Section 8.4.10.7
33	Reserved	
nint	Custom Condition	Section 8.4.9.5

Table 8

11.2. SUIT Parameters

Label	Name	Reference
1	Vendor ID	Section 8.4.8.3
2	Class ID	Section 8.4.8.4
3	Image Digest	Section 8.4.8.6
4	Reserved	
5	Component Slot	Section 8.4.8.8
12	Strict Order	Section 8.4.8.13
13	Soft Failure	Section 8.4.8.14
14	Image Size	Section 8.4.8.7
18	Reserved	
19	Reserved	
20	Reserved	
21	URI	Section 8.4.8.9
22	Source Component	Section 8.4.8.10
23	Run Args	Section 8.4.8.11
24	Device ID	Section 8.4.8.5
26	Reserved	
27	Reserved	
28	Reserved	
29	Reserved	
30	Reserved	
nint	Custom	Section 8.4.8.15

Table 9

11.3. SUIT Text Values

Label	Name	Reference
1	Manifest Description	Section 8.4.4
2	Update Description	Section 8.4.4
3	Manifest JSON Source	Section 8.4.4
4	Manifest YAML Source	Section 8.4.4
nint	Custom	Section 8.4.4

Table 10

11.4. SUIT Component Text Values

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

Table 11

12. Security Considerations

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

13. Acknowledgements

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

*Milosch Meriac

*Geraint Luff

*Dan Ros

*John-Paul Stanford

*Hugo Vincent

*Carsten Bormann

*Oeyvind Roenningstad

*Frank Audun Kvamtroe

*Krzysztof Chruściński

*Andrzej Puzdrowski

*Michael Richardson

*David Brown

*Emmanuel Baccelli

14. References

14.1. Normative References

[I-D.moran-suit-mti]
Moran, B., "Mandatory-to-Implement Algorithms for Creators and Consumers of Software Update for the Internet of Things manifests", Work in Progress, Internet-Draft, draft-moran-suit-mti-00, 11 July 2022, <https://www.ietf.org/archive/id/draft-moran-suit-mti-00.txt>.

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

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

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

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

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

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/info/rfc8949>.

[RFC9019] Moran, B., Tschofenig, H., Brown, D., and M. Meriac, "A Firmware Update Architecture for Internet of Things", RFC 9019, DOI 10.17487/RFC9019, April 2021, <https://www.rfc-editor.org/info/rfc9019>.

[RFC9124] Moran, B., Tschofenig, H., and H. Birkholz, "A Manifest Information Model for Firmware Updates in Internet of

Things (IoT) Devices", RFC 9124, DOI 10.17487/RFC9124, January 2022, <https://www.rfc-editor.org/info/rfc9124>.

14.2. Informative References

[I-D.ietf-cbor-tags-oid] Bormann, C., "Concise Binary Object Representation (CBOR) Tags for Object Identifiers", Work in Progress, Internet-Draft, draft-ietf-cbor-tags-oid-08, 21 May 2021, <https://www.ietf.org/archive/id/draft-ietf-cbor-tags-oid-08.txt>.

[I-D.ietf-cose-hash-algs] Schaad, J., "CBOR Object Signing and Encryption (COSE): Hash Algorithms", Work in Progress, Internet-Draft, draft-ietf-cose-hash-algs-09, 14 September 2020, <https://www.ietf.org/archive/id/draft-ietf-cose-hash-algs-09.txt>.

[I-D.ietf-suit-firmware-encryption] Tschofenig, H., Housley, R., and B. Moran, "Firmware Encryption with SUIT Manifests", Work in Progress, Internet-Draft, draft-ietf-suit-firmware-encryption-06, 11 July 2022, <https://www.ietf.org/archive/id/draft-ietf-suit-firmware-encryption-06.txt>.

[I-D.ietf-suit-report] Moran, B. and H. Birkholz, "Secure Reporting of Update Status", Work in Progress, Internet-Draft, draft-ietf-suit-report-01, 12 January 2022, <https://www.ietf.org/archive/id/draft-ietf-suit-report-01.txt>.

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

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

[I-D.ietf-teep-architecture] Pei, M., Tschofenig, H., Thaler, D., and D. Wheeler, "Trusted Execution Environment Provisioning (TEEP) Architecture", Work in Progress, Internet-Draft, draft-ietf-teep-architecture-18, 11 July

2022, <https://www.ietf.org/archive/id/draft-ietf-teep-architecture-18.txt>.

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

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

Appendix A. A. Full CDDL

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

To be valid, the following CDDL MUST have the COSE CDDL appended to it. The COSE CDDL can be obtained by following the directions in [RFC8152], Section 1.3.

```
SUIT_Envelope_Tagged = #6.107(SUIT_Envelope)
SUIT_Envelope = {
  suit-authentication-wrapper => bstr .cbor SUIT_Authentication,
  suit-manifest  => bstr .cbor SUIT_Manifest,
  SUIT_Severable_Manifest_Members,
  * SUIT_Integrated_Payload,
  * $$SUIT_Envelope_Extensions,
}

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

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

SUIT_Authentication_Block /= COSE_Mac_Tagged
SUIT_Authentication_Block /= COSE_Sign_Tagged
SUIT_Authentication_Block /= COSE_Mac0_Tagged
SUIT_Authentication_Block /= COSE_Sign1_Tagged

SUIT_Severable_Manifest_Members = (
  ? suit-payload-fetch => bstr .cbor SUIT_Command_Sequence,
  ? suit-install => bstr .cbor SUIT_Command_Sequence,
  ? suit-text => bstr .cbor SUIT_Text_Map,
  * $$SUIT_severable-members-extensions,
)

SUIT_Integrated_Payload = (suit-integrated-payload-key => bstr)
suit-integrated-payload-key = tstr

SUIT_Manifest_Tagged = #6.1070(SUIT_Manifest)

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

SUIT_Unseverable_Members = (
  ? suit-validate => bstr .cbor SUIT_Command_Sequence,
  ? suit-load => bstr .cbor SUIT_Command_Sequence,
```

```
  ? suit-run => bstr .cbor SUIT_Command_Sequence,
  * $$unseverable-manifest-member-extensions,
)

SUIT_Severable_Members_Choice = (
  ? suit-payload-fetch => SUIT_Digest /
      bstr .cbor SUIT_Command_Sequence,
  ? suit-install => SUIT_Digest / bstr .cbor SUIT_Command_Sequence,
  ? suit-text => SUIT_Digest / bstr .cbor SUIT_Text_Map,
  * $$severable-manifest-members-choice-extensions
)

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

SUIT_Components           = [ + SUIT_Component_Identifier ]

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

;REQUIRED to implement:
suit-cose-hash-algs /= cose-alg-sha-256

;OPTIONAL to implement:
suit-cose-hash-algs /= cose-alg-shake128
suit-cose-hash-algs /= cose-alg-sha-384
suit-cose-hash-algs /= cose-alg-sha-512
suit-cose-hash-algs /= cose-alg-shake256

SUIT_Component_Identifier =  [* bstr]

SUIT_Common_Sequence = [
    + ( SUIT_Condition // SUIT_Common_Commands )
]

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

IndexArg /= uint
```

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


SUIT_Directive_Try_Each_Argument_Common = [
    2* bstr .cbor SUIT_Common_Sequence,
    ?nil
]

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

SUIT_Command_Custom = (suit-command-custom, bstr/tstr/int/nil)
SUIT_Condition //= (suit-condition-vendor-identifier, SUIT_Rep_Policy)
SUIT_Condition //= (suit-condition-class-identifier,  SUIT_Rep_Policy)
SUIT_Condition //= (suit-condition-device-identifier, SUIT_Rep_Policy)
SUIT_Condition //= (suit-condition-image-match,       SUIT_Rep_Policy)
SUIT_Condition //= (suit-condition-component-slot,    SUIT_Rep_Policy)
SUIT_Condition //= (suit-condition-abort,             SUIT_Rep_Policy)

SUIT_Directive //= (suit-directive-set-component-index,  IndexArg)
SUIT_Directive //= (suit-directive-run-sequence,
    bstr .cbor SUIT_Command_Sequence)
SUIT_Directive //= (suit-directive-try-each,
    SUIT_Directive_Try_Each_Argument)
SUIT_Directive //= (suit-directive-override-parameters,
    {+ SUIT_Parameters})
SUIT_Directive //= (suit-directive-fetch,             SUIT_Rep_Policy)
SUIT_Directive //= (suit-directive-copy,              SUIT_Rep_Policy)
SUIT_Directive //= (suit-directive-swap,              SUIT_Rep_Policy)
SUIT_Directive //= (suit-directive-run,               SUIT_Rep_Policy)

SUIT_Directive_Try_Each_Argument = [
    2* bstr .cbor SUIT_Command_Sequence,
    ?nil
]

SUIT_Rep_Policy = uint .bits suit-reporting-bits

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

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

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

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

SUIT_Parameters //= (suit-parameter-device-identifier => RFC4122_UUID)

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

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

RFC4122_UUID = bstr .size 16

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

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

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

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

;REQUIRED to implement:
cose-alg-sha-256 = -16
```

```
;OPTIONAL to implement:
cose-alg-shake128 = -18
cose-alg-sha-384 = -43
cose-alg-sha-512 = -44
cose-alg-shake256 = -45

;Unseverable, recipient-necessary
suit-manifest-version = 1
suit-manifest-sequence-number = 2
suit-common = 3
suit-reference-uri = 4
suit-validate = 7
suit-load = 8
suit-run = 9
;Severable, recipient-necessary
suit-payload-fetch = 16
suit-install = 17
;Severable, recipient-unnecessary
suit-text = 23

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

suit-command-custom = nint

suit-condition-vendor-identifier = 1
suit-condition-class-identifier  = 2
suit-condition-image-match       = 3
suit-condition-component-slot    = 5

suit-condition-abort                    = 14
suit-condition-device-identifier        = 24

suit-directive-set-component-index      = 12
suit-directive-try-each                 = 15
suit-directive-override-parameters      = 20
suit-directive-fetch                    = 21
suit-directive-copy                     = 22
suit-directive-run                      = 23

suit-directive-swap                     = 31
suit-directive-run-sequence             = 32

suit-parameter-vendor-identifier = 1
suit-parameter-class-identifier  = 2
suit-parameter-image-digest      = 3
suit-parameter-component-slot    = 5

suit-parameter-strict-order      = 12
```

```
suit-parameter-soft-failure      = 13
suit-parameter-image-size        = 14

suit-parameter-uri               = 21
suit-parameter-source-component  = 22
suit-parameter-run-args          = 23

suit-parameter-device-identifier = 24

suit-parameter-custom = nint

suit-text-manifest-description  = 1
suit-text-update-description    = 2
suit-text-manifest-json-source  = 3
suit-text-manifest-yaml-source  = 4

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

Appendix B. B. Examples

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

The examples are signed using the following ECDSA secp256r1 key:


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


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


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


Each example uses SHA256 as the digest function.

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

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

Table 12

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

Table 13

B.1. Example 0: Secure Boot

This example covers the following templates:

* Compatibility Check (Section 7.1)

* Secure Boot (Section 7.2)

It also serves as the minimum example.

```
107({
        / authentication-wrapper / 2:<<[
            digest: <<[
                / algorithm-id / -16 / "sha256" /,
                / digest-bytes /
h'6658ea560262696dd1f13b782239a064da7c6c5cbaf52fded428a6fc83c7e5af'
            ]>>,
            signature: <<18([
                    / protected / <<{
                        / alg / 1:-7 / "ES256" /,
                    }>>,
                    / unprotected / {
                    },
                    / payload / F6 / nil /,
                    / signature / h'01d9cbf51b1ccadf839ec867da85735118
ba4c1b513f5df60583e8dd7f8ce4274cd2e2a0f70f8832bfea63983172e77a951e9a36
e74cada19321b903a097e4e7'
                ])>>
            ]
        ]>>,
        / manifest / 3:<<{
            / manifest-version / 1:1,
            / manifest-sequence-number / 2:0,
            / common / 3:<<{
                / components / 2:[
                    [h'00']
                ],
                / common-sequence / 4:<<[
                    / directive-override-parameters / 20,{
                        / vendor-id /
1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
                        / class-id /
2:h'1492af1425695e48bf429b2d51f2ab45' /
1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
                        / image-digest / 3:<<[
                            / algorithm-id / -16 / "sha256" /,
                            / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
                        ]>>,
                        / image-size / 14:34768,
                    } ,
                    / condition-vendor-identifier / 1,15 ,
                    / condition-class-identifier / 2,15
                ]>>,
            }>>,
            / validate / 7:<<[
                / condition-image-match / 3,15
            ]>>,
```

```
        / run / 9:<<[
            / directive-run / 23,2
        ]>>,
    }>>,
})
```

Total size of Envelope without COSE authentication object: 161

Envelope:

d86ba2025827815824822f58206658ea560262696dd1f13b782239a064da
7c6c5cbaf52fded428a6fc83c7e5af035871a50101020003585fa2028181
41000458568614a40150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492
af1425695e48bf429b2d51f2ab45035824822f582000112233445566778899aabbccddeeff0123456789abcdeffedcba98765432100e1987d0010f02
0f074382030f0943821702

Total size of Envelope with COSE authentication object: 237

Envelope with COSE authentication object:

d86ba2025873825824822f58206658ea560262696dd1f13b782239a064da
7c6c5cbaf52fded428a6fc83c7e5af584ad28443a10126a0f6584001d9cb
f51b1ccadf839ec867da85735118ba4c1b513f5df60583e8dd7f8ce4274c
d2e2a0f70f8832bfea63983172e77a951e9a36e74cada19321b903a097e4
e7035871a50101020003585fa202818141000458568614a40150fa6b4a53
d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab45
035824822f582000112233445566778899aabbccddeeff0123456789abcd
effedcba98765432100e1987d0010f020f074382030f0943821702

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

This example covers the following templates:

*Compatibility Check (Section 7.1)

*Firmware Download (Section 7.3)

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

```
107({
        / authentication-wrapper / 2:<<[
            digest: <<[
                / algorithm-id / -16 / "sha256" /,
                / digest-bytes /
h'176653ec03837a8d32b1eb0b91ea5e8a2256bc2e7c02e5bb7f45f431fa926f3e'
            ]>>,
            signature: <<18([
                    / protected / <<{
                        / alg / 1:-7 / "ES256" /,
                    }>>,
                    / unprotected / {
                    },
                    / payload / F6 / nil /,
                    / signature / h'a78dbd584dbdea66ab024179dac2f4b244
940e9dfe1337bb833e3e3df2562a0fceb77e64980e0ddcc24b8abb63b78aad9ad42fd2
44b684dd39859f84740239e8'
                ])>>
            ]
        ]>>,
        / manifest / 3:<<{
            / manifest-version / 1:1,
            / manifest-sequence-number / 2:1,
            / common / 3:<<{
                / components / 2:[
                    [h'00']
                ],
                / common-sequence / 4:<<[
                    / directive-override-parameters / 20,{
                        / vendor-id /
1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
                        / class-id /
2:h'1492af1425695e48bf429b2d51f2ab45' /
1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
                        / image-digest / 3:<<[
                            / algorithm-id / -16 / "sha256" /,
                            / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
                        ]>>,
                        / image-size / 14:34768,
                    } ,
                    / condition-vendor-identifier / 1,15 ,
                    / condition-class-identifier / 2,15
                ]>>,
            }>>,
            / validate / 7:<<[
                / condition-image-match / 3,15
            ]>>,
```

```
        / install / 17:<<[
            / directive-set-parameters / 19,{
                / uri / 21:'http://example.com/file.bin',
            } ,
            / directive-fetch / 21,2 ,
            / condition-image-match / 3,15
        ]>>,
    }>>,
})
```

Total size of Envelope without COSE authentication object: 196

Envelope:


```
d86ba2025827815824822f5820176653ec03837a8d32b1eb0b91ea5e8a22
56bc2e7c02e5bb7f45f431fa926f3e035894a50101020103585fa2028181
41000458568614a40150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492
af1425695e48bf429b2d51f2ab45035824822f582000112233445566778899aabbccddeeff0123456789abcdeffedcba98765432100e1987d0010f02
0f074382030f1158258613a115781b687474703a2f2f6578616d706c652e
636f6d2f66696c652e62696e1502030f
```


Total size of Envelope with COSE authentication object: 272

Envelope with COSE authentication object:


```
d86ba2025873825824822f5820176653ec03837a8d32b1eb0b91ea5e8a22
56bc2e7c02e5bb7f45f431fa926f3e584ad28443a10126a0f65840a78dbd
584dbdea66ab024179dac2f4b244940e9dfe1337bb833e3e3df2562a0fce
b77e64980e0ddcc24b8abb63b78aad9ad42fd244b684dd39859f84740239
e8035894a50101020103585fa202818141000458568614a40150fa6b4a53
d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab45
035824822f582000112233445566778899aabbccddeeff0123456789abcd
effedcba98765432100e1987d0010f020f074382030f1158258613a11578
1b687474703a2f2f6578616d706c652e636f6d2f66696c652e62696e1502
030f
```


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

This example covers the following templates:

*Compatibility Check (Section 7.1)

*Secure Boot (Section 7.2)

*Firmware Download (Section 7.3)

This example also demonstrates severable elements (Section 5.4), and text (Section 8.4.4).

```
107({
        / authentication-wrapper / 2:<<[
            digest: <<[
                / algorithm-id / -16 / "sha256" /,
                / digest-bytes /
h'a85153c05f709e681877ee23c0de3e2f92bcc66c1ad6f41b39157ac7cb6a5a62'
            ]>>,
            signature: <<18([
                    / protected / <<{
                        / alg / 1:-7 / "ES256" /,
                    }>>,
                    / unprotected / {
                    },
                    / payload / F6 / nil /,
                    / signature / h'4ba6e9c4bdd65212e2e4775b7f5bb32faf
14209f88d9f8b198e21c338770aa542dde07e31fa17ca9dadee4d94c43dfba81819d35
88d7fde5eff010b8c7c89277'
                ])>>
            ]
        ]>>,
        / manifest / 3:<<{
            / manifest-version / 1:1,
            / manifest-sequence-number / 2:2,
            / common / 3:<<{
                / components / 2:[
                    [h'00']
                ],
                / common-sequence / 4:<<[
                    / directive-override-parameters / 20,{
                        / vendor-id /
1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
                        / class-id /
2:h'1492af1425695e48bf429b2d51f2ab45' /
1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
                        / image-digest / 3:<<[
                            / algorithm-id / -16 / "sha256" /,
                            / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
                        ]>>,
                        / image-size / 14:34768,
                    } ,
                    / condition-vendor-identifier / 1,15 ,
                    / condition-class-identifier / 2,15
                ]>>,
            }>>,
            / validate / 7:<<[
                / condition-image-match / 3,15
            ]>>,
```

```
            / run / 9:<<[
                / directive-run / 23,2
            ]>>,
            / install / 17:[
                / algorithm-id / -16 / "sha256" /,
                / digest-bytes /
h'3ee96dc79641970ae46b929ccf0b72ba9536dd846020dbdc9f949d84ea0e18d2'
            ],
            / text / 23:[
                / algorithm-id / -16 / "sha256" /,
                / digest-bytes /
h'2bfc4d0cc6680be7dd9f5ca30aa2bb5d1998145de33d54101b80e2ca49faf918'
            ],
        }>>,
        / install / 17:<<[
            / directive-set-parameters / 19,{
                / uri /
21:'http://example.com/very/long/path/to/file/file.bin',
            } ,
            / directive-fetch / 21,2 ,
            / condition-image-match / 3,15
        ]>>,
        / text / 13:<<{
            [h'00']:{
                    / vendor-domain / 3:'arm.com',
                    / component-description / 5:'This component is a
demonstration. The digest is a sample pattern, not a real one.',
                }
        }>>,
    })
```

Total size of the Envelope without COSE authentication object or Severable Elements: 235

Envelope:

d86ba2025827815824822f5820a85153c05f709e681877ee23c0de3e2f92
bcc66c1ad6f41b39157ac7cb6a5a620358bba70101020203585fa2028181
41000458568614a40150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492
af1425695e48bf429b2d51f2ab45035824822f5820001122334455667788
99aabbccddeeff0123456789abcdeffedcba98765432100e1987d0010f02
0f074382030f094382170211822f58203ee96dc79641970ae46b929ccf0b
72ba9536dd846020dbdc9f949d84ea0e18d217822f58202bfc4d0cc6680b
e7dd9f5ca30aa2bb5d1998145de33d54101b80e2ca49faf918

Total size of the Envelope with COSE authentication object but without Severable Elements: 311

Envelope:

d86ba2025873825824822f5820a85153c05f709e681877ee23c0de3e2f92
bcc66c1ad6f41b39157ac7cb6a5a62584ad28443a10126a0f658404ba6e9
c4bdd65212e2e4775b7f5bb32faf14209f88d9f8b198e21c338770aa542d
de07e31fa17ca9dadee4d94c43dfba81819d3588d7fde5eff010b8c7c892
770358bba70101020203585fa2028181410004585686144a40150fa6b4a53
d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab45
035824822f58200011223344556677889aabbccddeeff0123456789abcd
effedcba98765432100e1987d0010f020f074382030f094382170211822f
58203ee96dc79641970ae46b929ccf0b72ba9536dd846020dbdc9f949d84
ea0e18d217822f58202bfc4d0cc6680be7dd9f5ca30aa2bb5d1998145de3
3d54101b80e2ca49faf918

Total size of Envelope with COSE authentication object and Severable Elements: 894

Envelope with COSE authentication object:

d86ba4025873825824822f5820a85153c05f709e681877ee23c0de3e2f92
bcc66c1ad6f41b39157ac7cb6a5a62584ad28443a10126a0f658404ba6e9
c4bdd65212e2e4775b7f5bb32faf14209f88d9f8b198e21c338770aa542d
de07e31fa17ca9dadee4d94c43dfba81819d3588d7fde5eff010b8c7c892
770358bba70101020203585fa202818141000458568614a40150fa6b4a53
d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab45
035824822f582000112233445566778899aabbccddeeff0123456789abcd
effedcba98765432100e1987d0010f020f074382030f094382170211822f
58203ee96dc79641970ae46b929ccf0b72ba9536dd846020dbdc9f949d84
ea0e18d217822f58202bfc4d0cc6680be7dd9f5ca30aa2bb5d1998145de3
3d54101b80e2ca49faf9180d590204a20179019d2323204578616d706c65
20323a2053696d756c74616e656f757320446f776e6c6f61642c20496e73
74616c6c6174696f6e2c20536563757265204f6f742c20536576657265
64204669656c64730a0a2020202054686973206578616d706c6520636f76
65727320746865206666f6c6c6f77696e672074656d706c617465733a0a20
2020200a20202020202a20436f6d7061746962696c69747920436865636b20
287b7b74656d706c6174652d636f6d7061746962696c6974792d63686563
6b7d7d290a20202020202a20536563757265204f6f7420287b7b74656d70
6c6174652d7365637572652d626f6f747d7d290a20202020202a204669726d
7761726520446f776e6c6f616420287b7b6669726d776172652d646f776e
6c6f61642d74656d706c6174657d7d290a202020200a2020202054686973
206578616d706c6520616c736f2064656d6f6e737472617465732073657665
657261626c6520656c656d656e747320287b7b6f76722d736576657261626
6c657d7d292c20616e64207465787420287b7b6d616e69666573742d6469
676573742d746578747d7d292e814100a2036761726d2e636f6d05785254
68697320636f6d706f6e656e7420697320612064656d6f6e737472617469
6f6e2e205468652064696765737420697320612073616d706c6520706174
7465726e2c206e6f742061207265616c206f6e652e11583c8613a1157832
687474703a2f2f6578616d706c652e636f6d2f766572792f6c6f6e672f70
6174682f746f2f66696c652f66696c652e62696e1502030f

B.4. Example 3: A/B images

This example covers the following templates:

*Compatibility Check (Section 7.1)

*Secure Boot (Section 7.2)

*Firmware Download (Section 7.3)

*A/B Image Template (Section 7.7)

```
107({
        / authentication-wrapper / 2:<<[
            digest: <<[
                / algorithm-id / -16 / "sha256" /,
                / digest-bytes /
h'c98d9240343ee1ac12ba833c04fb9006e70f62c7e4c36edb0b2a356d59c2f86c'
            ]>>,
            signature: <<18([
                    / protected / <<{
                        / alg / 1:-7 / "ES256" /,
                    }>>,
                    / unprotected / {
                    },
                    / payload / F6 / nil /,
                    / signature / h'451b3099c7914ef4c54b633688471b8d0f
940d09eeca41c159927a9f044bddec536f83da5f1b1047bc415be013d71524ad82e4ac
792a61f93dbdc875a7a6adeb'
                ])>>
            ]
        ]>>,
        / manifest / 3:<<{
            / manifest-version / 1:1,
            / manifest-sequence-number / 2:3,
            / common / 3:<<{
                / components / 2:[
                    [h'00']
                ],
                / common-sequence / 4:<<[
                    / directive-override-parameters / 20,{
                        / vendor-id /
1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
                        / class-id /
2:h'1492af1425695e48bf429b2d51f2ab45' /
1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
                    } ,
                    / directive-try-each / 15,[
                        <<[
                            / directive-override-parameters / 20,{
                                / slot / 5:0,
                            } ,
                            / condition-component-slot / 5,5 ,
                            / directive-override-parameters / 20,{
                                / image-digest / 3:<<[
                                    / algorithm-id / -16 / "sha256" /,
                                    / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
                                ]>>,
                                / image-size / 14:34768,
```

```
                        }
                    ]>> ,
                    <<[
                        / directive-override-parameters / 20,{
                            / slot / 5:1,
                        } ,
                        / condition-component-slot / 5,5 ,
                        / directive-override-parameters / 20,{
                            / image-digest / 3:<<[
                                / algorithm-id / -16 / "sha256" /,
                                / digest-bytes /
h'0123456789abcdeffedcba98765432100011223344556677889 9aabbccddeeff'
                            ]>>,
                            / image-size / 14:76834,
                        }
                    ]>>
                ] ,
                / condition-vendor-identifier / 1,15 ,
                / condition-class-identifier / 2,15
            ]>>,
        }>>,
        / validate / 7:<<[
            / condition-image-match / 3,15
        ]>>,
        / install / 17:<<[
            / directive-try-each / 15,[
                <<[
                    / directive-set-parameters / 19,{
                        / slot / 5:0,
                    } ,
                    / condition-component-slot / 5,5 ,
                    / directive-set-parameters / 19,{
                        / uri / 21:'http://example.com/file1.bin',
                    }
                ]>> ,
                <<[
                    / directive-set-parameters / 19,{
                        / slot / 5:1,
                    } ,
                    / condition-component-slot / 5,5 ,
                    / directive-set-parameters / 19,{
                        / uri / 21:'http://example.com/file2.bin',
                    }
                ]>>
            ] ,
            / directive-fetch / 21,2 ,
            / condition-image-match / 3,15
        ]>>,
```

```
    }>>,
})
```

Total size of Envelope without COSE authentication object: 320

Envelope:


```
d86ba2025827815824822f5820c98d9240343ee1ac12ba833c04fb9006e7
0f62c7e4c36edb0b2a356d59c2f86c0359010fa5010102030358a4a20281
81410004589b8814a20150fa6b4a53d5ad5fdfbe9de663e4d41ffe025014
92af1425695e48bf429b2d51f2ab450f8258348614a10500050514a20358
24822f582000112233445566778899aabbccddeeff0123456789abcdeffe
dcba98765432100e1987d058368614a10501050514a2035824822f582001
23456789abcdeffedcba987654321000112233445566778899aabbccddee
ff0e1a00012c22010f020f074382030f11585b860f8258288613a1050005
0513a115781c687474703a2f2f6578616d706c652e636f6d2f66696c6531
2e62696e58288613a10501050513a115781c687474703a2f2f6578616d70
6c652e636f6d2f66696c65322e62696e1502030f
```


Total size of Envelope with COSE authentication object: 396

Envelope with COSE authentication object:


```
d86ba2025873825824822f5820c98d9240343ee1ac12ba833c04fb9006e7
0f62c7e4c36edb0b2a356d59c2f86c584ad28443a10126a0f65840451b30
99c7914ef4c54b633688471b8d0f940d09eeca41c159927a9f044bddec53
6f83da5f1b1047bc415be013d71524ad82e4ac792a61f93dbdc875a7a6ad
eb0359010fa5010102030358a4a2028181410004589b8814a20150fa6b4a
53d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab
450f8258348614a10500050514a2035824822f5820001122334455667788
99aabbccddeeff0123456789abcdeffedcba98765432100e1987d0583686
14a10501050514a2035824822f58200123456789abcdeffedcba98765432
1000112233445566778899aabbccddeeff0e1a00012c22010f020f074382
030f11585b860f8258288613a10500050513a115781c687474703a2f2f65
78616d706c652e636f6d2f66696c65312e62696e58288613a10501050513
a115781c687474703a2f2f6578616d706c652e636f6d2f66696c65322e62
696e1502030f
```


B.5. Example 4: Load from External Storage

This example covers the following templates:

- *Compatibility Check (Section 7.1)
- *Secure Boot (Section 7.2)
- *Firmware Download (Section 7.3)
- *Install (Section 7.4)
- *Load (Section 7.6)

```
107({
        / authentication-wrapper / 2:<<[
            digest: <<[
                / algorithm-id / -16 / "sha256" /,
                / digest-bytes /
h'601ebc1bb2e12cbaf408b1bca72fae0d9987498acfa16130ce4cf5cc9ea74c7c'
            ]>>,
            signature: <<18([
                    / protected / <<{
                        / alg / 1:-7 / "ES256" /,
                    }>>,
                    / unprotected / {
                    },
                    / payload / F6 / nil /,
                    / signature / h'2e263599b0f3613fd3feb0cec1ff55c6b3
7c521339ef2680dc63de3a5cdfb0e3f44237313e1c17c35f7fa84af82234f50cea551c
fdd8179a40dac5136167cd5e'
                ])>>
            ]
        ]>>,
        / manifest / 3:<<{
            / manifest-version / 1:1,
            / manifest-sequence-number / 2:4,
            / common / 3:<<{
                / components / 2:[
                    [h'00'] ,
                    [h'02'] ,
                    [h'01']
                ],
                / common-sequence / 4:<<[
                    / directive-set-component-index / 12,0 ,
                    / directive-override-parameters / 20,{
                        / vendor-id /
1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
                        / class-id /
2:h'1492af1425695e48bf429b2d51f2ab45' /
1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
                        / image-digest / 3:<<[
                            / algorithm-id / -16 / "sha256" /,
                            / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
                        ]>>,
                        / image-size / 14:34768,
                    } ,
                    / condition-vendor-identifier / 1,15 ,
                    / condition-class-identifier / 2,15
                ]>>,
            }>>,
```

```
            / validate / 7:<<[
                / directive-set-component-index / 12,0 ,
                / condition-image-match / 3,15
            ]>>,
            / load / 8:<<[
                / directive-set-component-index / 12,2 ,
                / directive-set-parameters / 19,{
                    / image-digest / 3:<<[
                        / algorithm-id / -16 / "sha256" /,
                        / digest-bytes /
h'0123456789abcdeffedcba987654321000112233445566778899aabbccddeeff'
                    ]>>,
                    / image-size / 14:76834,
                    / source-component / 22:0 / [h'00'] /,
                    / compression-info / 19:<<{
                        / compression-algorithm / 1:1 / "zlib" /,
                    }>>,
                } ,
                / directive-copy / 22,2 ,
                / condition-image-match / 3,15
            ]>>,
            / run / 9:<<[
                / directive-set-component-index / 12,2 ,
                / directive-run / 23,2
            ]>>,
            / payload-fetch / 16:<<[
                / directive-set-component-index / 12,1 ,
                / directive-set-parameters / 19,{
                    / image-digest / 3:<<[
                        / algorithm-id / -16 / "sha256" /,
                        / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
                    ]>>,
                    / uri / 21:'http://example.com/file.bin',
                } ,
                / directive-fetch / 21,2 ,
                / condition-image-match / 3,15
            ]>>,
            / install / 17:<<[
                / directive-set-component-index / 12,0 ,
                / directive-set-parameters / 19,{
                    / source-component / 22:1 / [h'02'] /,
                } ,
                / directive-copy / 22,2 ,
                / condition-image-match / 3,15
            ]>>,
        }>>,
    })
```

Total size of Envelope without COSE authentication object: 332

Envelope:


```
d86ba2025827815824822f5820601ebc1bb2e12cbaf408b1bca72fae0d99
87498acfa16130ce4cf5cc9ea74c7c0359011ba801010204035867a20283
814100814102814101045858880c0014a40150fa6b4a53d5ad5fdfbe9de6
63e4d41ffe02501492af1425695e48bf429b2d51f2ab45035824822f5820
00112233445566778899aabbccddeeff0123456789abcdeffedcba987654
32100e1987d0010f020f0745840c00030f08583d880c0213a4035824822f
58200123456789abcdeffedcba987654321000112233445566778899aabb
ccddeeff0e1a00012c221343a1010116001602030f0945840c0217021058
4e880c0113a2035824822f582000112233445566778899aabbccddeeff01
23456789abcdeffedcba98765432101578 1b687474703a2f2f6578616d70
6c652e636f6d2f66696c652e62696e1502030f114b880c0013a116011602
030f
```


Total size of Envelope with COSE authentication object: 408

Envelope with COSE authentication object:


```
d86ba2025873825824822f5820601ebc1bb2e12cbaf408b1bca72fae0d99
87498acfa16130ce4cf5cc9ea74c7c584ad28443a10126a0f658402e2635
99b0f3613fd3feb0cec1ff55c6b37c521339ef2680dc63de3a5cdfb0e3f4
4237313e1c17c35f7fa84af82234f50cea551cfdd8179a40dac5136167cd
5e0359011ba801010204035867a20283814100814102814101045858880c
0014a40150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492af1425695e
48bf429b2d51f2ab45035824822f582000112233445566778899aabbccdd
eeff0123456789abcdeffedcba98765432100e1987d0010f020f0745840c
00030f08583d880c0213a4035824822f58200123456789abcdeffedcba98
7654321000112233445566778899aabbccddeeff0e1a00012c221343a101
0116001602030f0945840c02170210584e880c0113a2035824822f582000
112233445566778899aabbccddeeff0123456789abcdeffedcba98765432
1015781b687474703a2f2f6578616d706c652e636f6d2f66696c652e6269
6e1502030f114b880c0013a116011602030f
```


B.6. Example 5: Two Images

This example covers the following templates:

*Compatibility Check (Section 7.1)

*Secure Boot (Section 7.2)

*Firmware Download (Section 7.3)

Furthermore, it shows using these templates with two images.

```
107({
        / authentication-wrapper / 2:<<[
            digest: <<[
                / algorithm-id / -16 / "sha256" /,
                / digest-bytes /
h'a4c6d5f5c3800c19c4af55aacc1c2dc6e37e2bf10b2aab335f70226961e310d3'
            ]>>,
            signature: <<18([
                    / protected / <<{
                        / alg / 1:-7 / "ES256" /,
                    }>>,
                    / unprotected / {
                    },
                    / payload / F6 / nil /,
                    / signature / h'91d95d3bb2eaae7b31ff11f4761056e491
bcb07470119f9c69388982c3238eabfcb477ec7887f36c31e7d957fe8830b3ae8b9d7d
71372de2e71a9a3b67444c4a'
                ])>>
            ]
        ]>>,
        / manifest / 3:<<{
            / manifest-version / 1:1,
            / manifest-sequence-number / 2:5,
            / common / 3:<<{
                / components / 2:[
                    [h'00'] ,
                    [h'01']
                ],
                / common-sequence / 4:<<[
                    / directive-set-component-index / 12,0 ,
                    / directive-override-parameters / 20,{
                        / vendor-id /
1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
                        / class-id /
2:h'1492af1425695e48bf429b2d51f2ab45' /
1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
                        / image-digest / 3:<<[
                            / algorithm-id / -16 / "sha256" /,
                            / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
                        ]>>,
                        / image-size / 14:34768,
                    } ,
                    / condition-vendor-identifier / 1,15 ,
                    / condition-class-identifier / 2,15 ,
                    / directive-set-component-index / 12,1 ,
                    / directive-override-parameters / 20,{
                        / image-digest / 3:<<[
```

```
                            / algorithm-id / -16 / "sha256" /,
                            / digest-bytes /
h'0123456789abcdeffedcba987654321000112233445566778899aabbccddeeff'
                        ]>>,
                        / image-size / 14:76834,
                    }
                ]>>,
            }>>,
            / validate / 7:<<[
                / directive-set-component-index / 12,0 ,
                / condition-image-match / 3,15 ,
                / directive-set-component-index / 12,1 ,
                / condition-image-match / 3,15
            ]>>,
            / run / 9:<<[
                / directive-set-component-index / 12,0 ,
                / directive-run / 23,2
            ]>>,
            / install / 17:<<[
                / directive-set-component-index / 12,0 ,
                / directive-set-parameters / 19,{
                    / uri / 21:'http://example.com/file1.bin',
                } ,
                / directive-fetch / 21,2 ,
                / condition-image-match / 3,15 ,
                / directive-set-component-index / 12,1 ,
                / directive-set-parameters / 19,{
                    / uri / 21:'http://example.com/file2.bin',
                } ,
                / directive-fetch / 21,2 ,
                / condition-image-match / 3,15
            ]>>,
        }>>,
    })
```

Total size of Envelope without COSE authentication object: 306

Envelope:


```
d86ba2025827815824822f5820a4c6d5f5c3800c19c4af55aacc1c2dc6e3
7e2bf10b2aab335f70226961e310d303590101a601010205035895a20282
8141008141010458898c0c0014a40150fa6b4a53d5ad5fdfbe9de663e4d4
1ffe02501492af1425695e48bf429b2d51f2ab45035824822f5820001122
33445566778899aabbccddeeff0123456789abcdeffedcba98765432100e
1987d0010f020f0c0114a2035824822f58200123456789abcdeffedcba98
76543210001122334455667788899aabbccddeeff0e1a00012c220749880c
00030f0c01030f0945840c00170211584f900c0013a115781c687474703a
2f2f6578616d706c652e636f6d2f66696c65312e62696e1502030f0c0113
a115781c687474703a2f2f6578616d706c652e636f6d2f66696c65322e62
696e1502030f
```


Total size of Envelope with COSE authentication object: 382

Envelope with COSE authentication object:


```
d86ba2025873825824822f5820a4c6d5f5c3800c19c4af55aacc1c2dc6e3
7e2bf10b2aab335f70226961e310d3584ad28443a10126a0f6584091d95d
3bb2eaae7b31ff11f4761056e491bcb07470119f9c69388982c3238eabfc
b477ec7887f36c31e7d957fe8830b3ae8b9d7d71372de2e71a9a3b67444c
4a03590101a601010205035895a202828141008141010458898c0c0014a4
0150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf42
9b2d51f2ab45035824822f582000112233445566778899aabbccddeeff01
23456789abcdeffedcba98765432100e1987d0010f020f0c0114a2035824
822f58200123456789abcdeffedcba987654321000112233445566778899
aabbccddeeff0e1a00012c220749880c00030f0c01030f0945840c001702
11584f900c0013a115781c687474703a2f2f6578616d706c652e636f6d2f
66696c65312e62696e1502030f0c0113a115781c687474703a2f2f657861
6d706c652e636f6d2f66696c65322e62696e1502030f
```


Appendix C. C. Design Rational

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

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

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

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

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

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

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

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

C.1. C.1 Design Rationale: Envelope

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

1. This enables the use of Severable Elements (Section 8.5)

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

3. This enables multiple authentication schemes.

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

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

Four authentication objects are supported by the Envelope:

* COSE_Sign_Tagged

* COSE_Sign1_Tagged

* COSE_Mac_Tagged

* COSE_Mac0_Tagged

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

C.2. C.2 Byte String Wrappers

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

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

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

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

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

Appendix D. D. Implementation Conformance Matrix

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

The subsequent table shows the conditions.

Name	Reference	Implementation
Vendor Identifier	Section 8.4.8.2	REQUIRED
Class Identifier	Section 8.4.8.2	REQUIRED
Device Identifier	Section 8.4.8.2	OPTIONAL
Image Match	Section 8.4.9.2	REQUIRED
Component Slot	Section 8.4.9.3	OPTIONAL
Abort	Section 8.4.9.4	OPTIONAL
Custom Condition	Section 8.4.9.5	OPTIONAL

Table 14

The subsequent table shows the directives.

Name	Reference	Implementation
Set Component Index	Section 8.4.10.1	REQUIRED if more than one component
Try Each	Section 8.4.10.2	OPTIONAL
Override Parameters	Section 8.4.10.3	REQUIRED
Fetch	Section 8.4.10.4	REQUIRED for Updater
Copy	Section 8.4.10.5	OPTIONAL
Run	Section 8.4.10.6	REQUIRED for Bootloader
Run Sequence	Section 8.4.10.7	OPTIONAL
Swap	Section 8.4.10.8	OPTIONAL

Table 15

The subsequent table shows the parameters.

Name	Reference	Implementation
Vendor ID	Section 8.4.8.3	REQUIRED
Class ID	Section 8.4.8.4	REQUIRED
Image Digest	Section 8.4.8.6	REQUIRED
Image Size	Section 8.4.8.7	REQUIRED
Component Slot	Section 8.4.8.8	OPTIONAL
URI	Section 8.4.8.9	REQUIRED for Updater
Source Component	Section 8.4.8.10	OPTIONAL
Run Args	Section 8.4.8.11	OPTIONAL
Device ID	Section 8.4.8.5	OPTIONAL
Strict Order	Section 8.4.8.13	OPTIONAL
Soft Failure	Section 8.4.8.14	OPTIONAL
Custom	Section 8.4.8.15	OPTIONAL

Table 16

Authors' Addresses

Brendan Moran
Arm Limited

Email: Brendan.Moran@arm.com

Hannes Tschofenig
Arm Limited

Email: hannes.tschofenig@arm.com

Henk Birkholz
Fraunhofer SIT

Email: henk.birkholz@sit.fraunhofer.de

Koen Zandberg
Inria

Email: koen.zandberg@inria.fr