SUIT
Internet-Draft
Intended status: Standards Track
Expires: January 13, 2022

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

July 12, 2021

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

draft-ietf-suit-manifest-14

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on January 13, 2022.

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 [I-D.ietf-suit-information-model].

- Simple to parse on a constrained node

- Simple to process on a constrained node

- Compact encoding

- Comprehensible by an intermediate system

- Expressive enough to enable advanced use cases on advanced nodes

- Extensible

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

- a Firmware Author to reason about releasing a firmware.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2. Conventions and Terminology

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

Additionally, the following terminology is used throughout this document:

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

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

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

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

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

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

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

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

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

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

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

- Command: A Condition or a Directive.

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

- Directive: An action for the Recipient to perform.

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

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

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

- Report: A list of Records.

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

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

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

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

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

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

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

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

3. How to use this Document

This specification covers five aspects of firmware update:

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

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

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

- Section 7 describes the process of creating a Manifest.

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

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

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

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

4. Background

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

4.1. IoT Firmware Update Constraints

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

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

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

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

- small internal storage may require external storage support.

- multiple microcontrollers may require coordinated update of all applications.

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

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

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

4.2. SUIT Workflow Model

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

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

- All dependency manifests should be present before any payload is fetched.

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

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

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

- All dependencies and payloads must be validated prior to loading.

- All loaded images must be validated prior to execution.

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

1. Verify the signature of the manifest.

2. Verify the applicability of the manifest.

3. Resolve dependencies.

4. Fetch payload(s).

5. Install payload(s).

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

1. Verify image(s).

2. Load image(s).

3. Run image(s).

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

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

5. Metadata Structure Overview

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

The manifest is structured from several key components:

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

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

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

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

    1. Critical metadata, such as sequence number.

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

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

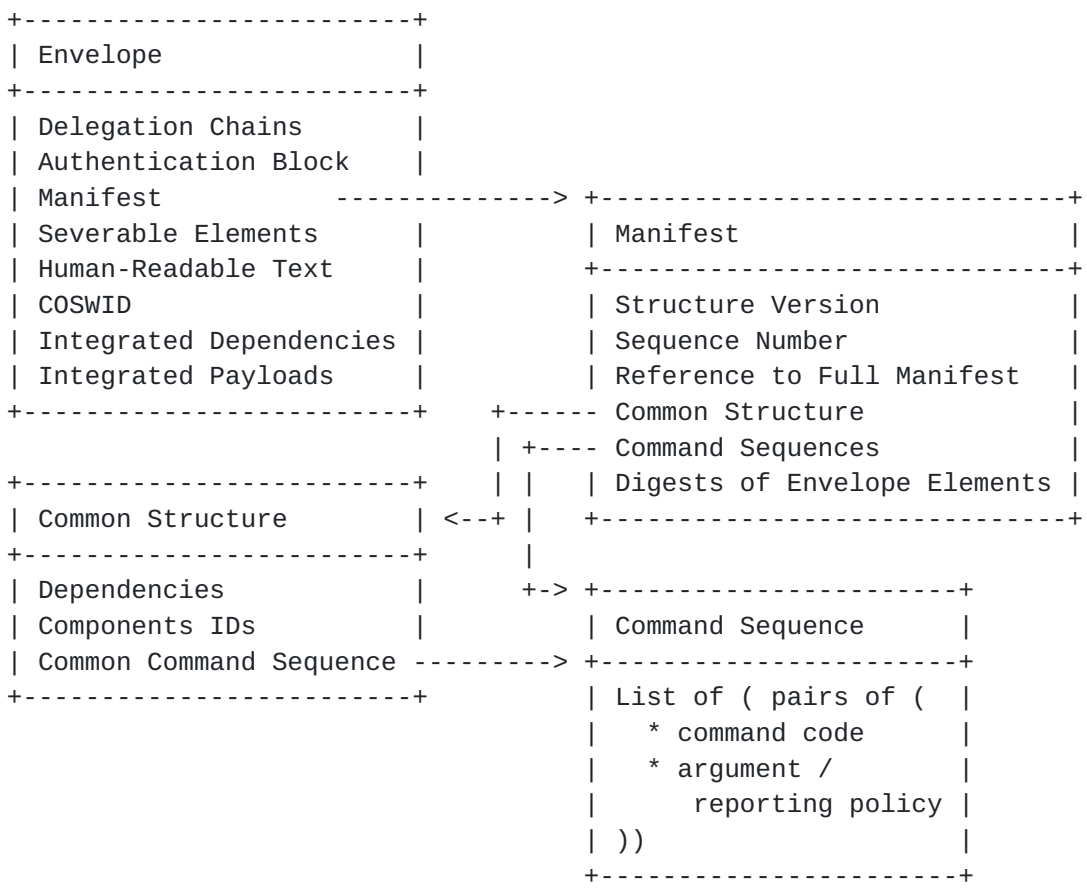
4. Integrity check values for severable elements.

5. Severable elements (see Section 5.5).

6. Integrated dependencies (see Section 5.6).

7. Integrated payloads (see Section 5.6).

The diagram below illustrates the hierarchy of the Envelope.



5.1. Envelope

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

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

See Section 8.2 for more detail.

5.2. Delegation Chains

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

See Section 8.3 for more detail.

5.3. Authentication Block

The Authentication Block contains a bstr-wrapped 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 8.4 for more detail.

5.4. Manifest

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

See Section 8.6 for more detail.

5.4.1. Critical Metadata

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

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

5.4.2. Common

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

See Section 8.7.2, and Section 8.7.2.1 for more detail.

5.4.3. Command Sequences

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

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

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

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

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

See Section 8.7.3 for more detail.

5.4.4. Integrity Check Values

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

See Section 8.7.9 for more detail.

5.4.5. Human-Readable Text

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

See Section 8.6.4 for more detail.

5.5. Severable Elements

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

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

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

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

See Section 8.8 for more detail.

5.6. Integrated Dependencies and Payloads

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

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

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

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

See Section 7.9.1, Section 8.5 for more detail.

6. Manifest Processor Behavior

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

6.1. Manifest Processor Setup

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

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

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

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

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

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

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

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

6.2. Required Checks

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

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

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

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

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

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

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

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

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

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

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

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

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

1. have sufficient permissions imparted by their signatures

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

The single dependent manifest is sometimes called a Root Manifest.

6.2.1. Minimizing Signature Verifications

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

- The Authentication Block contains a COSE_Sign_Tagged or COSE_Sign1_Tagged

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

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

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

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

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

6.3. Interpreter Fundamental Properties

The interpreter has a small set of design goals:

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

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

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

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

6.4. Abstract Machine Description

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

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

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

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

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

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

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

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

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

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

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

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

6.5. Special Cases of Component Index and Dependency Index

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

1. Integer

2. Array of integers

3. True

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

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

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

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

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

6.6. Serialized Processing Interpreter

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

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

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

6.7. Parallel Processing Interpreter

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

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

- Set Parameters.

- Override Parameters.

- Set Strict Order = True.

- Set Dependency Index.

- Set Component Index.

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

6.8. Processing Dependencies

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

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

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

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

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

6.9. Multiple Manifest Processors

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

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

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

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

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

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

7. Creating Manifests

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

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

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

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

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

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

7.1. Compatibility Check Template

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

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

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

7.2. Trusted Invocation Template

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

The following commands are placed into the common sequence:

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

Then, the run sequence contains the following commands:

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

7.3. Component Download Template

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

The following commands are placed into the common sequence:

- Set Component Index directive (see Section 8.7.7.1)

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

Then, the install sequence contains the following commands:

- Set Component Index directive (see Section 8.7.7.1)

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

- Fetch directive (see Section 8.7.7.7)

- Check Image Match condition (see Section 8.7.6.2)

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

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

7.4. Install Template

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

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

Then, the install sequence contains the following commands:

- Set Component Index directive (see Section 8.7.7.1)

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

- Copy directive (see Section 8.7.7.9)

- Check Image Match condition (see Section 8.7.6.2)

7.5. Install and Transform Template

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

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

The following commands are placed into the common sequence:

- Set Component Index directive for install component identifier (see Section 8.7.7.1)

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

Then, the install sequence contains the following commands:

- Set Component Index directive for install component identifier (see Section 8.7.7.1)

- Set Parameters directive (see Section 8.7.7.5) for:

  - Source Component for download component identifier (see Section 8.7.5.14)

  - Encryption Info (see Section 8.7.5.10)

  - Compression Info (see Section 8.7.5.11)

  - Unpack Info (see Section 8.7.5.12)

- Copy directive (see Section 8.7.7.9)

- Check Image Match condition (see Section 8.7.6.2)

7.6. Integrated Payload Template

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

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

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

7.7. Load from Nonvolatile Storage Template

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

The following commands are placed into the load sequence:

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

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

7.8. Load & Decompress from Nonvolatile Storage Template

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

The following commands are placed into the load sequence:

- Set Component Index directive (see Section 8.7.7.1)

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

- Copy directive (see Section 8.7.7.9)

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

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

7.9. Dependency Template

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

The following commands are placed into the dependency resolution sequence:

- Set Dependency Index directive (see Section 8.7.7.2)

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

- Fetch directive (see Section 8.7.7.7)

- Check Image Match condition (see Section 8.7.6.2)

- Process Dependency directive (see Section 8.7.7.4)

Then, the validate sequence contains the following operations:

- Set Dependency Index directive (see Section 8.7.7.2)

- Check Image Match condition (see Section 8.7.6.2)

- Process Dependency directive (see Section 8.7.7.4)

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

7.9.1. Composite Manifests

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

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

7.10. Encrypted Manifest Template

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

The following operations are placed into the dependency resolution block:

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

Then, the validate block contains the following operations:

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

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

7.11. A/B Image Template

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

The following commands are placed in the common block:

- Set Component Index directive (see Section 8.7.7.1)

- Try Each

  - First Sequence:

    * Override Parameters directive (see Section 8.7.7.6, Section 8.7.5) for Slot A

    * Check Slot Condition (see Section 8.7.6.5)

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

  - Second Sequence:

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

    * Check Slot Condition (see Section 8.7.6.5)

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

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

- Set Component Index directive (see Section 8.7.7.1)

- Try Each

  - First Sequence:

    * Override Parameters directive (see Section 8.7.7.6, Section 8.7.5) for Slot A

    * Check Slot Condition (see Section 8.7.6.5)

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

  - Second Sequence:

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

* Check Slot Condition (see Section 8.7.6.5)

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

- Fetch

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

NOTE: Any test can be used to select between images, Check 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, Delegation Chains, Authentication Information, Manifest, and Severable Elements.

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

8.1. Encoding Considerations

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

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

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

8.2. Envelope

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

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

8.3. Delegation Chains

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

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

8.4. Authenticated Manifests

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

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

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

8.5. Encrypted Manifests

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

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

8.6. Manifest

The manifest contains:

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

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

determine whether an element has been severed. The key used for a severable element is the same in the SUIT_Manifest and in the SUIT_Envelope so that a Recipient can easily identify the correct data in the envelope. See Section 8.7.9 for more detail.

8.6.1. suit-manifest-version

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

8.6.2. suit-manifest-sequence-number

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

8.6.3. suit-reference-uri

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

8.6.4. suit-text

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

- integer => text

- SUIT_Component_Identifier => map

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

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

manifests to be distributed without the text, since end-nodes do not require text. The meaning of each field is described below.

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

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

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

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

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

suit-text is OPTIONAL to implement.

8.7. text-version-required

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

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

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

8.7.1. suit-coswid

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

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

8.7.2. suit-common

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

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

suit-components is a list of SUIT_Component_Identifier (Section 8.7.2.2) blocks that specify the component identifiers that will be affected by the content of the current manifest. suit-components is REQUIRED to implement; at least one manifest in a dependency tree MUST contain a suit-components block.

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

8.7.2.1. Dependencies

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

The suit-dependency-digest specifies the dependency manifest uniquely by identifying a particular Manifest structure. This is identical to the digest that would be present as the payload of any suit-authentication-block in the dependency's Envelope. The digest is calculated over the Manifest structure instead of the COSE Sig_structure or Mac_structure. This is necessary to ensure that removing a signature from a manifest does not break dependencies due to missing signature elements. This is also necessary to support the trusted intermediary use case, where an intermediary re-signs the Manifest, removing the original signature, potentially with a different algorithm, or trading COSE_Sign for COSE_Mac.

The suit-dependency-prefix element contains a SUIT_Component_Identifier (see Section 8.7.2.2). This specifies the scope at which the dependency operates. This allows the dependency to be forwarded on to a component that is capable of parsing its own manifests. It also allows one manifest to be deployed to multiple dependent Recipients without those Recipients needing consistent component hierarchy. This element is OPTIONAL for Recipients to implement.

A dependency prefix can be used with a component identifier. This allows complex systems to understand where dependencies need to be applied. The dependency prefix can be used in one of two ways. The first simply prepends the prefix to all Component Identifiers in the dependency.

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

8.7.2.2. SUIT_Component_Identifier

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

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

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

8.7.3. SUIT_Command_Sequence

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

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

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

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

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

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

6. Run or Boot: suit-run is a SUIT_Command_Sequence to execute in order to run an image. suit-run typically contains a single instruction: either the "run" directive for the invocable manifest or the "process dependencies" directive for any

dependents of the invocable manifest. suit-run is OPTIONAL to implement.

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

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

Lists of commands are constructed from two kinds of element:

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

2. Directives that MUST be executed.

Each condition is composed of:

1. A command code identifier

2. A SUIT_Reporting_Policy (Section 8.7.4)

Each directive is composed of:

1. A command code identifier

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

Argument blocks are consumed only by flow-control directives:

- Set Component/Dependency Index

- Set/Override Parameters

- Try Each

- Run Sequence

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

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

For the purpose of setting the index, the two Component Identifier tables are considered to be concatenated together.

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

8.7.4. Reporting Policy

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

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

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

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

The information elements provided to the Reporting Engine are:

- The reporting policy

- The result of the command

- The values of parameters consumed by the command

- The system information consumed by the command

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

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

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

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

8.7.5. SUIT_Parameters

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

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

The defined manifest parameters are described below.

Name	CDDL Structure	Reference
Vendor ID	suit-parameter-vendor-identifier	Section 8.7.5 .3
Class ID	suit-parameter-class-identifier	Section 8.7.5 .4

Device ID	suit-parameter-device-identifier	Section 8.7.5 .5
Image Digest	suit-parameter-image-digest	Section 8.7.5 .6
Image Size	suit-parameter-image-size	Section 8.7.5 .7
Use Before	suit-parameter-use-before	Section 8.7.5 .8
Component Slot	suit-parameter-component-slot	Section 8.7.5 .9
Encryption Info	suit-parameter-encryption-info	Section 8.7.5 .10
Compression Info	suit-parameter-compression-info	Section 8.7.5 .11
Unpack Info	suit-parameter-unpack-info	Section 8.7.5 .12
URI	suit-parameter-uri	Section 8.7.5 .13
Source Component	suit-parameter-source-component	Section 8.7.5 .14
Run Args	suit-parameter-run-args	Section 8.7.5 .15
Minimum Battery	suit-parameter-minimum-battery	Section 8.7.5 .16
Update Priority	suit-parameter-update-priority	Section 8.7.5 .17
Version	suit-parameter-version	Section 8.7.5 .18
Wait Info	suit-parameter-wait-info	Section 8.7.5 .19
URI List	suit-parameter-uri-list	Section 8.7.5 .20

Fetch Arguments	suit-parameter-fetch-arguments	Section 8.7.5 .21
Strict Order	suit-parameter-strict-order	Section 8.7.5 .22
Soft Failure	suit-parameter-soft-failure	Section 8.7.5 .23
Custom	suit-parameter-custom	Section 8.7.5 .24

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

8.7.5.1. CBOR PEN UUID Namespace Identifier

The CBOR PEN UUID Namespace Identifier is constructed as follows:

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

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

Computing a type 5 UUID from these produces:

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

8.7.5.2. Constructing UUIDs

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

A Recipient MAY match any number of UUIDs for vendor or class identifier. This may be relevant to physical or software modules.

For example, a Recipient that has an OS and one or more applications might list one Vendor ID for the OS and one or more additional Vendor IDs for the applications. This Recipient might also have a Class ID that must be matched for the OS and one or more Class IDs for the applications.

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

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

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

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

1. Hardware model/revision

2. OS

3. WiFi module model/revision

4. Application

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

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

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

The RECOMMENDED method to create a vendor ID is:

Vendor ID = UUID5(DNS_PREFIX, vendor domain name)

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

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

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

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

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

- Model number.

- Hardware revision.

- Bootloader version (for immutable bootloaders).

8.7.5.3. suit-parameter-vendor-identifier

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

- A Private Enterprise Number

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

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

8.7.5.4. suit-parameter-class-identifier

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

8.7.5.5. suit-parameter-device-identifier

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

8.7.5.6. suit-parameter-image-digest

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

8.7.5.7. suit-parameter-image-size

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

8.7.5.8. suit-parameter-use-before

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

8.7.5.9. suit-parameter-component-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.

8.7.5.10. suit-parameter-encryption-info

Encryption Info defines the keys and algorithm information Fetch or Copy has to use to decrypt the confidentiality protected data. SUIT_Parameter_Encryption_Info is encoded as a COSE_Encrypt_Tagged structure wrapped in a bstr. A separate document will profile the COSE specification for use of manifest and firmware encrytion.

8.7.5.11. suit-parameter-compression-info

SUIT_Compression_Info defines any information that is required for a Recipient to perform decompression operations. SUIT_Compression_Info is a map containing this data. The only element defined for the map in this specification is the suit-compression-algorithm. This document defines the following suit-compression-algorithm's: ZLIB [RFC1950], Brotli [RFC7932], and ZSTD [RFC8878].

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

8.7.5.12. suit-parameter-unpack-info

SUIT_Unpack_Info defines the information required for a Recipient to interpret a packed format. This document defines the use of the following binary encodings: Intel HEX [HEX], Motorola S-record

[SREC], Executable and Linkable Format (ELF) [ELF], and Common Object File Format (COFF) [COFF].

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

8.7.5.13. suit-parameter-uri

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

8.7.5.14. suit-parameter-source-component

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

8.7.5.15. suit-parameter-run-args

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

8.7.5.16. suit-parameter-minimum-battery

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

8.7.5.17. suit-parameter-update-priority

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

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

8.7.5.18. suit-parameter-version

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

The component version can be compared as:

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

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

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

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

suit-condition-version is OPTIONAL to implement.

Versions SHOULD be provided as follows:

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

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

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

4. The fourth integer is the build number.

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

8.7.5.19. suit-parameter-wait-info

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

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

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

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

suit-wait-event-other-device-version reuses the encoding of suit-parameter-version-match. It is encoded as a sequence that contains an implementation-defined bstr identifier for the other device, and a list of one or more SUIT_Parameter_Version_Match.

8.7.5.20. suit-parameter-uri-list

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

8.7.5.21. suit-parameter-fetch-arguments

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

8.7.5.22. suit-parameter-strict-order

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

See Section 6.7 for behavioral description of Strict Order.

8.7.5.23. suit-parameter-soft-failure

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

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

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

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

8.7.5.24. suit-parameter-custom

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

8.7.6. SUIT_Condition

Conditions are used to define mandatory properties of a system in order for an update to be applied. They can be pre-conditions or post-conditions of any directive or series of directives, depending on where they are placed in the list. All Conditions specify a Reporting Policy as described Section 8.7.4. Conditions include:

Name	CDDL Structure	Reference
Vendor Identifier	suit-condition-vendor-identifier	Section 8.7.6 .1
Class Identifier	suit-condition-class-identifier	Section 8.7.6 .1
Device Identifier	suit-condition-device-identifier	Section 8.7.6 .1
Image Match	suit-condition-image-match	Section 8.7.6 .2
Image Not Match	suit-condition-image-not-match	Section 8.7.6 .3
Use Before	suit-condition-use-before	Section 8.7.6 .4
Component Slot	suit-condition-component-slot	Section 8.7.6 .5
Minimum Battery	suit-condition-minimum-battery	Section 8.7.6 .6
Update Authorized	suit-condition-update-authorized	Section 8.7.6 .7
Version	suit-condition-version	Section 8.7.6 .8
Abort	suit-condition-abort	Section 8.7.6 .9
Custom Condition	suit-condition-custom	Section 8.7.6 .10

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

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

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

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

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

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

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

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

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

8.7.6.2. suit-condition-image-match

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

8.7.6.3. suit-condition-image-not-match

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

8.7.6.4. suit-condition-use-before

Verify that the current time is BEFORE the specified time. suit-condition-use-before is used to specify the last time at which an update should be installed. The recipient evaluates the current time against the suit-parameter-use-before parameter (Section 8.7.5.8), which must have already been set as a parameter, encoded as seconds after 1970-01-01 00:00:00 UTC. Timestamp conditions MUST be evaluated in 64 bits, regardless of encoded CBOR size. suit-condition-use-before is OPTIONAL to implement.

8.7.6.5. suit-condition-component-slot

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

8.7.6.6. suit-condition-minimum-battery

suit-condition-minimum-battery provides a mechanism to test a Recipient's battery level before installing an update. This condition is primarily for use in primary-cell applications, where the battery is only ever discharged. For batteries that are charged, suit-directive-wait is more appropriate, since it defines a "wait" until the battery level is sufficient to install the update. suit-condition-minimum-battery is specified in mWh. suit-condition-minimum-battery is OPTIONAL to implement. suit-condition-minimum-battery consumes suit-parameter-minimum-battery (Section 8.7.5.16).

8.7.6.7. suit-condition-update-authorized

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

8.7.6.8. suit-condition-version

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

8.7.6.9. suit-condition-abort

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

8.7.6.10. suit-condition-custom

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

8.7.7. SUIT_Directive

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

Name	CDDL Structure	Reference
Set Component Index	suit-directive-set-component-index	Section 8.7.1
Set Dependency Index	suit-directive-set-dependency-index	Section 8.7.2
Try Each	suit-directive-try-each	Section 8.7.3
Process Dependency	suit-directive-process-dependency	Section 8.7.4
Set Parameters	suit-directive-set-parameters	Section 8.7.5
Override Parameters	suit-directive-override-parameters	Section 8.7.6
Fetch	suit-directive-fetch	Section 8.7.7
Fetch URI list	suit-directive-fetch-uri-list	Section 8.7.8
Copy	suit-directive-copy	Section 8.7.9
Run	suit-directive-run	Section 8.7.10
Wait For Event	suit-directive-wait	Section 8.7.11
Run Sequence	suit-directive-run-sequence	Section 8.7.12
Swap	suit-directive-swap	Section 8.7.13
Unlink	suit-directive-unlink	Section 8.8

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

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

8.7.7.1. suit-directive-set-component-index

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

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

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

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

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

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

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

8.7.7.2. suit-directive-set-dependency-index

Set Dependency Index defines the manifest to which successive directives and conditions will apply. The supplied argument MUST be either a boolean or an unsigned integer index into the dependencies, or an array of unsigned integer indices into the list of dependencies. If the following directives apply to ALL dependencies, then the boolean value "True" is used instead of an index. If the following directives apply to NO dependencies, then the boolean value

"False" is used. When suit-directive-set-component-index is used, suit-directive-set-dependency-index = False is implied. When suit-directive-set-dependency-index is used, suit-directive-set-component-index = False is implied.

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

See Section 6.5 for more details.

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

8.7.7.3. suit-directive-try-each

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

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

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

8.7.7.4. suit-directive-process-dependency

Execute the commands in the common section of the current dependency, followed by the commands in the equivalent section of the current dependency. For example, if the current section is "fetch payload,"

this will execute "common" in the current dependency, then "fetch payload" in the current dependency. Once this is complete, the command following suit-directive-process-dependency will be processed.

If the current dependency is False, this directive has no effect. If the current dependency is True, then this directive applies to all dependencies. If the current section is "common," then the command sequence MUST be terminated with an error.

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

8.7.7.5. suit-directive-set-parameters

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

Available parameters are defined in Section 8.7.5.

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

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

8.7.7.6. suit-directive-override-parameters

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

Available parameters are defined in Section 8.7.5.

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

8.7.7.7. suit-directive-fetch

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

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

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

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

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

8.7.7.8. suit-directive-fetch-uri-list

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

8.7.7.9. suit-directive-copy

suit-directive-copy instructs the manifest processor to obtain one or more payloads, as specified by the component index. As described in Section 6.5 component index may be a single integer, a list of integers, or True. suit-directive-copy retrieves each component specified by the current component-index, respectively. The current manifest's dependent-components are not automatically copied. In order to copy these, they MUST be specified in a component-index integer.

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

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

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

8.7.7.10. suit-directive-run

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

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

8.7.7.11. suit-directive-wait

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

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

8.7.7.12. suit-directive-run-sequence

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

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

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

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

8.7.7.13. suit-directive-swap

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

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

8.7.8. suit-directive-unlink

suit-directive-unlink marks the current component as unused in the current manifest. This can be used to remove temporary storage or remove components that are no longer needed. Example use cases:

- Temporary storage for encrypted download

- Temporary storage for verifying decompressed file before writing to flash

- Removing Trusted Service no longer needed by Trusted Application

Once the current Command Sequence is complete, the manifest processors checks each marked component to see whether any other manifests have referenced it. Those marked components with no other references are deleted. The manifest processor MAY choose to ignore a Unlink directive depending on device policy.

suit-directive-unlink is OPTIONAL to implement in manifest processors.

8.7.9. Integrity Check Values

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

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

8.8. Severable Elements

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

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

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

9. Access Control Lists

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

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

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

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

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

10. SUIT Digest Container

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 8.7.6.1	
2	Class Identifier	Section 8.7.6.1	
3	Image Match	Section 8.7.6.2	
4	Use Before	Section 8.7.6.4	
5	Component Slot	Section 8.7.6.5	
12	Set Component Index	Section 8.7.7.1	

13	Set Dependency Index	Section 8.7.7.2		
14	Abort			
15	Try Each	Section 8.7.7.3		
16	Reserved			
17	Reserved			
18	Process Dependency	suit-directive-process-dependency	Section 8.7.7.4	
19	Set Parameters	Section 8.7.7.5		
20	Override Parameters	Section 8.7.7.6		
21	Fetch	Section 8.7.7.7		
22	Copy	Section 8.7.7.9		
23	Run	Section 8.7.7.10		
24	Device Identifier	Section 8.7.6.1		
25	Image Not Match	Section 8.7.6.3		
26	Minimum Battery	Section 8.7.6.6		
27	Update Authorized	Section 8.7.6.7		
28	Version	Section 8.7.6.8		
29	Wait For Event	Section 8.7.7.11		
30	Fetch URI List	Section 8.7.7.8		
31	Swap	Section 8.7.7.13		

32	Run Sequence	Section 8.7.7.12	
33	Unlink	Section 8.7.8	
nint	Custom Condition	Section 8.7.6.10	

11.2. SUIT Parameters

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

11.3. SUIT Text Values

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

11.4. SUIT Component Text Values

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

11.5. SUIT Algorithm Identifiers

11.5.1. SUIT Compression Algorithm Identifiers

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

11.5.2. Unpack Algorithms

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

12. Security Considerations

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

13. Acknowledgements

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

- Milosch Meriac
- Geraint Luff
- Dan Ros
- John-Paul Stanford
- Hugo Vincent

- Carsten Bormann
- Oeyvind Roenningstad
- Frank Audun Kvamtroe
- Krzysztof Chruściński
- Andrzej Puzdrowski
- Michael Richardson
- David Brown
- Emmanuel Baccelli

14. References

14.1. Normative References

[I-D.ietf-cose-hash-algs]
Schaad, J., "CBOR Object Signing and Encryption (COSE): Hash Algorithms", draft-ietf-cose-hash-algs-09 (work in progress), September 2020.

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

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

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

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

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

14.2. Informative References

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

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

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

[I-D.ietf-cbor-tags-oid] Bormann, C., "Concise Binary Object Representation (CBOR) Tags for Object Identifiers", draft-ietf-cbor-tags-oid-06 (work in progress), March 2021.

[I-D.ietf-sacm-coswid] Birkholz, H., Fitzgerald-McKay, J., Schmidt, C., and D. Waltermire, "Concise Software Identification Tags", draft-ietf-sacm-coswid-17 (work in progress), February 2021.

[I-D.ietf-suit-architecture] Moran, B., Tschofenig, H., Brown, D., and M. Meriac, "A Firmware Update Architecture for Internet of Things", draft-ietf-suit-architecture-16 (work in progress), January 2021.

[I-D.ietf-suit-information-model] Moran, B., Tschofenig, H., and H. Birkholz, "A Manifest Information Model for Firmware Updates in IoT Devices", draft-ietf-suit-information-model-11 (work in progress), April 2021.

[I-D.ietf-teep-architecture] Pei, M., Tschofenig, H., Thaler, D., and D. Wheeler, "Trusted Execution Environment Provisioning (TEEP) Architecture", draft-ietf-teep-architecture-14 (work in progress), February 2021.

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

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

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

[RFC8392] Jones, M., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "CBOR Web Token (CWT)", RFC 8392, DOI 10.17487/RFC8392, May 2018, <https://www.rfc-editor.org/info/rfc8392>.

[RFC8747] Jones, M., Seitz, L., Selander, G., Erdtman, S., and H. Tschofenig, "Proof-of-Possession Key Semantics for CBOR Web Tokens (CWTs)", RFC 8747, DOI 10.17487/RFC8747, March 2020, <https://www.rfc-editor.org/info/rfc8747>.

[RFC8878] Collet, Y. and M. Kucherawy, Ed., "Zstandard Compression and the 'application/zstd' Media Type", RFC 8878, DOI 10.17487/RFC8878, February 2021, <https://www.rfc-editor.org/info/rfc8878>.

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

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

Appendix A. A. Full CDDL

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

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 1.4.

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

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

CWT = SUIT_Authentication_Block

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

SUIT_Digest = [
  suit-digest-algorithm-id : suit-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-dependency-resolution => bstr .cbor SUIT_Command_Sequence,
  ? suit-payload-fetch => bstr .cbor SUIT_Command_Sequence,
  ? suit-install => bstr .cbor SUIT_Command_Sequence,
  ? suit-text => bstr .cbor SUIT_Text_Map,
  ? suit-coswid => bstr .cbor concise-software-identity,
```

```
  * $$SUIT_severable-members-extensions,
)

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

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_Severable_Members_Choice,
    SUIT_Unseverable_Members,
    * $$SUIT_Manifest_Extensions,
}

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

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

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

SUIT_Dependencies         = [ + SUIT_Dependency ]
```

```
SUIT_Components           = [ + SUIT_Component_Identifier ]

concise-software-identity = any

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

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

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

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

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

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

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

SUIT_Wait_Events //= (suit-wait-event-authorization => int)
SUIT_Wait_Events //= (suit-wait-event-power => int)
SUIT_Wait_Events //= (suit-wait-event-network => int)
SUIT_Wait_Events //= (suit-wait-event-other-device-version
```

```
    => SUIT_Wait_Event_Argument_Other_Device_Version)
SUIT_Wait_Events //= (suit-wait-event-time => uint); Timestamp
SUIT_Wait_Events //= (suit-wait-event-time-of-day
    => uint); Time of Day (seconds since 00:00:00)
SUIT_Wait_Events //= (suit-wait-event-day-of-week
    => uint); Days since Sunday

SUIT_Wait_Event_Argument_Other_Device_Version = [
    other-device: bstr,
    other-device-version: [ + SUIT_Parameter_Version_Match ]
]

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

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

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

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

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

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

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

SUIT_Parameters //= (suit-parameter-uri-list =>
```

```
    bstr .cbor SUIT_URI_List)

RFC4122_UUID = bstr .size 16

SUIT_Parameter_Version_Match = [
    suit-condition-version-comparison-type:
        SUIT_Condition_Version_Comparison_Types,
    suit-condition-version-comparison-value:
        SUIT_Condition_Version_Comparison_Value
]
SUIT_Condition_Version_Comparison_Types /=
    suit-condition-version-comparison-greater
SUIT_Condition_Version_Comparison_Types /=
    suit-condition-version-comparison-greater-equal
SUIT_Condition_Version_Comparison_Types /=
    suit-condition-version-comparison-equal
SUIT_Condition_Version_Comparison_Types /=
    suit-condition-version-comparison-lesser-equal
SUIT_Condition_Version_Comparison_Types /=
    suit-condition-version-comparison-lesser

suit-condition-version-comparison-greater = 1
suit-condition-version-comparison-greater-equal = 2
suit-condition-version-comparison-equal = 3
suit-condition-version-comparison-lesser-equal = 4
suit-condition-version-comparison-lesser = 5

SUIT_Condition_Version_Comparison_Value = [+int]

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

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

SUIT_Compression_Algorithm_zlib = 1
SUIT_Compression_Algorithm_brotli = 2
SUIT_Compression_Algorithm_zstd = 3

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

}
```

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

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

SUIT_URI_List = [+ tstr ]

SUIT_Text_Map = {
    SUIT_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-version-required      => tstr,
    * $$suit-text-component-key-extensions
)

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

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

;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

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

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

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

suit-command-custom = nint

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

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

suit-directive-set-component-index      = 12
suit-directive-set-dependency-index     = 13
suit-directive-try-each                 = 15
suit-directive-process-dependency       = 18
suit-directive-set-parameters           = 19
suit-directive-override-parameters      = 20
suit-directive-fetch                    = 21
suit-directive-copy                     = 22
suit-directive-run                      = 23
```

```
suit-directive-wait                    = 29
suit-directive-fetch-uri-list          = 30
suit-directive-swap                    = 31
suit-directive-run-sequence            = 32
suit-directive-unlink                  = 33

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

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

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

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

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

suit-parameter-custom = nint

suit-compression-algorithm = 1

suit-unpack-algorithm  = 1

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

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

Appendix B. B. Examples

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

The examples are signed using the following ECDSA secp256r1 key:

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

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

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

Each example uses SHA256 as the digest function.

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

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

Command	Sys-Fail	Sys-Pass	Rec-Fail	Rec-Pass
suit-condition-vendor-identifier	1	1	1	1
suit-condition-class-identifier	1	1	1	1
suit-condition-image-match	1	1	1	1
suit-condition-component-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

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'a6c4590ac53043a98e8c4106e1e31b305516d7cf0a655eddfac6d45c810e036a'
            ]>>,
            signature: <<18([
                    / protected / <<{
                        / alg / 1:-7 / "ES256" /,
                    }>>,
                    / unprotected / {
                    },
                    / payload / F6 / nil /,
                    / signature / h'd11a2dd9610fb62a707335f58407922570
9f96e8117e7eeed98a2f207d05c8ecfba1755208f6abea977b8a6efe3bc2ca3215e119
```

```
  3be201467d052b42db6b7287'
                  ])>>
              ]
          ]>>,
          / 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 / 10:<<[
                  / condition-image-match / 3,15
              ]>>,
              / run / 12:<<[
                  / directive-run / 23,2
              ]>>,
          }>>,
      })
```

Total size of Envelope without COSE authentication object: 161

Envelope:

```
d86ba2025827815824822f5820a6c4590ac53043a98e8c4106e1e31b3055
16d7cf0a655eddfac6d45c810e036a035871a50101020003585fa2028181
41000458568614a40150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492
af1425695e48bf429b2d51f2ab45035824822f5820001122334455667788
99aabbccddeeff0123456789abcdeffedcba98765432100e1987d0010f02
0f0a4382030f0c43821702
```

Total size of Envelope with COSE authentication object: 237

Envelope with COSE authentication object:

```
d86ba2025873825824822f5820a6c4590ac53043a98e8c4106e1e31b3055
16d7cf0a655eddfac6d45c810e036a584ad28443a10126a0f65840d11a2d
d9610fb62a707335f584079225709f96e8117e7eeed98a2f207d05c8ecfb
a1755208f6abea977b8a6efe3bc2ca3215e1193be201467d052b42db6b72
87035871a50101020003585fa202818141000458568614a40150fa6b4a53
d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab45
035824822f58200011223344556778899aabbccddeeff0123456789abcd
effedcba98765432100e1987d0010f020f0a4382030f0c43821702
```

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'60c61d6eb7a1aaeddc49ce8157a55cff0821537eeee77a4ded44155b03045132'
            ]>>,
            signature: <<18([
                    / protected / <<{
                        / alg / 1:-7 / "ES256" /,
                    }>>,
                    / unprotected / {
                    },
                    / payload / F6 / nil /,
                    / signature / h'5249dacaf0ffc8326931b09586eb7e3769
e71a0e6a40ad8153db4980db9b05bd1742ddb46085fa11e62b65a79895c12ac7abe266
8ccc5afdd74466aed7bca389'
```

```
                ])>>
            ]
        ]>>,
        / 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
                ]>>,
            }>>,
            / install / 9:<<[
                / directive-set-parameters / 19,{
                    / uri / 21:'http://example.com/file.bin',
                } ,
                / directive-fetch / 21,2 ,
                / condition-image-match / 3,15
            ]>>,
            / validate / 10:<<[
                / condition-image-match / 3,15
            ]>>,
        }>>,
    })
```

Total size of Envelope without COSE authentication object: 196

Envelope:

```
d86ba2025827815824822f582060c61d6eb7a1aaeddc49ce8157a55cff08
21537eeee77a4ded44155b03045132035894a50101020103585fa2028181
41000458568614a40150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492
af1425695e48bf429b2d51f2ab45035824822f5820001122334455667788
99aabbccddeeff0123456789abcdeffedcba98765432100e1987d0010f02
0f0958258613a115781b687474703a2f2f6578616d706c652e636f6d2f66
696c652e62696e1502030f0a4382030f
```

Total size of Envelope with COSE authentication object: 272

Envelope with COSE authentication object:

```
d86ba2025873825824822f582060c61d6eb7a1aaeddc49ce8157a55cff08
21537eeee77a4ded44155b03045132584ad28443a10126a0f658405249da
caf0ffc8326931b09586eb7e3769e71a0e6a40ad8153db4980db9b05bd17
42ddb46085fa11e62b65a79895c12ac7abe2668ccc5afdd74466aed7bca3
89035894a50101020103585fa202818141000458568614a40150fa6b4a53
d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab45
035824822f58200011223344556778899aabbccddeeff0123456789abcd
effedcba98765432100e1987d0010f020f0958258613a115781b68747470
3a2f2f6578616d706c652e636f6d2f66696c652e62696e1502030f0a4382
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 5.5), and text (Section 8.6.4).

```
107({
        / authentication-wrapper / 2:<<[
            digest: <<[
                / algorithm-id / -16 / "sha256" /,
                / digest-bytes /
h'e45dcdb2074b951f1c88b866469939c2a83ed433a31fc7dfcb3f63955bd943ec'
            ]>>,
            signature: <<18([
                    / protected / <<{
                        / alg / 1:-7 / "ES256" /,
                    }>>,
```

```
                    / unprotected / {
                    },
                    / payload / F6 / nil /,
                    / signature / h'b4fd3a6a18fe1062573488cf24ac96ef9f
  30ac746696e50be96533b356b8156e4332587fe6f4e8743ae525d72005fddd4c1213d5
  5a8061b2ce67b83640f4777c'
                ])>>
            ]
        ]>>,
        / 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
                ]>>,
            }>>,
            / install / 9:[
                / algorithm-id / -16 / "sha256" /,
                / digest-bytes /
  h'3ee96dc79641970ae46b929ccf0b72ba9536dd846020dbdc9f949d84ea0e18d2'
            ],
            / validate / 10:<<[
                / condition-image-match / 3,15
            ]>>,
            / run / 12:<<[
                / directive-run / 23,2
            ]>>,
            / text / 13:[
                / algorithm-id / -16 / "sha256" /,
```

```
                / digest-bytes /
  h'2bfc4d0cc6680be7dd9f5ca30aa2bb5d1998145de33d54101b80e2ca49faf918'
            ],
        }>>,
        / install / 9:<<[
            / 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:

```
d86ba2025827815824822f5820e45dcdb2074b951f1c88b866469939c2a8
3ed433a31fc7dfcb3f63955bd943ec0358bba70101020203585fa2028181
41000458568614a40150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492
af1425695e48bf429b2d51f2ab45035824822f5820001122334455667788
99aabbccddeeff0123456789abcdeffedcba98765432100e1987d0010f02
0f09822f58203ee96dc79641970ae46b929ccf0b72ba9536dd846020dbdc
9f949d84ea0e18d20a4382030f0c438217020d822f58202bfc4d0cc6680b
e7dd9f5ca30aa2bb5d1998145de33d54101b80e2ca49faf918
```

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

Envelope:

```
d86ba2025873825824822f5820e45dcdb2074b951f1c88b866469939c2a8
3ed433a31fc7dfcb3f63955bd943ec584ad28443a10126a0f65840b4fd3a
6a18fe1062573488cf24ac96ef9f30ac746696e50be96533b356b8156e43
32587fe6f4e8743ae525d72005fddd4c1213d55a8061b2ce67b83640f477
7c0358bba70101020203585fa202818141000458568614a40150fa6b4a53
d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab45
035824822f58200011223344556778899aabbccddeeff0123456789abcd
effedcba98765432100e1987d0010f020f09822f58203ee96dc79641970a
e46b929ccf0b72ba9536dd846020dbdc9f949d84ea0e18d20a4382030f0c
438217020d822f58202bfc4d0cc6680be7dd9f5ca30aa2bb5d1998145de3
3d54101b80e2ca49faf918
```

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

Envelope with COSE authentication object:

```
d86ba4025873825824822f5820e45dcdb2074b951f1c88b866469939c2a8
3ed433a31fc7dfcb3f63955bd943ec584ad28443a10126a0f65840b4fd3a
6a18fe1062573488cf24ac96ef9f30ac746696e50be96533b356b8156e43
32587fe6f4e8743ae525d72005fddd4c1213d55a8061b2ce67b83640f477
7c0358bba70101020203585fa202818141000458568614a40150fa6b4a53
d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab45
035824822f58200011223344556778899aabbccddeeff0123456789abcd
effedcba98765432100e1987d0010f020f09822f58203ee96dc79641970a
e46b929ccf0b72ba9536dd846020dbdc9f949d84ea0e18d20a4382030f0c
438217020d822f58202bfc4d0cc6680be7dd9f5ca30aa2bb5d1998145de3
3d54101b80e2ca49faf91809583c8613a1157832687474703a2f2f657861
6d706c652e636f6d2f766572792f6c6f6e672f706174682f746f2f66696c
652f66696c652e62696e1502030f0d590204a20179019d2323204578616d
706c6520323a2053696d756c74616e656f757320446f776e6c6f61642c20
496e7374616c6c6174696f6e2c20536563757265520426f6f742c20536576
6572656420469656c64730a0a2020202054686973206578616d706c6520
636f766572732074686520666f6c6c6f77696e672074656d706c61746573
3a0a202020200a202020202a20436f6d706174696962696c69747920436865
636b20287b7b74656d706c6174652d636f6d706174696962696c6974792d63
6865636b7d7d290a202020202a20536563757265520426f6f7420287b7b74
656d706c6174652d7365637572652d626f6f747d7d290a202020202a2046
69726d7761726520446f776e6c6f616420287b7b6669726d776172652d64
6f776e6c6f61642d74656d706c6174657d7d290a202020200a2020202054
686973206578616d706c6520616c736f2064656d6f6e7374726174657320
736576572616 26c6520656c656d656e747320287b7b6f76722d73657665
7261626c657d7d292c20616e64207465787420287b7b6d616e6966657374
2d6469676573742d746578747d7d292e814100a2036761726d2e636f6d05
78525468697320636f6d706f6e656e74206973206120646 56d6f6e737472
6174696f6e2e20546865206469676573742069732061207361 6d706c6520
7061747465726e2c206e6f742061207265616c206f6e652e
```

B.4. Example 3: A/B images

This example covers the following templates:

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

```
107({
        / authentication-wrapper / 2:<<[
            digest: <<[
                / algorithm-id / -16 / "sha256" /,
                / digest-bytes /
h'7c9b3cb72c262608a42f944d59d659ff2b801c78af44def51b8ff51e9f45721b'
            ]>>,
            signature: <<18([
                    / protected / <<{
                        / alg / 1:-7 / "ES256" /,
                    }>>,
                    / unprotected / {
                    },
                    / payload / F6 / nil /,
                    / signature / h'e33d618df0ad21e609529ab1a876afb231
faff1d6a3189b5360324c2794250b87cf00cf83be50ea17dc721ca85393cd8e839a066
d5dec0ad87a903ab31ea9afa'
                ])>>
            ]
        ]>>,
        / 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,{
                                / offset / 5:33792,
                            } ,
                            / condition-component-offset / 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,{
                                / offset / 5:541696,
                            } ,
                            / condition-component-offset / 5,5 ,
                            / directive-override-parameters / 20,{
                                / image-digest / 3:<<[
                                    / algorithm-id / -16 / "sha256" /,
                                    / digest-bytes /
 h'0123456789abcdeffedcba98765432100011223344556677899aabbccddeeff'
                                ]>>,
                                / image-size / 14:76834,
                            }
                        ]>>
                    ] ,
                    / condition-vendor-identifier / 1,15 ,
                    / condition-class-identifier / 2,15
                ]>>,
            }>>,
            / install / 9:<<[
                / directive-try-each / 15,[
                    <<[
                        / directive-set-parameters / 19,{
                            / offset / 5:33792,
                        } ,
                        / condition-component-offset / 5,5 ,
                        / directive-set-parameters / 19,{
                            / uri / 21:'http://example.com/file1.bin',
                        }
                    ]>> ,
                    <<[
                        / directive-set-parameters / 19,{
                            / offset / 5:541696,
```

```
                        } ,
                        / condition-component-offset / 5,5 ,
                        / directive-set-parameters / 19,{
                            / uri / 21:'http://example.com/file2.bin',
                        }
                    ]>>
                ] ,
                / directive-fetch / 21,2 ,
                / condition-image-match / 3,15
            ]>>,
            / validate / 10:<<[
                / condition-image-match / 3,15
            ]>>,
        }>>,
    })
```

Total size of Envelope without COSE authentication object: 332

Envelope:

```
d86ba2025827815824822f58207c9b3cb72c262608a42f944d59d659ff2b
801c78af44def51b8ff51e9f45721b0359011ba5010102030358aaa20281
8141000458a18814a20150fa6b4a53d5ad5fdfbe9de663e4d41ffe025014
92af1425695e48bf429b2d51f2ab450f8258368614a105198400050514a2
035824822f58200011223344556778899aabbccddeeff0123456789abcd
effedcba98765432100e1987d0583a8614a1051a00084400050514a20358
24822f58200123456789abcdeffedcba9876543210001122334455667788
99aabbccddeeff0e1a00012c22010f020f095861860f82582a8613a10519
8400050513a115781c687474703a2f2f6578616d706c652e636f6d2f6669
6c65312e62696e582c8613a1051a00084400050513a115781c687474703a
2f2f6578616d706c652e636f6d2f66696c65322e62696e1502030f0a4382
030f
```

Total size of Envelope with COSE authentication object: 408

Envelope with COSE authentication object:

```
d86ba2025873825824822f58207c9b3cb72c262608a42f944d59d659ff2b
801c78af44def51b8ff51e9f45721b584ad28443a10126a0f65840e33d61
8df0ad21e609529ab1a876afb231faff1d6a3189b5360324c2794250b87c
f00cf83be50ea17dc721ca85393cd8e839a066d5dec0ad87a903ab31ea9a
fa0359011ba5010102030358aaa202818141000458a18814a20150fa6b4a
53d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab
450f8258368614a105198400050514a2035824822f582000112233445566
778899aabbccddeeff0123456789abcdeffedcba98765432100e1987d058
3a8614a1051a00084400050514a2035824822f58200123456789abcdeffe
dcba9876543210001122334455667788 99aabbccddeeff0e1a00012c2201
0f020f095861860f82582a8613a105198400050513a115781c687474703a
2f2f6578616d706c652e636f6d2f66696c65312e62696e582c8613a1051a
00084400050513a115781c687474703a2f2f6578616d706c652e636f6d2f
66696c65322e62696e1502030f0a4382030f
```

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

This example covers the following templates:

- Compatibility Check (Section 7.1)
- Secure Boot (Section 7.2)
- Firmware Download (Section 7.3)
- Install (Section 7.4)
- Load & Decompress (Section 7.8)

```
107({
        / authentication-wrapper / 2:<<[
            digest: <<[
                / algorithm-id / -16 / "sha256" /,
                / digest-bytes /
h'15736702a00f510805dcf89d6913a2cfb417ed414faa760f974d6755c68ba70a'
            ]>>,
            signature: <<18([
                    / protected / <<{
                        / alg / 1:-7 / "ES256" /,
                    }>>,
                    / unprotected / {
                    },
                    / payload / F6 / nil /,
                    / signature / h'3ada2532326d512132c388677798c24ffd
cc979bfae2a26b19c8c8bbf511fd7dd85f1501662c1a9e1976b759c4019bab44ba5434
efb45d3868aedbca593671f3'
                ])>>
            ]
```

```
        ]>>,
        / 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
                ]>>,
            }>>,
            / payload-fetch / 8:<<[
                / directive-set-component-index / 12,1 ,
                / directive-set-parameters / 19,{
                    / uri / 21:'http://example.com/file.bin',
                } ,
                / directive-fetch / 21,2 ,
                / condition-image-match / 3,15
            ]>>,
            / install / 9:<<[
                / directive-set-component-index / 12,0 ,
                / directive-set-parameters / 19,{
                    / source-component / 22:1 / [h'02'] /,
                } ,
                / directive-copy / 22,2 ,
                / condition-image-match / 3,15
            ]>>,
            / validate / 10:<<[
                / directive-set-component-index / 12,0 ,
```

```
                / condition-image-match / 3,15
            ]>>,
            / load / 11:<<[
                / directive-set-component-index / 12,2 ,
                / directive-set-parameters / 19,{
                    / image-digest / 3:<<[
                        / algorithm-id / -16 / "sha256" /,
                        / digest-bytes /
  h'0123456789abcdeffedcba98765432100011223344556677889 9aabbccddeeff'
                    ]>>,
                    / image-size / 14:76834,
                    / source-component / 22:0 / [h'00'] /,
                    / compression-info / 19:<<{
                        / compression-algorithm / 1:1 / "gzip" /,
                    }>>,
                } ,
                / directive-copy / 22,2 ,
                / condition-image-match / 3,15
            ]>>,
            / run / 12:<<[
                / directive-set-component-index / 12,2 ,
                / directive-run / 23,2
            ]>>,
        }>>,
    })
```

Total size of Envelope without COSE authentication object: 292

Envelope:

```
d86ba2025827815824822f582015736702a00f510805dcf89d6913a2cfb4
17ed414faa760f974d6755c68ba70a0358f4a801010204035867a2028381
4100814102814101045858880c0014a40150fa6b4a53d5ad5fdfbe9de663
e4d41ffe02501492af1425695e48bf429b2d51f2ab45035824822f582000
112233445566778899aabbccddeeff0123456789abcdeffedcba98765432
100e1987d0010f020f085827880c0113a115781b687474703a2f2f657861
6d706c652e636f6d2f66696c652e62696e1502030f094b880c0013a11601
1602030f0a45840c00030f0b583d880c0213a4035824822f582001234567
89abcdeffedcba98765432100011223344556677889 9aabbccddeeff0e1a
00012c221343a10101160016020 30f0c45840c021702
```

Total size of Envelope with COSE authentication object: 368

Envelope with COSE authentication object:

```
d86ba2025873825824822f582015736702a00f510805dcf89d6913a2cfb4
17ed414faa760f974d6755c68ba70a584ad28443a10126a0f658403ada25
32326d512132c388677798c24ffdcc979bfae2a26b19c8c8bbf511fd7dd8
5f1501662c1a9e1976b759c4019bab44ba5434efb45d3868aedbca593671
f30358f4a801010204035867a202838141008141028141010458588 0c00
14a40150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492af1425695e48
bf429b2d51f2ab45035824822f582000112233445566778899aabbccddee
ff0123456789abcdeffedcba98765432100e1987d0010f020f085827880c
0113a115781b687474703a2f2f6578616d706c652e636f6d2f66696c652e
62696e1502030f094b880c0013a116011602030f0a45840c00030f0b583d
880c0213a4035824822f58200123456789abcdeffedcba98765432100011
2233445566778899aabbccddeeff0e1a00012c221343a101011600160203
0f0c45840c021702
```

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'd1e73f16e4126007bc4d804cd33b0209fbab34728e60ee8c00f3387126748dd2'
            ]>>,
            signature: <<18([
                    / protected / <<{
                        / alg / 1:-7 / "ES256" /,
                    }>>,
                    / unprotected / {
                    },
                    / payload / F6 / nil /,
                    / signature / h'b7ae0a46a28f02e25cda6d9a255bbaf863
30141831fae5a78012d648bc6cee55102e0f1890bdeacc3adaa4fae0560f83a45eecae
65cabce642f56d84ab97ef8d'
                ])>>
            ]
        ]>>,
        / 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'0123456789abcdeffedcba98765432100011223344556677889aabbccddeeff'
                        ]>>,
                        / image-size / 14:76834,
                    }
                ]>>,
            }>>,
            / install / 9:<<[
                / directive-set-component-index / 12,0 ,
                / directive-set-parameters / 19,{
                    / uri / 21:'http://example.com/file1.bin',
                } ,
                / directive-fetch / 21,2 ,
                / condition-image-match / 3,15 ,
                / directive-set-component-index / 12,1 ,
                / directive-set-parameters / 19,{
                    / uri / 21:'http://example.com/file2.bin',
                } ,
                / directive-fetch / 21,2 ,
                / condition-image-match / 3,15
```

```
            ]>>,
            / validate / 10:<<[
                / directive-set-component-index / 12,0 ,
                / condition-image-match / 3,15 ,
                / directive-set-component-index / 12,1 ,
                / condition-image-match / 3,15
            ]>>,
            / run / 12:<<[
                / directive-set-component-index / 12,0 ,
                / directive-run / 23,2
            ]>>,
        }>>,
    })
```

Total size of Envelope without COSE authentication object: 306

Envelope:

```
d86ba2025827815824822f5820d1e73f16e4126007bc4d804cd33b0209fb
ab34728e60ee8c00f3387126748dd203590101a601010205035895a20282
8141008141010458898c0c0014a40150fa6b4a53d5ad5fdfbe9de663e4d4
1ffe02501492af1425695e48bf429b2d51f2ab45035824822f5820001122
33445566778899aabbccddeeff0123456789abcdeffedcba98765432100e
1987d0010f020f0c0114a2035824822f58200123456789abcdeffedcba98
76543210001122334455667788 99aabbccddeeff0e1a00012c2209584f90
0c0013a115781c687474703a2f2f6578616d706c652e636f6d2f66696c65
312e62696e1502030f0c0113a115781c687474703a2f2f6578616d706c65
2e636f6d2f66696c65322e62696e1502030f0a49880c00030f0c01030f0c
45840c001702
```

Total size of Envelope with COSE authentication object: 382

Envelope with COSE authentication object:

```
d86ba2025873825824822f5820d1e73f16e4126007bc4d804cd33b0209fb
ab34728e60ee8c00f3387126748dd2584ad28443a10126a0f65840b7ae0a
46a28f02e25cda6d9a255bbaf86330141831fae5a78012d648bc6cee5510
2e0f1890bdeacc3adaa4fae0560f83a45eecae65cabce642f56d84ab97ef
8d03590101a601010205035895a202828141008141010458898c0c0014a4
0150fa6b4a53d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf42
9b2d51f2ab45035824822f582000112233445566778899aabbccddeeff01
23456789abcdeffedcba98765432100e1987d0010f020f0c0114a2035824
822f58200123456789abcdeffedcba9876543210001122334455667788 99
aabbccddeeff0e1a00012c2209584f900c0013a115781c687474703a2f2f
6578616d706c652e636f6d2f66696c65312e62696e1502030f0c0113a115
781c687474703a2f2f6578616d706c652e636f6d2f66696c65322e62696e
1502030f0a49880c00030f0c01030f0c45840c001702
```

Appendix C. C. Design Rational

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

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

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

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

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

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

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

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

C.1. C.1 Design Rationale: Envelope

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

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

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

3. This enables multiple authentication schemes.

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

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

Four authentication objects are supported by the Envelope:

- COSE_Sign_Tagged

- COSE_Sign1_Tagged

- COSE_Mac_Tagged

- COSE_Mac0_Tagged

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

C.2. C.2 Byte String Wrappers

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

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

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

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

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

Appendix D. D. Implementation Conformance Matrix

This section summarizes the functionality a minimal 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 8.7.5.2	REQUIRED
Class Identifier	Section 8.7.5.2	REQUIRED
Device Identifier	Section 8.7.5.2	OPTIONAL
Image Match	Section 8.7.6.2	REQUIRED
Image Not Match	Section 8.7.6.3	OPTIONAL
Use Before	Section 8.7.6.4	OPTIONAL
Component Slot	Section 8.7.6.5	OPTIONAL
Abort	Section 8.7.6.9	OPTIONAL
Minimum Battery	Section 8.7.6.6	OPTIONAL
Update Authorized	Section 8.7.6.7	OPTIONAL
Version	Section 8.7.6.8	OPTIONAL
Custom Condition	Section 8.7.6.10	OPTIONAL

The subsequent table shows the directives.

Name	Reference	Implementation
Set Component Index	Section 8.7.7. 1	REQUIRED if more than one component
Set Dependency Index	Section 8.7.7. 2	REQUIRED if dependencies used
Try Each	Section 8.7.7. 3	OPTIONAL
Process Dependency	Section 8.7.7. 4	OPTIONAL
Set Parameters	Section 8.7.7. 5	OPTIONAL
Override Parameters	Section 8.7.7. 6	REQUIRED
Fetch	Section 8.7.7. 7	REQUIRED for Updater
Copy	Section 8.7.7. 9	OPTIONAL
Run	Section 8.7.7. 10	REQUIRED for Bootloader
Wait For Event	Section 8.7.7. 11	OPTIONAL
Run Sequence	Section 8.7.7. 12	OPTIONAL
Swap	Section 8.7.7. 13	OPTIONAL
Fetch URI List	Section 8.7.7. 8	OPTIONAL
Unlink	Section 8.7.8	OPTIONAL

The subsequent table shows the parameters.

Name	Reference	Implementation
Vendor ID	Section 8.7.5.3	REQUIRED
Class ID	Section 8.7.5.4	REQUIRED
Image Digest	Section 8.7.5.6	REQUIRED
Image Size	Section 8.7.5.7	REQUIRED
Use Before	Section 8.7.5.8	RECOMMENDED
Component Slot	Section 8.7.5.9	OPTIONAL
Encryption Info	Section 8.7.5.10	RECOMMENDED
Compression Info	Section 8.7.5.11	RECOMMENDED
Unpack Info	Section 8.7.5.12	RECOMMENDED
URI	Section 8.7.5.13	REQUIRED for Updater
Source Component	Section 8.7.5.14	OPTIONAL
Run Args	Section 8.7.5.15	OPTIONAL
Device ID	Section 8.7.5.5	OPTIONAL
Minimum Battery	Section 8.7.5.16	OPTIONAL
Update Priority	Section 8.7.5.17	OPTIONAL
Version Match	Section 8.7.5.18	OPTIONAL
Wait Info	Section 8.7.5.19	OPTIONAL
URI List	Section 8.7.5.20	OPTIONAL
Strict Order	Section 8.7.5.22	OPTIONAL
Soft Failure	Section 8.7.5.23	OPTIONAL
Custom	Section 8.7.5.24	OPTIONAL

Authors' Addresses

Brendan Moran
Arm Limited

EMail: Brendan.Moran@arm.com

Hannes Tschofenig
Arm Limited

EMail: hannes.tschofenig@arm.com

Henk Birkholz
Fraunhofer SIT

EMail: henk.birkholz@sit.fraunhofer.de

Koen Zandberg
Inria

EMail: koen.zandberg@inria.fr