SUIT
Internet-Draft
Intended status: Standards Track
Expires: 23 January 2026

B. Moran
Arm Limited
K. Takayama
SECOM CO., LTD.

22 July 2025

Software Update for the Internet of Things (SUIT) Manifest Extensions for Multiple Trust Domain
draft-ietf-suit-trust-domains-12

Abstract

A device has more than one trust domain when it enables delegation of different rights to mutually distrusting entities for use for different purposes or Components in the context of firmware or software update. This specification describes extensions to the Software Update for the Internet of Things (SUIT) Manifest format for use in deployments with multiple trust domains.

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 23 January 2026.

Copyright Notice

Table of Contents

1. Introduction

Devices that require more advanced configurations than a Manifest signed by a single authority also require more complex rules for deploying software updates. For example, devices may require:

* Components from multiple software signing authorities
* a mechanism to remove an unneeded Component
* Dependencies delivered in the same envelope as the Manifest
* a partly encrypted Manifest so that distribution does not reveal private information
* installation performed by a different execution mode than payload fetch

Devices implementing this specification typically partition their software, dividing it, according to physical or logical features, into multiple "domains" with different requirements for authorities: multiple trust domains. Because of the more complex use cases that are typically targetted by devices implementing this specification, the applicable device class is typically Class 2 or higher and often isolation level Is8, for example Arm TrustZone for Cortex-M, as described in [I-D.ietf-iotops-7228bis].

Dependencies enable several additional use cases. In particular, they enable two or more entities who are trusted for different privileges to coordinate. This can be used in many scenarios. For example:

* Devices with network interface controllers (NICs), including radios, may contain secondary processors in the NICs in addition to the device primary processor. These two processors may have separate Software with separate signing authorities. Dependencies allow the Manifest for the primary processor to reference a Manifest signed by a different authority.

* A network operator may wish to provide local caching of Update Payloads. The network creates a Dependent Manifest that provides a different URI for any Payloads they wish to cache the parameter override mechanism described in Section 5.6.1.

* A Device Administrator provides a device with some additional configuration. The Device Administrator wants to test their configuration with each new Software version before releasing it. The configuration is delivered as a binary in the same way as a Software Image. The Device Administrator references the Software Manifest from the Software author in their own Manifest which also defines the configuration.

* An Author wants to entrust a Distributor to provide devices with firmware decryption keys, but not permit the Distributor to sign code. Dependencies allow the Distributor to deliver a device's decryption information without also granting code signing authority.

* A Trusted Application Manager (TAM) wants to distribute personalisation information to a Trusted Execution Environment in addition to a Trusted Application (TA), but does not have code signing authority (see [RFC9397], Section 2). Dependencies enable the TAM to construct an update containing the personalisation information and a dependency on the TA, but leaves the TA signed by the TA's Author.

When a system has multiple trust domains, each domain might require independent verification of authenticity or security policies. Trust domains might be divided by separation technology such as Arm TrustZone, Intel SGX, or another Trusted Execution Environment (TEE) technology. Trust 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.

Dependencies enable Components such as Software, configuration, and other Resource data authenticated by different Trust Anchors to be delivered to devices.

These mechanisms are not part of the core Manifest specification ([I-D.ietf-suit-manifest]), but they are needed for more advanced use cases, such as the architecture described in [RFC9397].

This specification extends the SUIT Manifest specification ([I-D.ietf-suit-manifest]) with:

* Integrated Components
* Dependencies
* Manifest Component Identifier
* Candidate Verification
* Parameter Override support
* Uninstall support

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.

The terminology from [I-D.ietf-suit-manifest], Section 2 and [RFC9397], Section 2 is used in this specification. Additionally, the following terminology is used:

* Dependency: A Manifest that is required by a second Manifest in order for operations described by the second Manifest to complete successfully.
* Dependent: A Manifest that depends on another Mani
* Root Manifest: A manifest that has no dependents and, combined with all Dependencies (recursively) specifies a complete Component Set.
* Staging Procedure: A procedure that fetches dependencies and images referenced by an Update and stores them to a Staging Area.
* Installation Procedure: A procedure that installs dependencies and images stored in a Staging Area; copying (and optionally, transforming them) into an active Image storage location.
* Staging Area: A Component or group of Components that are used for transient storage of Images between fetch and installation. Images in this area are opaque, except for use by the Installation Procedure.

* Reference Count: An implementation-defined mechanism to track the number of manifests that refer to another manifest.

3. Changes to SUIT Workflow Model

The use of the features presented for use with multiple trust domains requires some augmentation of the workflow presented in the SUIT Manifest specification ([I-D.ietf-suit-manifest]):

One additional assumption is added to the list of assumptions for the Update Procedure in [I-D.ietf-suit-manifest], Section 4.2:

* All Dependencies must be fetched and integrity checked before any Payload is fetched.

One additional assumption is added to the list of assumptions for the Invocation Procedure in [I-D.ietf-suit-manifest], Section 4.2:

* All Dependencies must be validated prior to loading.

Steps 3 and 5 are added to the expected installation workflow of a Recipient:

1. Verify the signature of the Manifest.
2. Verify the applicability of the Manifest.
3. Resolve Dependencies.
4. Fetch Payload(s).
5. Verify Candidate Component Set.
6. Install Payload(s).
7. Verify image(s).

In addition, 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. The lockstep process is described in Section 5.4.

4. Changes to Manifest Metadata Structure

To accommodate the additional metadata needed to enable these features, the Envelope and Manifest are augmented with several new elements:

* Envelope
  - Integrated Dependency
* Manifest
  - Common
    o Dependency Metadata
  - Component Identifier
  - Dependency Resolution SUIT_Command_Sequence
  - Candidate Verification SUIT_Command_Sequence

In addition several new SUIT_Commands are added:

* SUIT Conditions
  - Dependency Integrity Check
  - Component Is Dependency Check
* SUIT Directives
  - Process Dependency
  - Set Parameters
  - Unlink

The Envelope gains two more elements: Integrated Dependencies and Integrated Payloads. The Common metadata section in the Manifest also gains a list of Dependencies.

The new metadata structure is shown below.

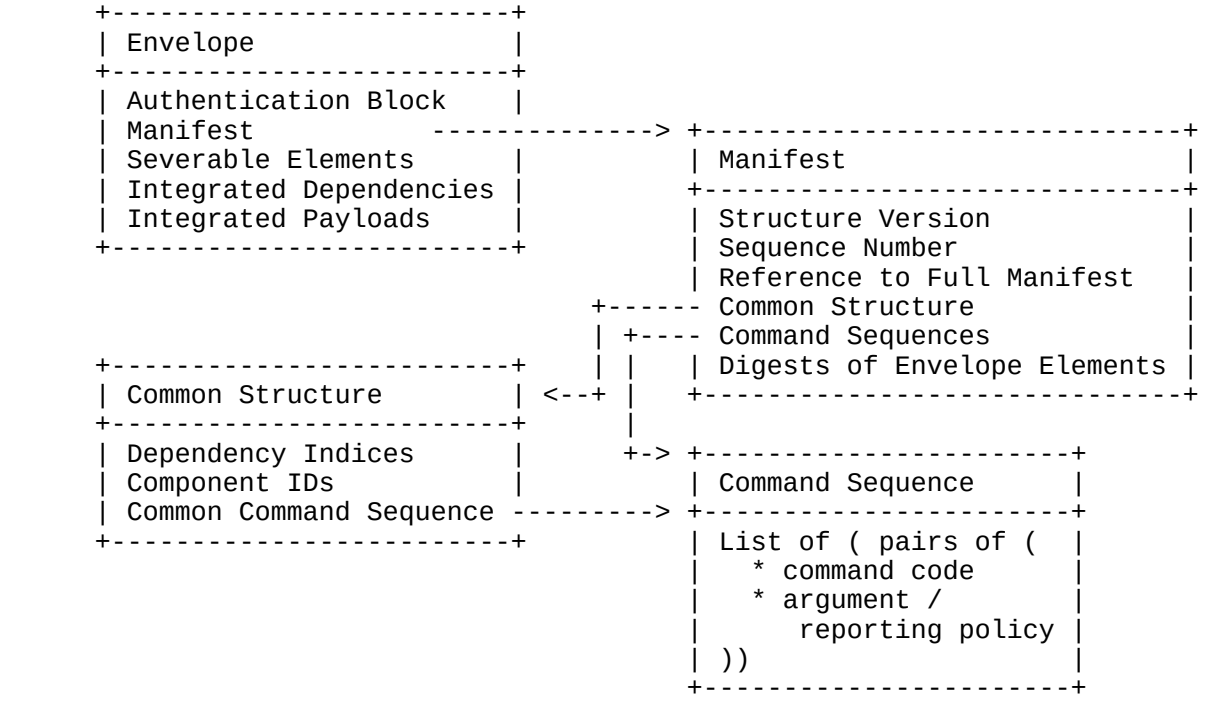

Figure 1: SUIT Metadata Structure

This is an update of the figure in Section 4.2 of [I-D.ietf-suit-manifest]

5. Dependencies

A Dependency is another SUIT_Envelope ([I-D.ietf-suit-manifest], section 8.2) that describes additional Components.

5.1. Changes to Required Checks

This section augments the definitions in Required Checks ([I-D.ietf-suit-manifest], Section 6.2).

More checks are required when handling Dependencies. By default, any signature of a Dependency MUST be verified. However, there are some exceptions to this rule: where a device supports only one level of access (no ACLs, [I-D.ietf-suit-manifest], Section 9, declaring which authorities have access to different Components/Commands/Parameters), it MAY choose to skip signature verification of Dependencies, since they are verified by digest. Where a device differentiates between trust levels, such as with an ACL, 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, if a dependent's signer has access rights to all Components specified in a Dependency, then that Dependency 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.

Components that should be treated as Dependencies are identified in the suit-common metadata (Section 5.2).

Any required check that fails MUST result in an Abort.

Prior to executing any Command Sequence:

1. If the interpreter does not support Dependencies and a Manifest specifies a Dependency, then the interpreter MUST Abort.

2. If the Manifest contains more than one Component and/or Dependency, each Command sequence MUST begin with a Set Component Index Command.

If a Dependency is specified, then the Manifest Processor MUST perform the following additional checks:

1. Prior to executing any Command Sequence: the dependent MUST populate all Command Sequences for each Procedure specified by the Dependency; either the Staging Procedure, the Update Procedure, the Installation Procedure, or the Invocation Procedure.

2. At the end of each section in the dependent: The corresponding section in each Dependency has been executed, if present.

If a Recipient supports groups of interdependent Components (a Component Set), then prior to fetching any payload, it SHOULD verify that all Components in the Component Set are specified by 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.

Failing to verify the availablility of all components may lead to API mismatches and other version mismatch problems.

The single dependent Manifest is called a Root Manifest.

5.2. Changes to Manifest Structure

This section augments the Manifest Structure (Section 8.4) in [I-D.ietf-suit-manifest].

5.2.1. Manifest Component ID

In complex systems, it may not always be clear where the Root Manifest is stored; this is particularly complex when a system has multiple, independent Root Manifests. The Manifest Component ID resolves this contention. The manifest-component-id is intended to be used by the Root Manifest. The manifest-component-id is only used when storing a Root Manifest. The manifest-component-id is ignored when processing Dependencies.

The following CDDL (see [RFC8610]) describes the Manifest Component ID:

```
$$SUIT_Manifest_Extensions //=
    (suit-manifest-component-id => SUIT_Component_Identifier)
```

5.2.2. SUIT_Dependencies Manifest Element

The suit-common section, as described in [I-D.ietf-suit-manifest], Section 8.4.5 is extended with a map of Component indices that indicate a Dependency. The keys of the map are the Component indices and the values of the map are any extra metadata needed to describe those Dependencies.

Because some operations treat Dependencies differently from other Components, it is necessary to identify them. SUIT_Dependencies identifies which Components from suit-components ([I-D.ietf-suit-manifest], Section 8.4.5) are to be treated as the SUIT_Envelope of a Dependency. SUIT_Dependencies is a map of Components, referenced by Component Index. Optionally, a Component prefix or other metadata may be delivered with the Component index. The CDDL for suit-dependencies is shown below:

```
$$SUIT_Common-extensions //= (
    suit-dependencies => SUIT_Dependencies
)
SUIT_Dependencies = {
    + uint => SUIT_Dependency_Metadata
}
SUIT_Dependency_Metadata = {
    ? suit-dependency-prefix => SUIT_Component_Identifier
    * $$SUIT_Dependency_Extensions
}
```

If no extended metadata is needed for an extension, SUIT_Dependency_Metadata is an empty map (this is the same encoding size as a null). SUIT_Dependencies MUST be sorted according to Core Deterministic Encoding Requirements ([RFC8949], Section 4.2.1).

The Components specified by SUIT_Dependency_Metadata will contain a Manifest Envelope 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.

When executing suit-condition-image-match over a Component that is designated in SUIT_Dependency_Metadata, the digest MUST be computed over just the bstr-wrapped SUIT_Manifest contained in the Manifest Envelope designated by the Component Index. This enables a Dependency reference to uniquely identify a particular Manifest structure. This is identical to the digest that is present as the first element of the suit-authentication-block in the Dependency's Envelope. The digest is calculated over the Manifest structure 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 ([I-D.ietf-suit-manifest], Section 8.4.5.1). 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. suit-dependency-prefix 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 needs to be processed by a secondary Manifest Processor, as described in Section 5.4.1.

5.3. Changes to Abstract Machine Description

This section augments the Abstract Machine Description in [I-D.ietf-suit-manifest], Section 6.4. With the addition of Dependencies, some changes are necessary to the abstract machine, outside the typical scope of added Commands. These changes alter the behaviour of an existing Command and way that the parser processes Manifests:

* Five new Commands are introduced in Section 5.6:
  - Set Parameters
  - Process Dependency
  - Is Dependency
  - Dependency Integrity
  - Unlink
* Dependencies have Component Identifiers. All Commands may target Dependencies as well as Components, with one exception: suit-directive-process-dependency. Future commands MAY define their own restrictions on applicability to Dependencies and non-Dependency Components.
* Dependencies are processed in lockstep with the Root Manifest. This means that every Dependency’s current Command sequence must be executed before a dependent’s later Command sequence may be executed. For example, every Dependency’s Dependency Resolution step must be executed before any dependent’s Payload fetch step.
* When a Manifest Processor supports multiple independent Components, they may have shared Dependencies.
* When a Manifest Processor supports shared Dependencies, it MUST support reference counting of those Dependencies.

* When reference counting is used, Components MUST NOT be overwritten. The Manifest Uninstall section must be called, then the component MUST be Unlinked.

5.4. Processing Dependencies

As described in Section 5.1, the Manifest Processor must ensure that a Manifest with Dependencies invokes suit-directive-process-dependency for 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 Manifest Processor:

1. Checks whether the map of Dependencies contains an entry for the current Component Index. If not present, it causes an immediate Abort.

2. Checks whether the Dependency has been the target of a Dependency integrity check. If not, it causes an immediate Abort.

3. Performs any application-specific setup that is required to parse the specified Component as a SUIT_Envelope of a Dependency.

4. Authenticates the Dependency.

5. Executes the common-sequence section of the Dependency.

6. 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 interpreter also performs the checks described in Section 5.1 to ensure that the dependent is processing the Dependency correctly.

5.4.1. Multiple Manifest Processors

When there are two or more trust domains, 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 of [I-D.ietf-suit-manifest]. 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, and 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 trust 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 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.

5.5. Dependency Resolution

The Dependency Resolution Command Sequence is a container for the Commands needed to acquire and process the Dependencies of the current Manifest. All Dependencies MUST be fetched before any Payload is fetched to ensure that all Manifests are available and authenticated before any of the (larger) Payloads are acquired.

5.6. Added and Modified Commands

All Commands are modified in that they can also target Dependencies. However, Set Component Index has a larger modification.

Command Name	Semantic of the Operation
Set Parameters	current.params[k] := v if not k in current.params for-each k,v in arg
Process Dependency	exec(current[common]); exec(current[current-segment])
Dependency Integrity	verify(current, current.params[image-digest])
Is Dependency	assert(current exists in Dependencies)
Unlink	unlink(current)

Table 1: Added/Modified Abstract Machine Commands

5.6.1. suit-directive-set-parameters

Similar to suit-directive-override-parameters ([I-D.ietf-suit-manifest], section 8.4.10.3), suit-directive-set-parameters allows the Manifest to configure behavior of future Directives by changing Parameters that are read by those Directives. Set Parameters is for use when Dependencies are used because it allows a Manifest to modify the behavior of its Dependencies. Because of this modification behavior, suit-directive-set-parameters MUST only be used for parameters that are intended to be overridden.

Available Parameters are defined in [I-D.ietf-suit-manifest], section 8.4.8.

If a Parameter is already set, suit-directive-set-parameters will skip setting the Parameter to its argument. This enables parameter replacement in Manifest trees. A Dependency can specify a default Parameter using suit-directive-set-parameters. Then, a dependent of that Dependency can use suit-directive-set-parameters prior to invoking suit-directive-process-dependency. Since suit-directive-set-parameters has set-if-unset behaviour, this means that the dependent has effectively overriden the Dependency’s Parameter. Manifests that wish to enforce a specific value of a Parameter MUST use suit-directive-override-parameters instead. This satisfies USER_STORY.OVERRIDE and REQ.USE.MFST.COMPONENT of [RFC9124].

While suit-directive-set-parameters can be used outside of a Dependency use case, it has limited applicability: in linear manifests (without try-each, [I-D.ietf-suit-manifest], section 8.4.10.2) it either behaves as suit-directive-override-parameters or has no effect, depending on whether its targets are already set. When used as a set-if-unset construction following a try-each, suit-directive-override-parameters has the same effect as if a suit-directive-override-parameters were placed in the final element of the try-each with no preceding condition. This limits the applicability of suit-directive-set-parameters outside dependency use cases.

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

5.6.2. 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 "Payload Fetch," this will execute "Common metadata" in the current Dependency, then "Payload Fetch" in the current Dependency. Once this is complete, the Command following suit-directive-process-dependency will be processed.

If the current Component Index matches any of the following conditions, the Manifest Processor MUST Abort:

* The current Component index does not have an entry in the suit-dependencies map

* The current Component index has not been the target of a suit-condition-dependency-integrity

* The current section is "Common metadata"

If the current Component is True, then this Directive applies to all Dependencies.

When suit-directive-process-dependency completes, it forwards the last status code that occurred in the Dependency; an Abort in a Dependency causes an Abort in the suit-directive-process-dependency of the Dependent.

5.6.3. suit-condition-is-dependency

Check whether the current Component index is present in the Dependency list. If the current Component is in the Dependency list, suit-condition-is-dependency succeeds. Otherwise, it fails. This can be used along with component-id = True to act on all Dependencies or on all non-Dependency Components (Section 8).

5.6.4. suit-condition-dependency-integrity

Verify the integrity of a Dependency. When a Manifest Processor executes suit-condition-dependency-integrity, it performs the following operations:

1. Verify the signature of the Dependency’s suit-authentication-wrapper.

2. Compare the Dependency’s suit-authentication-wrapper digest to the dependent’s suit-parameter-image-digest

3. Verify the Dependency against the Depedency’s suit-authentication-wrapper digest

If any of these steps fails, the Manifest Processor MUST immediately Abort.

The Manifest Processor MAY cache the results of these operations for later use from the context of the current Manifest. The Manifest Processor MUST NOT use cached results from any other Manifest context. The Manifest Processor MUST prevent tampering with the cached results, e.g. through tamper-evident memory. If the Manifest Processor caches the results of these checks, it MUST eliminate this cache if:

* Any Fetch, or Copy operation targets the Dependency’s Component ID

* An Abort is encountered

* A Procedure completes

5.6.5. suit-directive-unlink

A Manifest Processor that supports multiple independent root manifests MUST support suit-directive-unlink. When a Component is no longer needed, the Manifest Processor unlinks the Component to inform the Manifest Processor that it is no longer needed.

If a Manifest is no longer needed, the Manifest Processor unlinks it. This causes the Manifest Processor to execute the suit-uninstall section of the unlinked Manifest, after which it decrements the reference count of the unlinked Manifest. The suit-uninstall section of a manifest typically contains an unlink of all its dependencies and components.

All components, including Manifests must be unlinked before deletion or overwrite. If the reference count of a component is non-zero, any command that alters that component MUST cause the Manifest Processor to Abort. Affected commands are:

* suit-directive-copy

* suit-directive-fetch

* suit-directive-write

The unlink Command decrements an implementation-defined reference counter. This reference counter MUST persist across restarts. The reference counter MUST NOT be decremented by a given Manifest more than once, and the Manifest Processor must enforce this. The Manifest Processor MAY choose to ignore an Unlink Directive depending on device policy.

When the reference counter of a Manifest reaches zero, the suit-uninstall Command sequence is invoked (Section 6).

suit-directive-unlink is OPTIONAL to implement in Manifest Processors, but Manifest Processors that support multiple independent Root Manifests MUST support suit-directive-unlink.

6. Uninstall

In some systems, particularly with multiple, independent, optional Components, it may be that there is a need to uninstall the Components that have been installed by a Manifest. Where this is expected, the uninstall Command sequence can provide the sequence needed to cleanly remove the Components defined by the Manifest and its Dependencies. In general, the suit-uninstall Command Sequence will contain primarily unlink Directives.

WARNING: This can cause faults where there are loose Dependencies (e.g., version range matching, [I-D.ietf-suit-update-management], Section 5.5), since a Component can be removed while it is depended upon by another Component. To avoid Dependency faults, a Manifest author MUST use explicit Dependencies where possible. To enable applications where explicit Dependency matching is not possible, a

Manifest Processor can track references to loose Dependencies via reference counting in the same way as explicit Dependencies, as described in Section 5.6.5.

The suit-uninstall Command Sequence is not severable, since it must always be available to enable uninstalling.

7. Staging and Installation

In order to coordinate between download and installation in different trust domains, the Update Procedure defined in [I-D.ietf-suit-manifest], Section 8.4.6 is divided into two sub-procedures:

* The Staging Procedure: This procedure is responsible for dependency resolution and acquiring all payloads required for the Update to proceed. It is composed of two command sequences
  - suit-dependency-resolution
  - suit-payload-fetch
* The Installation Procedure: This procedure is responsible for verifying staged components and installing them. It is composed of:
  - suit-candidate-verification
  - suit-install

This extension is backwards compatible when used with a Manifest Processor that supports the Update Procedure but does not support the Staging Procedure and Installation Procedure: the payload-fetch command sequence already contains suit-condition-image tests for each payload ([I-D.ietf-suit-manifest], section 7.3) which means that images are already validated when suit-install is invoked. This makes suit-candidate-verification OPTIONAL to implement.

The Staging and Installation Procedures are only required when Staging occurs in a different trust domain to Installation.

7.1. suit-candidate-verification

This command sequence is responsible for verifying that all elements of an update are present and correct prior to installation. This is only required when Installation occurs in a trust domain different from Staging, such as an installer invoked by the bootloader.

8. Creating Manifests

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.

8.1. Dependency Template

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

The following Commands are added to the shared sequence:

* Set Component Index Directive ([I-D.ietf-suit-manifest], Section 8.4.10.1)

* Set Parameters Directive (Section 5.6.1) for digest ([I-D.ietf-suit-manifest], Section 8.4.8.6). Note that the digest MUST match the SUIT_Digest in the Dependency's suit-authentication-block ([I-D.ietf-suit-manifest], Section 8.3).

The following Commands are placed into the Dependency resolution sequence:

* Set Component Index Directive ([I-D.ietf-suit-manifest], Section 8.4.10.1)

* Set Parameters Directive (Section 5.6.1) for a URI ([I-D.ietf-suit-manifest], Section 8.4.8.10)

* Fetch Directive ([I-D.ietf-suit-manifest], Section 8.4.10.4)

* Dependency Integrity Condition (Section 5.6.4)

* Process Dependency Directive (Section 5.6.2)

Then, the validate sequence contains the following operations:

* Set Component Index Directive ([I-D.ietf-suit-manifest], Section 8.4.10.1)

* Dependency Integrity Condition (Section 5.6.4)

* Process Dependency Directive (Section 5.6.2)

If any Dependency is declared, the dependent MUST populate all Command sequences for the current Procedure (Update or Invoke).

NOTE: Any changes made to Parameters in a Dependency persist in the dependent.

8.1.1. Integrated Dependencies

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 be a text string. The URI for the Dependency MUST match the text string key of the dependent's Envelope key. It is RECOMMENDED to make the text string key a resolvable URI so that a Dependency that is removed from the Envelope can still be fetched.

8.2. 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.

NOTE: This template also requires the extensions defined in [I-D.ietf-suit-firmware-encryption].

The following Commands are added to the shared sequence:

* Set Component Index Directive ([I-D.ietf-suit-manifest], Section 8.4.10.1)

* Set Parameters Directive (Section 5.6.1) for digest ([I-D.ietf-suit-manifest], Section 8.4.8.6). Note that the digest MUST match the SUIT_Digest in the Dependency's suit-authentication-block ([I-D.ietf-suit-manifest], Section 8.3).

The following operations are placed into the Dependency resolution block:

* Set Component Index Directive ([I-D.ietf-suit-manifest], Section 8.4.10.1)

* Set Parameters Directive (Section 5.6.1) for

  - URI ([I-D.ietf-suit-manifest], Section 8.4.8.9)

  - Encryption Info ([I-D.ietf-suit-firmware-encryption])

* Fetch Directive ([I-D.ietf-suit-manifest], Section 8.4.10.4)

* Dependency Integrity Condition (Section 5.6.4)

* Process Dependency Directive (Section 5.6.2)

Then, the validate block contains the following operations:

* Set Component Index Directive ([I-D.ietf-suit-manifest], Section 8.4.10.1)

* Check Image Match Condition ([I-D.ietf-suit-manifest], Section 8.4.9.2)

* Process Dependency Directive (Section 5.6.2)

A plaintext Manifest and its encrypted Dependency may also form a composite Manifest (Section 8.1.1).

8.3. Overriding Encryption Info Template

The goal of overriding the Encryption Info template is to separate the role of generating encrypted Payload and Encryption Info with Key-Encryption Key addressing Section 3 of [I-D.ietf-suit-firmware-encryption].

As an example, this template describes two manifests: - The dependent Manifest created by the Distribution System contains Encryption Info, allowing the Device to generate the Content-Encryption Key. - The Dependency created by the Author contains Commands to decrypt the encrypted Payload using Encryption Info above and to validate the plaintext Payload with SUIT_Digest.

NOTE: This template also requires the extensions defined in [I-D.ietf-suit-firmware-encryption].

The following operations are placed into the Dependency resolution block of dependent Manifest:

* Set Component Index Directive ([I-D.ietf-suit-manifest], Section 8.4.10.1) pointing at Dependency

* Set Parameters Directive (Section 5.6.1) for

  - Image Digest ([I-D.ietf-suit-manifest], Section 8.4.8.6)

  - URI ([I-D.ietf-suit-manifest], Section 8.4.8.9) of Dependency

* Fetch Directive ([I-D.ietf-suit-manifest], Section 8.4.10.4)

* Dependency Integrity Condition (Section 5.6.4)

The following Commands are placed into the Fetch/Install block of dependent Manifest

* Set Component Index Directive ([I-D.ietf-suit-manifest], Section 8.4.10.1) pointing at encrypted Payload

* Set Parameters Directive (Section 5.6.1) for

  - URI ([I-D.ietf-suit-manifest], Section 8.4.8.9)

* Set Component Index Directive ([I-D.ietf-suit-manifest], Section 8.4.10.1) pointing at Dependency

* Set Parameters Directive (Section 5.6.1) for

  - Encryption Info ([I-D.ietf-suit-firmware-encryption])

* Process Dependency Directive (Section 5.6.2)

The following Commands are placed into the same block of Dependency:

* Set Component Index Directive ([I-D.ietf-suit-manifest], Section 8.4.10.1) pointing at encrypted Payload

* Fetch Directive ([I-D.ietf-suit-manifest], Section 8.4.10.4)

* Set Component Index Directive ([I-D.ietf-suit-manifest], Section 8.4.10.1) pointing at to be decrypted Payload

* Override Parameters Directive ([I-D.ietf-suit-manifest], Section 8.4.10.3) for

  - Source Component ([I-D.ietf-suit-manifest], Section 8.4.8.11) pointing at encrypted Payload

* Copy Directive ([I-D.ietf-suit-manifest], Section 8.4.10.5) consuming the Encryption Info above

The Distribution System can Set the URI Parameter in the Fetch/Install block of dependent Manifest if it wants to overwrite the URI of the encrypted Payload.

Because the Author and the Distribution System have different roles and may be separate entities, it is highly recommended to leverage permissions ([I-D.ietf-suit-manifest], Section 9). For example, the Device can protect itself from an attacker who breaches the Distribution System by allowing only the Author's Manifest to modify the Component of (to be) decrypted Payload.

8.4. Operating on Multiple Components

In order to produce compact encoding, it is efficient to perform operations on multiple Components simultaneously. Because Dependencies and Component Images are processed at different times, there is a mechanism to distinguish between these elements: suit-condition-is-dependency. This can be used with suit-directive-try-each to perform operations just on Dependencies or just on Component Images.

For example, to fetch all Dependencies, the following Commands are added to the Dependency resolution block:

* Set Component Index Directive ([I-D.ietf-suit-manifest], Section 8.4.10.1)

* Set Parameters Directive (Section 5.6.1) for a URI ([I-D.ietf-suit-manifest], Section 8.4.8.9)

* Set Component Index Directive, with argument "True" ([I-D.ietf-suit-manifest], Section 8.4.10.1)

* Try Each Directive

  - Sequence 0

    o Condition Is Dependency

    o Fetch

    o Dependency Integrity Condition (Section 5.6.4)

    o Process Dependency

  - Sequence 1 (Empty; no Commands, succeeds immediately)

Another example is to fetch and validate all Component Images. The Image fetch sequence contains the following Commands:

* Set Component Index Directive ([I-D.ietf-suit-manifest], Section 8.4.10.1)

* Set Parameters Directive (Section 5.6.1) for a URI ([I-D.ietf-suit-manifest], Section 8.4.8.9)

* Set Component Index Directive, with argument "True" ([I-D.ietf-suit-manifest], Section 8.4.10.1)

* Try Each Directive

  - Sequence 0

    o Condition Is Dependency

    o Process Dependency

  - Sequence 1

    o Fetch

    o Condition Image Match

When some Components are "installed" or "loaded" it is more productive to use lists of Component indices rather than Component Index = True. For example, to install several Components, the following Commands should be placed in the Image Install Sequence:

* Set Component Index Directive ([I-D.ietf-suit-manifest], Section 8.4.10.1)

* Set Parameters Directive (Section 5.6.1) for the Source Component ([I-D.ietf-suit-manifest], Section 8.4.8.11)

* Set Component Index Directive, with argument containing list of destination Component indices ([I-D.ietf-suit-manifest], Section 8.4.10.1)

* Copy

* Set Component Index Directive, with argument containing list Dependency Component indices ([I-D.ietf-suit-manifest], Section 8.4.10.1)

* Process Dependency

9. IANA Considerations

IANA is requested to allocate the following numbers in the listed registries created by draft-ietf-suit-manifest:

9.1. SUIT Envelope Elements

Label	Name	Reference
15	Dependency Resolution	Section 5.5
18	Candidate Verification	Section 7.1

Table 2: New SUIT Envelope Elements

9.2. SUIT Manifest Elements

Label	Name	Reference
5	Manifest Component ID	Section 5.2.1
15	Dependency Resolution	Section 5.5
18	Candidate Verification	Section 7.1
24	Uninstall	Section 6

Table 3: New SUIT Manifest Elements

9.3. SUIT Common Elements

Label	Name	Reference
1	Dependencies	Section 5.2.2

Table 4: New SUIT Common Elements

9.4. SUIT Commands

Label	Name	Reference
7	Dependency Integrity	Section 5.6.4
8	Is Dependency	Section 5.6.3
11	Process Dependency	Section 5.6.2

19	Set Parameters	Section 5.6.1
33	Unlink	Section 5.6.5

Table 5: New SUIT Commands

10. Security Considerations

This specification is about a Manifest format protecting and describing how to retrieve, install, and invoke Images and as such it is part of a larger solution for delivering software updates to devices. A detailed security treatment can be found in the SUIT architecture [RFC9019] and in the SUIT information model [RFC9124].

The features added in this specification introduce several new threats. The introduction of Dependencies enables multiple entities to operate on a device with different privileges. While this is necessary to fulfill REQ.USE.MFST.COMPONENT ([RFC9124], Section 4.5.4), it also introduces a new requirement: REQ.SEC.ACCESS_CONTROL ([RFC9124], Section 4.3.13), which is required to address THREAT.MFST.OVERRIDE ([RFC9124], Section 4.2.13) and THREAT.UPD.UNAPPROVED ([RFC9124], Section 4.2.11).

Simultaneous processing of multiple Manifests, as enabled by Dependency processing, introduces risks of TOCTOU threats (THREAT.MFST.TOCTOU: [RFC9124], Section 4.2.18). Holding multiple Manifest Envelopes in memory simultaneously can exceed the capacity of the Manifest Processor's tamper-protected memory (REQ.SEC.MFST.CONST: [RFC9124], Section 4.3.21). To address this threat, the Manifest Processor MAY use modular processing as described in REQ.USE.PAYLOAD ([RFC9124], Section 4.5.12). If retaining the Manifests only, excluding envelopes, in immutable memory does not provide enough capacity, the Manifest Processor MAY reduce overhead by retaining the following elements for each manifest in immutable memory:

* Manifest Digest
* Parameters
* Current Component Index
* Current Command Sequence
* Current Command Sequence Offset

This allows a Manifest Processor to resume processing a manifest as follows:

* Copy the Manifest into immutable memory
* Validate the Manifest using the stored Manifest Digest
* Parse forward to find the Current Command Sequence
* Jump within the Command Seqeunce to the stored Command Sequence Offset

When identifying a Root Manifest's correct storage location, the Component Identifier MUST be evaluated vs. the access priviliges of an Author. Otherwise, the Component Identifier may permit an escalation of privilege: an authorised Author causes a manifest to be installed in a location for which the Author does not have access rights.

Since Dependencies are stored as Components, Dependency Integrity Checks and Image Verification are slightly different operations. While a typical Image is immutable, a Manifest Envelope can be modified in some ways (e.g. removing a Severable Element) without changing the Integrity Check result. Because of these factors, suit-directive-process-dependency requires that a dependency first be validated with suit-check suit-condition-dependency-integrity.

11. References

11.1. Normative References

[I-D.ietf-suit-firmware-encryption]
Tschofenig, H., Housley, R., Moran, B., Brown, D., and K. Takayama, "Encrypted Payloads in SUIT Manifests", Work in Progress, Internet-Draft, draft-ietf-suit-firmware-encryption-25, 7 July 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-suit-firmware-encryption-25>.

[I-D.ietf-suit-manifest]
Moran, B., Tschofenig, H., Birkholz, H., Zandberg, K., and O. R淡nningstad, "A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest", Work in Progress, Internet-Draft, draft-ietf-suit-manifest-34, 28 May 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-suit-manifest-34>.

[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/rfc/rfc2119>.

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

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/rfc/rfc8610>.

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

11.2. Informative References

[I-D.ietf-iotops-7228bis] Bormann, C., Ersue, M., Ker辰nen, A., and C. Gomez, "Terminology for Constrained-Node Networks", Work in Progress, Internet-Draft, draft-ietf-iotops-7228bis-02, 7 July 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-iotops-7228bis-02>.

[I-D.ietf-suit-update-management] Moran, B. and K. Takayama, "Update Management Extensions for Software Updates for Internet of Things (SUIT) Manifests", Work in Progress, Internet-Draft, draft-ietf-suit-update-management-09, 17 March 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-suit-update-management-09>.

[RFC6024] Reddy, R. and C. Wallace, "Trust Anchor Management Requirements", RFC 6024, DOI 10.17487/RFC6024, October 2010, <https://www.rfc-editor.org/rfc/rfc6024>.

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

[RFC9124] Moran, B., Tschofenig, H., and H. Birkholz, "A Manifest Information Model for Firmware Updates in Internet of Things (IoT) Devices", RFC 9124, DOI 10.17487/RFC9124, January 2022, <https://www.rfc-editor.org/rfc/rfc9124>.

[RFC9397] Pei, M., Tschofenig, H., Thaler, D., and D. Wheeler, "Trusted Execution Environment Provisioning (TEEP) Architecture", RFC 9397, DOI 10.17487/RFC9397, July 2023, <https://www.rfc-editor.org/rfc/rfc9397>.

Appendix A. A. Full CDDL

To be valid, the following CDDL (see [RFC8610]) MUST be appended to the SUIT Manifest CDDL. The SUIT CDDL is defined in Appendix A of [I-D.ietf-suit-manifest]

```
$$SUIT_Envelope_Extensions //= (
    suit-integrated-dependency-key => bstr .cbor SUIT_Envelope)

$$SUIT_Manifest_Extensions //=
    (suit-manifest-component-id => SUIT_Component_Identifier)

$$SUIT_severable-members-extensions //=
    (suit-dependency-resolution => bstr .cbor SUIT_Command_Sequence)

$$SUIT_severable-members-extensions //=
    (suit-candidate-verification => bstr .cbor SUIT_Command_Sequence)

$$unseverable-manifest-member-extensions //=
    (suit-uninstall => bstr .cbor SUIT_Command_Sequence)

suit-integrated-dependency-key = tstr

$$severable-manifest-members-choice-extensions //= (
    suit-dependency-resolution =>
        bstr .cbor SUIT_Command_Sequence / SUIT_Digest)

$$SUIT_Common-extensions //= (
    suit-dependencies => SUIT_Dependencies
)
SUIT_Dependencies = {
    + uint => SUIT_Dependency_Metadata
}
SUIT_Dependency_Metadata = {
    ? suit-dependency-prefix => SUIT_Component_Identifier
    * $$SUIT_Dependency_Extensions
}
```

```
   SUIT_Condition //= (
       suit-condition-dependency-integrity, SUIT_Rep_Policy)
   SUIT_Condition //= (
       suit-condition-is-dependency, SUIT_Rep_Policy)

   SUIT_Directive //= (
       suit-directive-process-dependency, SUIT_Rep_Policy)
   SUIT_Directive //= (suit-directive-set-parameters,
       {+ $$SUIT_Parameters})
   SUIT_Directive //= (
       suit-directive-unlink, SUIT_Rep_Policy)

   suit-manifest-component-id = 5

   suit-delegation = 1
   suit-dependency-resolution = 15
   suit-candidate-verification = 18
   suit-uninstall = 24

   suit-dependencies = 1

   suit-dependency-prefix = 1

   suit-condition-dependency-integrity     = 7
   suit-condition-is-dependency            = 8
   suit-directive-process-dependency       = 11
   suit-directive-set-parameters           = 19
   suit-directive-unlink                   = 33
```

Appendix B. B. Examples

The following examples demonstrate a small subset of the functionalities in this document.

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.

B.1. Example 0: Process Dependency

This example uses functionalities:

* manifest component id
* dependency resolution
* process dependency

The Dependency:

```
/ SUIT_Envelope_Tagged / 107({
  / authentication-wrapper / 2: << [
    << [
      / digest-algorithm-id: / -16 / SHA256 /,
      / digest-bytes: / h'AEBA316A9A1E38253B29E6C99B605383
                          68B8AC8B5E6B9ACE1D239970830BBE62'
    ] >>,
    << / COSE_Sign1_Tagged / 18([
      / protected: / << {
        / algorithm-id / 1: -9 / ESP256 /
      } >>,
      / unprotected: / {},
      / payload: / null,
      / signature: / h'3F3E9A2CA98208FEAEAEEADF7E1A0323
                      C97896ABFB79F91E8D0C1509B0A533CD
                      0B96BFC876A8F3B8ACE712FFF8EF7EA9
                      45E62A61E0BA5BD9929E4A1B47EC6475'
    ]) >>
  ] >>,
  / manifest / 3: << {
    / manifest-version / 1: 1,
    / manifest-sequence-number / 2: 0,
    / common / 3: << {
      / dependencies / 1: {
        / component-index / 1: {
          / dependency-prefix / 1: [
            'dependent.suit'
          ]
        }
      },
      / components / 2: [
        ['10']
      ]
    } >>,
```

```
      / manifest-component-id / 5: [
        'depending.suit'
      ],
      / invoke / 9: << [
        / directive-set-component-index / 12, 0,
        / directive-override-parameters / 20, {
          / parameter-invoke-args / 23: 'cat 00 10'
        },
        / directive-invoke / 23, 15
      ] >>,
      / dependency-resolution / 15: << [
        / directive-set-component-index / 12, 1,
        / directive-override-parameters / 20, {
          / parameter-image-digest / 3: << [
            / digest-algorithm-id: / -16 / SHA256 /,
            / digest-bytes: / h'0F02CAF6D3E61920D36BF3CEA7F862A1
                               3BB8FB1F09C3F4C29B121FEAB78EF3D8'
          ] >>,
          / parameter-image-size / 14: 190,
          / parameter-uri / 21: "http://example.com/dependent.suit"
        },
        / directive-fetch / 21, 2,
        / condition-image-match / 3, 15
      ] >>,
      / install / 20: << [
        / directive-set-component-index / 12, 1,
        / directive-override-parameters / 20, {
          / parameter-image-digest / 3: << [
            / digest-algorithm-id: / -16 / SHA256 /,
            / digest-bytes: / h'0F02CAF6D3E61920D36BF3CEA7F862A1
                               3BB8FB1F09C3F4C29B121FEAB78EF3D8'
          ] >>
        },
        / condition-dependency-integrity / 7, 15,
        / directive-process-dependency / 11, 0,

        / directive-set-component-index / 12, 0,
        / directive-override-parameters / 20, {
          / parameter-content / 18: ' in multiple trust domains'
        },
        / directive-write / 18, 15
      ] >>
    } >>
  })

  Total size of Envelope with COSE authentication object: 373
```

```
D86BA2025873825824822F5820AEBA316A9A1E38253B29E6C99B60538368
B8AC8B5E6B9ACE1D239970830BBE62584AD28443A10128A0F658403F3E9A
2CA98208FEAEAEEADF7E1A0323C97896ABFB79F91E8D0C1509B0A533CD0B
96BFC876A8F3B8ACE712FFF8EF7EA945E62A61E0BA5BD9929E4A1B47EC64
750358F9A70101020003581CA201A101A101814E646570656E64656E742E
737569740281814231300581 4E646570656E64696E672E73756974095286
0C0014A11749636174203030203130170F0F5857880C0114A3035824822F
58200F02CAF6D3E61920D36BF3CEA7F862A13BB8FB1F09C3F4C29B121FEA
B78EF3D80E18BE157821687474703A2F2F6578616D706C652E636F6D2F64
6570656E64656E742E737569741502030F1458538E0C0114A1035824822F
58200F02CAF6D3E61920D36BF3CEA7F862A13BB8FB1F09C3F4C29B121FEA
B78EF3D8070F0B000C0014A112581A20696E206D756C7469706C65207472
75737420646F6D61696E73120F
```

The dependent Manifest (fetched from "https://example.com/dependent.suit"):

```
/ SUIT_Envelope_Tagged / 107({
  / authentication-wrapper / 2: << [
    << [
      / digest-algorithm-id: / -16 / SHA256 /,
      / digest-bytes: / h'0F02CAF6D3E61920D36BF3CEA7F862A1
                          3BB8FB1F09C3F4C29B121FEAB78EF3D8'
    ] >>,
    << / COSE_Sign1_Tagged / 18([
      / protected: / << {
        / algorithm-id / 1: -9 / ESP256 /
      } >>,
      / unprotected: / {},
      / payload: / null,
      / signature: / h'A25F337126369D2E0B451C01DBD8CDB8
                       4A77E7F6C39E789DB3D227753494000C
                       9D250001FDDCA39B4B4E3755A7278C11
                       998171905F56C394CFBB907105DA804F'
    ]) >>
  ] >>,
  / manifest / 3: << {
    / manifest-version / 1: 1,
    / manifest-sequence-number / 2: 0,
    / common / 3: << {
      / components / 2: [
        ['00']
      ]
    } >>,
    / manifest-component-id / 5: [
      'dependent.suit'
    ],
    / invoke / 9: << [
      / directive-override-parameters / 20, {
        / parameter-invoke-args / 23: 'cat 00'
      },
      / directive-invoke / 23, 15
    ] >>,
    / install / 20: << [
      / directive-override-parameters / 20, {
        / parameter-content / 18: 'hello world'
      },
      / directive-write / 18, 15
    ] >>
  } >>
})

Total size of Envelope with COSE authentication object: 190
```

```
D86BA2025873825824822F58200F02CAF6D3E61920D36BF3CEA7F862A13B
B8FB1F09C3F4C29B121FEAB78EF3D8584AD28443A10128A0F65840A25F33
7126369D2E0B451C01DBD8CDB84A77E7F6C39E789DB3D227753494000C9D
250001FDDCA39B4B4E3755A7278C11998171905F56C394CFBB907105DA80
4F035842A6010102000347A10281814230300581 4E646570656E64656E74
2E73756974094D8414A1174663617420303017 0F14528414A1124B68656C
6C6F20776F726C64120F
```

B.2. Example 1: Integrated Dependency

* manifest component id

* dependency resolution

* process dependency

* integrated dependency

```
/ SUIT_Envelope_Tagged / 107({
  / authentication-wrapper / 2: << [
    << [
      / digest-algorithm-id: / -16 / SHA256 /,
      / digest-bytes: / h'88E1199580864EB1D1AD35EB5925BE68
                          CA565EE3BB39C27CDB31CEDA4DD667DF'
    ] >>,
    << / COSE_Sign1_Tagged / 18([
      / protected: / << {
        / algorithm-id / 1: -9 / ESP256 /
      } >>,
      / unprotected: / {},
      / payload: / null,
      / signature: / h'074A361F7BBFA2ACF4EC3CFDAF4FDD87
                       38414BAD672CAEA4F43607BE6031EA90
                       CB0C283A03C728608B0509C6FD2AFED4
                       0CFB0C3D341340830A00905E6A729890'
    ]) >>
  ] >>,
  / manifest / 3: << {
    / manifest-version / 1: 1,
    / manifest-sequence-number / 2: 0,
    / common / 3: << {
      / dependencies / 1: {
        / component-index / 1: {
          / dependency-prefix / 1: [
            'dependent.suit'
          ]
        }
      },
```

```
        / components / 2: [
          ['10']
        ]
      } >>,
      / manifest-component-id / 5: [
        'depending.suit'
      ],
      / invoke / 9: << [
        / directive-set-component-index / 12, 0,
        / directive-override-parameters / 20, {
          / parameter-invoke-args / 23: 'cat 00 10'
        },
        / directive-invoke / 23, 15
      ] >>,
      / dependency-resolution / 15: << [
        / directive-set-component-index / 12, 1,
        / directive-override-parameters / 20, {
          / parameter-image-digest / 3: << [
            / digest-algorithm-id: / -16 / SHA256 /,
            / digest-bytes: / h'0F02CAF6D3E61920D36BF3CEA7F862A1
                               3BB8FB1F09C3F4C29B121FEAB78EF3D8'
          ] >>,
          / parameter-image-size / 14: 190,
          / parameter-uri / 21: "#dependent.suit"
        },
        / directive-fetch / 21, 2,
        / condition-image-match / 3, 15
      ] >>,
      / install / 20: << [
        / directive-set-component-index / 12, 1,
        / directive-process-dependency / 11, 0,

        / directive-set-component-index / 12, 0,
        / directive-override-parameters / 20, {
          / parameter-content / 18: ' in multiple trust domains'
        },
        / directive-write / 18, 15
      ] >>
    } >>,
    "#dependent.suit":
      h'D86BA2025873825824822F58200F02CAF6D3E61920D36BF3CEA7F862A13B
        B8FB1F09C3F4C29B121FEAB78EF3D8584AD28443A10128A0F65840A25F33
        7126369D2E0B451C01DBD8CDB84A77E7F6C39E789DB3D227753494000C9D
        250001FDDCA39B4B4E3755A7278C11998171905F56C394CFBB907105DA80
        4F035842A6010102000347A102818142303005814E646570656E64656E74
        2E73756974094D8414A1174663617420303017 0F14528414A1124B68656C
        6C6F20776F726C64120F'
  })
```

Total size of Envelope with COSE authentication object: 519

Envelope with COSE authentication object:

```
D86BA3025873825824822F582088E1199580864EB1D1AD35EB5925BE68CA
565EE3BB39C27CDB31CEDA4DD667DF584AD28443A10128A0F65840074A36
1F7BBFA2ACF4EC3CFDAF4FDD8738414BAD672CAEA4F43607BE6031EA90CB
0C283A03C728608B0509C6FD2AFED40CFB0C3D341340830A00905E6A7298
900358BBA70101020003581CA201A101A101814E646570656E64656E742E
737569740281814231300581 4E646570656E64696E672E73756974095286
0C0014A11749636174203030203130170F0F5844880C0114A3035824822F
58200F02CAF6D3E61920D36BF3CEA7F862A13BB8FB1F09C3F4C29B121FEA
B78EF3D80E18BE156F23646570656E64656E742E737569741502030F1458
288A0C010B000C0014A112581A20696E206D756C7469706C652074727573
7420646F6D61696E73120F6F23646570656E64656E742E7375697458BED8
6BA2025873825824822F58200F02CAF6D3E61920D36BF3CEA7F862A13BB8
FB1F09C3F4C29B121FEAB78EF3D8584AD28443A10128A0F65840A25F3371
26369D2E0B451C01DBD8CDB84A77E7F6C39E789DB3D227753494000C9D25
0001FDDCA39B4B4E3755A7278C11998171905F56C394CFBB907105DA804F
035842A6010102000347A10281814230300581 4E646570656E64656E742E
73756974094D8414A11746636174203030170F14528414A1124B68656C6C
6F20776F726C64120F
```

Authors' Addresses

Brendan Moran
Arm Limited
Email: brendan.moran.ietf@gmail.com

Ken Takayama
SECOM CO., LTD.
Email: ken.takayama.ietf@gmail.com