SUIT
Internet-Draft
Intended status: Standards Track
Expires: 8 September 2022

B. Moran
Arm Limited

7 March 2022

SUIT Manifest Extensions for Multiple Trust Domains
draft-ietf-suit-trust-domains-00

Abstract

This specification describes extensions to the SUIT manifest format (as defined in [I-D.ietf-suit-manifest]) for use in deployments with multiple trust domains. A device has more than one trust domain when it uses different trust anchors for different purposes or components in the context of firmware update.

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 8 September 2022.

Table of Contents

1. Introduction

Devices that go beyond single-signer update require more complex rules for deploying firmware updates. For example, devices may require:

* long-term trust anchors with a mechanism to delegate trust to short term keys.

* software components from multiple software signing authorities.

* a mechanism to remove an uneeded component

* single-object dependencies

* a partly encrypted manifest so that distribution does not reveal private information

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

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

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. 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 for the Update Procedure:

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

One additional assumption is added to the Invocation Procedure:

* All dependencies must be validated prior to loading.

Two steps are added to the expected installation workflow of a Recipient:

1. *Verify delegation chains*

2. Verify the signature of the manifest.

3. Verify the applicability of the manifest.

4. *Resolve dependencies.*

5. Fetch payload(s).

6. Install payload(s).

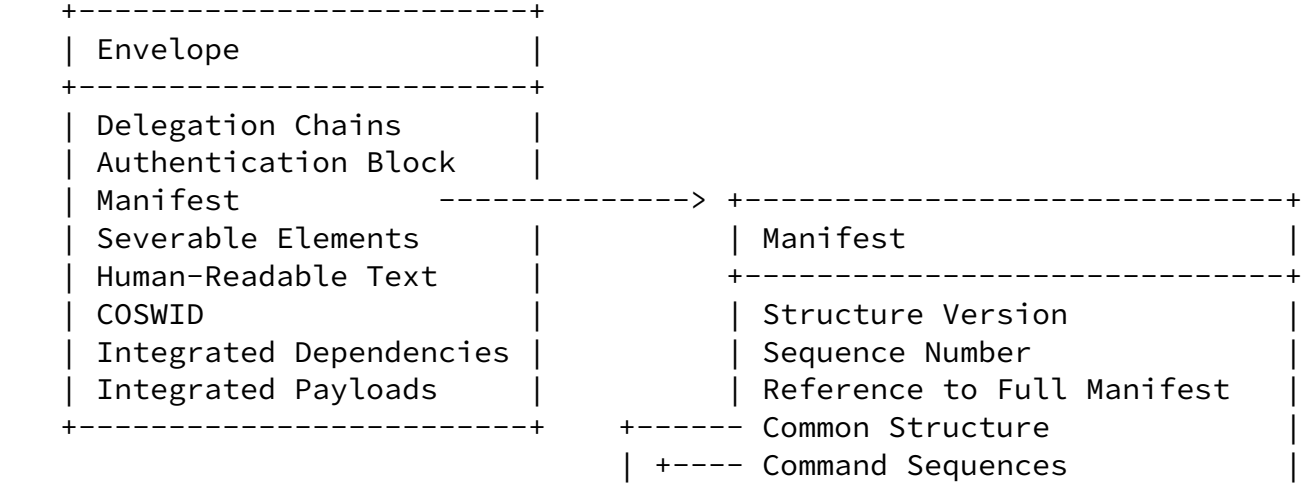
In addition, when multiple manifests are used for an update, each manifest's steps occur in a lockstep fashion; all manifests have dependency resolution performed before any manifest performs a payload fetch, etc.

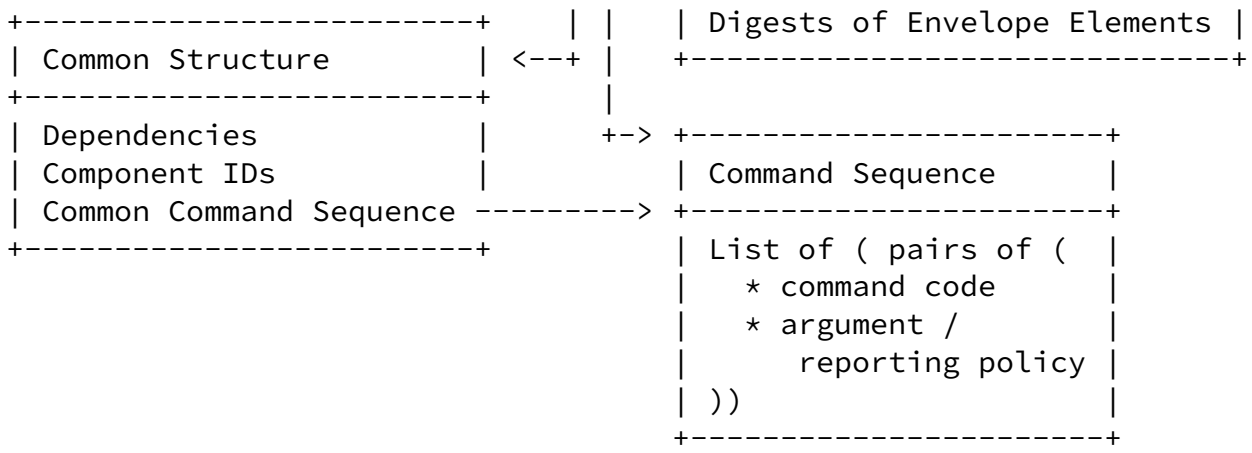
4. Changes to Manifest Metadata Structure

To accomodate the additional metadata needed to enable these features, the envelope and manifest have several new elements added.

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

The new metadata structure is shown below.





5. 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 (See Sectino 5.2 of [I-D.ietf-suit-manifest]).

See Section 5.1 for more detail.

5.1. 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 8.3 of [I-D.ietf-suit-manifest]). 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.

6. Dependencies

A dependency is another SUIT_Envelope that describes additional components.

Dependency manifests 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:

* An IoT device may contain a processor in its radio in addition to the primary processor. These two processors may have separate firmware with separate signing authorities. Dependencies allow the firmware 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 operator overrides the URI of payload by providing a dependent manifest that references the original manifest, but replaces its URI.

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

By using dependencies, components such as software, configuration,

models, and other resoruces authenticated by different trust anchors can be delivered to devices.

6.1. Changes to Required Checks

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

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 defining which authorities have access to different componetns), it MAY choose to skip signature verification of dependencies, since they are referenced 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, 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.

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. Changes to Abstract Machine Description

This section augments the Abstract Machine Description (Section 6.4) in [I-D.ietf-suit-manifest] 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:

* All commands may target dependency manifests as well as components. To support this behaviour, there is a new command instroduced: Set Dependency Index. This change works together with Set Component Index to choose the object on which the manifest is operating.

* Dependencies are processed in lock-step 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.

The logic of Set Componment Index is modified as below:

As in [I-D.ietf-suit-manifest], 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], but the actual operation performed is:

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

Dependencies are identified by digest, but referenced in commands by Dependency Index, the index into the array of Dependencies.

6.3. Changes to Special Cases of Component Index and Dependency Index

The considerations that apply in Special Cases of Component Index and Dependency Index (Section 6.5) of [I-D.ietf-suit-manifest] are augmented to include Dependency Index as well as Component Index.

The target(s) assigned for each command are defined 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)
```

6.4. Processing Dependencies

As described in Section 6.1, 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 6.3

The interpreter also performs the checks described in Section 6.1 to ensure that the dependent is processing the dependency correctly.

6.4.1. 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 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, 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.

6.5. 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 Component Index	current := components[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])
Unlink	unlink(current)

Table 1

6.5.1. 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 6.3 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.

6.5.2. 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 6.3 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.

6.5.3. 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.

6.5.3.1. 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.

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.

6.5.4. 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.

6.6. SUIT_Dependency Manifest Element

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 8.4.5.1 of [I-D.ietf-suit-manifest]). 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 6.4.1.

7. 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.

7.1. 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 6.5.2)

* Set Parameters directive (see Section 6.5.3.1) for URI (see Section 8.4.8.9 of [I-D.ietf-suit-manifest])

* Fetch directive (see Section 8.4.10.4 of [I-D.ietf-suit-manifest])

* Check Image Match condition (see Section 8.4.9.2 of [I-D.ietf-suit-manifest] of [I-D.ietf-suit-manifest])

* Process Dependency directive (see Section 6.5.3)

Then, the validate sequence contains the following operations:

* Set Dependency Index directive (see Section 6.5.2)

* Check Image Match condition (see Section 8.4.9.2 of [I-D.ietf-suit-manifest])

* Process Dependency directive (see Section 6.5.3)

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

7.1.1. 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".

7.2. 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 operations are placed into the dependency resolution block:

* Set Dependency Index directive (see Section 6.5.2)
* Set Parameters directive (see Section 6.5.3.1) for
  - URI (see Section 8.4.8.9 of [I-D.ietf-suit-manifest])
  - Encryption Info (See [I-D.ietf-suit-firmware-encryption])
* Fetch directive (see Section 8.4.10.4 of [I-D.ietf-suit-manifest])
* Check Image Match condition (see Section 8.4.9.2 of [I-D.ietf-suit-manifest])
* Process Dependency directive (see Section 6.5.3)

Then, the validate block contains the following operations:

* Set Dependency Index directive (see Section 6.5.2)
* Check Image Match condition (see Section 8.4.9.2 of

[I-D.ietf-suit-manifest])

* Process Dependency directive (see Section 6.5.3)

A plaintext manifest and its encrypted dependency may also form a composite manifest (Section 7.1.1).

8. IANA Considerations

IANA is requested to allocate the following numbers in the listed registries:

8.1. SUIT Commands

Label	Name	Reference	
13	Set Dependency Index	Section 6.5.2	
18	Process Dependency	suit-directive-process-dependency	Section 6.5.3
19	Set Parameters	Section 6.5.3.1	
33	Unlink	Section 6.5.4	

Table 2

9. Security Considerations

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

10. References

10.1. Normative References

[I-D.ietf-suit-manifest]
Moran, B., Tschofenig, H., Birkholz, H., and K. Zandberg, "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-16, 25 October 2021, <https://www.ietf.org/archive/id/draft-ietf-suit-manifest-16.txt>.

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

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

[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>.

[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>.

[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>.

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

10.2. Informative References

[I-D.ietf-suit-firmware-encryption]
Tschofenig, H., Housley, R., and B. Moran, "Firmware Encryption with SUIT Manifests", Work in Progress, Internet-Draft, draft-ietf-suit-firmware-encryption-03, 7 March 2022, <https://www.ietf.org/archive/id/draft-ietf-suit-firmware-encryption-03.txt>.

[I-D.ietf-suit-information-model]
Moran, B., Tschofenig, H., and H. Birkholz, "A Manifest Information Model for Firmware Updates in Internet of Things (IoT) Devices", Work in Progress, Internet-Draft, draft-ietf-suit-information-model-13, 8 July 2021, <https://www.ietf.org/archive/id/draft-ietf-suit-information-model-13.txt>.

[I-D.ietf-teep-architecture]
Pei, M., Tschofenig, H., Thaler, D., and D. Wheeler, "Trusted Execution Environment Provisioning (TEEP) Architecture", Work in Progress, Internet-Draft, draft-ietf-teep-architecture-16, 28 February 2022, <https://www.ietf.org/archive/id/draft-ietf-teep-architecture-16.txt>.

Appendix A. A. Full CDDL

To be valid, the following CDDL MUST be appended to the SUIT Manifest

CDDL. The SUIT CDDL is defined in Appendix A of [I-D.ietf-suit-manifest]

```
$$SUIT_Envelope_Extensions //=
    (suit-delegation => bstr .cbor SUIT_Delegation)
$$SUIT_Envelope_Extensions //= SUIT_Integrated_Dependency

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

CWT = SUIT_Authentication_Block

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

SUIT_Integrated_Dependency = (
    suit-integrated-dependency-key => bstr .cbor SUIT_Envelope)
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         = [ + SUIT_Dependency ]

SUIT_Dependency = {
```



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

SUIT_Directive //= (
    suit-directive-set-dependency-index, IndexArg)
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-delegation = 1
   suit-dependency-resolution = 7

   suit-dependencies = 1

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

   suit-directive-set-dependency-index     = 13
   suit-directive-process-dependency       = 18
   suit-directive-set-parameters           = 19
   suit-directive-unlink                   = 33
```

Author's Address

Brendan Moran
Arm Limited
Email: Brendan.Moran@arm.com