

SUIT
Internet-Draft
Intended status: Standards Track
Expires: 11 June 2026

H. Tschofenig
H-BRS
R. Housley
Vigil Security
B. Moran
Arm Limited
D. Brown
Linaro
K. Takayama
SECOM CO., LTD.
8 December 2025

Encrypted Payloads in SUIT Manifests
draft-ietf-suit-firmware-encryption-26

Abstract

This document specifies techniques for encrypting software, firmware, machine learning models, and personalization data by utilizing the IETF SUIT manifest. Key agreement is provided by ephemeral-static (ES) Diffie-Hellman (DH) and AES Key Wrap (AES-KW). ES-DH uses public key cryptography while AES-KW uses a pre-shared key. Encryption of the plaintext is accomplished with conventional symmetric key cryptography.

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 11 June 2026.

Copyright Notice

Copyright (c) 2025 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

- 1. Introduction 3
- 2. Conventions and Terminology 4
- 3. Architecture 5
- 4. Encryption Extensions 7
 - 4.1. Directive Write 8
 - 4.2. Directive Copy 9
 - 4.3. Authenticating the Payload 9
- 5. Content Key Distribution 10
 - 5.1. Content Key Distribution with AES Key Wrap 11
 - 5.1.1. Introduction 11
 - 5.1.2. Deployment Options 11
 - 5.1.3. The CDDL of SUIT_Encryption_Info for AES-KW binary 12
 - 5.2. Content Key Distribution with Ephemeral-Static Diffie-Hellman 13
 - 5.2.1. Introduction 13
 - 5.2.2. Deployment Options 14
 - 5.2.3. The CDDL of SUIT_Encryption_Info for ES-DH binary 15
 - 5.2.4. Context Information Structure 16
- 6. Content Encryption 17
 - 6.1. AES-GCM 18
 - 6.1.1. Introduction 18
 - 6.1.2. AES-KW + AES-GCM Example 19
 - 6.1.3. ECDH-ES+AES-KW + AES-GCM Example 20
 - 6.2. AES-CTR 21
 - 6.2.1. Introduction 22
 - 6.2.2. AES-KW + AES-CTR Example 23
 - 6.2.3. ECDH-ES+AES-KW + AES-CTR Example 24
- 7. Integrity Check on Encrypted and Decrypted Payloads 26
 - 7.1. Validating Payload Integrity 26
 - 7.1.1. Image Match after Decryption 27
 - 7.1.2. Image Match before Decryption 27
 - 7.1.3. Checking Authentication Tag while Decrypting 28
 - 7.2. Payload Integrity Validation 28
- 8. Firmware Updates on IoT Devices with Flash Memory 30
- 9. Complete Examples 33
 - 9.1. AES Key Wrap Example with Write Directive 34
 - 9.2. AES Key Wrap Example with Fetch + Copy Directives 36

9.3. ES-DH Example with Write + Copy Directives	42
9.4. ES-DH Example with Dependency	44
10. Operational Considerations	48
11. Security Considerations	49
12. IANA Considerations	51
13. References	51
13.1. Normative References	51
13.2. Informative References	52
Appendix A. Full CDDL	54
Acknowledgements	54
Authors' Addresses	54

1. Introduction

Vulnerabilities in Internet of Things (IoT) devices have highlighted the need for a reliable and secure firmware update mechanism, especially for constrained devices. To protect firmware images, the SUIT manifest format was developed [I-D.ietf-suit-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. [RFC9124] outlines the necessary information a SUIT manifest has to provide. In addition to protecting against modification via digital signatures or message authentication codes, the format can also offer confidentiality.

Encryption prevents third parties, including attackers, from accessing the payload. Attackers often require detailed knowledge of a binary, such as a firmware image, to launch successful attacks. For instance, return-oriented programming (ROP) [ROP] requires access to the binary, and encryption makes writing exploits significantly more difficult. Beyond ensuring the confidentiality of the binary itself, protecting the confidentiality of the source code will also be necessary to prevent reverse engineering and reproduction of the firmware.

The initial motivation for this document was firmware encryption. However, the use of SUIT manifests has expanded to encompass other scenarios that require integrity and confidentiality protection, including:

- * Software packages
- * Personalization and configuration data
- * Machine learning models

These additional use cases stem from the work on Trusted Execution Environment Provisioning (TEEP), as detailed in [RFC9397] and [I-D.ietf-teep-usecase-for-cc-in-network]. The distinction between software and firmware is clarified in [RFC9019].

For consistency and simplicity, we use the term "payload" generically to refer to all objects subject to encryption.

The payload is encrypted using a symmetric content encryption key, which can be established through various mechanisms. This document defines two content key distribution methods for use with the SUIT manifest:

- * Ephemeral-Static (ES) Diffie-Hellman (DH), and
- * AES Key Wrap (AES-KW).

The first method relies on asymmetric cryptography, while the second uses symmetric cryptography.

Our design aims to reduce the number of content key distribution methods for payload encryption, thereby increasing interoperability between different SUIT manifest parser implementations. The mandatory-to-implement algorithms are described in a separate document [I-D.ietf-suit-mti].

The goal of this specification is to protect payloads both during end-to-end transport (from the distribution system to the device) and at rest when stored on the device. Constrained devices often employ eXecute In Place (XIP), a method of executing code directly from flash memory rather than loading it into RAM. Many of these devices lack hardware-based, on-the-fly decryption for code stored in flash memory, which may require decrypting and storing firmware images in on-chip flash before execution. However, we expect hardware-based, on-the-fly decryption to become more common in the future, enhancing confidentiality at rest.

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.

This document assumes familiarity with the SUIT manifest [I-D.ietf-suit-manifest], the SUIT information model [RFC9124], and the SUIT architecture [RFC9019].

The following abbreviations are used in this document:

- * Key Wrap (KW), defined in [RFC3394] (for use with AES)
- * Key-Encryption Key (KEK) [RFC3394]
- * Content-Encryption Key (CEK) [RFC5652]
- * Ephemeral-Static (ES) Diffie-Hellman (DH) [RFC9052]
- * Authenticated Encryption with Associated Data (AEAD)
- * Execute in Place (XIP)

The terms sender and recipient have the following meaning:

- * Sender: Entity that sends an encrypted payload.
- * Recipient: Entity that receives an encrypted payload.

Additionally, we introduce the term "distribution system" (or distributor) to refer to an entity that knows the recipients of payloads. It is important to note that the distribution system is far more than a file server. For use of encryption, the distribution system either knows the public key of the recipient (for ES-DH), or the KEK (for AES-KW).

The author, which is responsible for creating the payload, does not know the recipients. The author may, for example, be a developer building a firmware image.

The author and the distribution system are logical roles. In some deployments these roles are separated in different physical entities and in others they are co-located.

3. Architecture

[RFC9019] outlines the architecture for distributing payloads and manifests from an author to devices. However, it does not cover payload encryption in detail. This document extends that architecture to support encryption, as illustrated in Figure 1.

To encrypt a payload, it is essential to know the recipient. For AES-KW, the Key Encryption Key (KEK) must be known, and for ES-DH, the sender needs access to the recipient's public key. This public key and its associated parameters may be found in the recipient's X.509 certificate [RFC5280]. For authentication and integrity protection, recipients must be provisioned with a trust anchor when

the manifest is protected by a digital signature. If a MAC is used for manifest protection, a symmetric key must be shared between the recipient and the sender.

With encryption, the author cannot simply create and sign a manifest for the payload, as the recipients are often unknown. Therefore, the author must collaborate with the distribution system. The degree of this collaboration is discussed below.

The primary purpose of encryption is to protect against adversaries along the path between the distribution system and the device. There is also a risk that adversaries may extract the decrypted firmware image from the device itself. Consequently, the device must be safeguarded against physical attacks. Such countermeasures are outside the scope of this specification.

Note: It is assumed that a mutually authenticated communication channel with integrity and confidentiality protection exists between the author and the distribution system. For example, the author could upload the manifest and firmware image to the distribution system via a mutually authenticated HTTPS REST API.

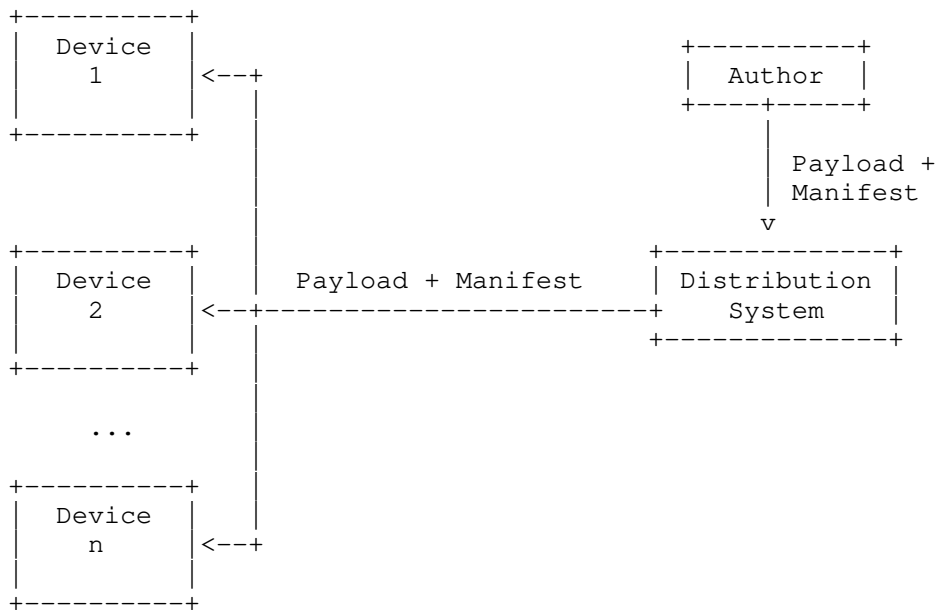


Figure 1: Architecture for the distribution of Encrypted Payloads.

When the author delegates encryption rights to the distributor, two models are possible:

1. **Replacing the COSE_Encrypt and Re-signing the Manifest:** The distributor replaces the COSE_Encrypt structure in the manifest and then signs the manifest again. However, since the COSE_Encrypt structure is within a signed container, this presents a challenge: replacing COSE_Encrypt alters the digest of the manifest, thereby invalidating the signature. As a result, the distributor must be able to sign the new manifest. If this is the case, the distributor gains the authority to construct and sign manifests, effectively allowing them to sign code and giving them full control over the recipient. Distributors typically perform re-encryption online to manage large numbers of devices efficiently, which prevents air-gapping the signing operations. This approach necessitates the secure storage of signing keys, as outlined in Section 4.3.17 and Section 4.3.18 of [RFC9124]. Despite these issues, this model represents the current standard practice for IoT firmware updates.
2. **Two-Layer Manifest System:** The distributor creates a new manifest that overrides the COSE_Encrypt using the dependency system defined in [I-D.ietf-suit-trust-domains]. This method introduces additional overhead, including one more signature verification, one extra manifest, and the need for extra mechanisms on the recipient side to handle dependency processing. While this adds complexity, it also enhances security.

These two models offer different threat profiles for the distributor. If the distributor is limited to encryption rights, an attacker who breaches the distributor can only launch a limited attack by encrypting a modified binary. However, recipients will detect the attack during the image digest check and immediately revert to the correct image.

It is RECOMMENDED that distributors adopt the two-layer manifest approach to distribute content encryption keys without re-signing the manifest, despite the added complexity and the increased number of signature verifications required on the recipient side.

4. Encryption Extensions

Extending the SUIT manifest to support payload encryption requires minimal changes and is achieved by adding the `suit-parameter-encryption-info` field to the `SUIT_Parameters` structure, as illustrated in Figure 2. When the `suit-parameter-encryption-info` is included, the manifest processor will attempt to decrypt data during copy or write operations.

The `SUIT_Encryption_Info` structure contains the content key distribution information. The details of the `SUIT_Encryption_Info` structure are provided in Section 5.1 (for AES-KW) and Section 5.2 (for ES-DH).

```
SUIT_Parameters // = (suit-parameter-encryption-info
    => bstr .cbor SUIT_Encryption_Info)

suit-parameter-encryption-info = TBD19
```

Figure 2: CDDL of the `SUIT_Parameters` Extension.

RFC Editor's Note (TBD19): The value for the `suit-parameter-encryption-info` parameter is set to 19, as the proposed value.

Once a CEK is available, the steps outlined in Section 6 apply to both content key distribution methods described in this section.

When used with the "Directive Write" and "Directive Copy" directives, the `SUIT_Encryption_Info` structure MUST be included in either the `suit-directive-override-parameters` or the `suit-directive-set-parameters`. An implementation conforming to this specification MUST support both of these parameters.

4.1. Directive Write

An author uses the Directive Write (`suit-directive-write`) to decrypt the content specified by `suit-parameter-content` using `suit-parameter-encryption-info`. This directive is used to write a specific data directly to a component.

Figure 3 illustrates an example of the Directive Write, which is described in the CDDL in Figure 2. The encrypted payload specified by `parameter-content`, represented as `h'EA1...CED'` in the example, is decrypted using the `SUIT_Encryption_Info` structure referenced by `parameter-encryption-info`, i.e., `h'D86...1F0'` in L3. The resulting plaintext payload is then stored in component #0, which is the default if no specific component is explicitly designated.

```
/ 1/ / directive-override-parameters / 20, {
/ 2/   / parameter-content / 18: h'EA1...CED',
/ 3/   / parameter-encryption-info / TBD19: h'D86...1F0'
/ 4/ },
/ 5/ / directive-write / 18, 15
```

Figure 3: Example showing the extended `suit-directive-write`.

RFC Editor's Note (TBD19): The value for the parameter-encryption-info parameter is set to 19, as the proposed value.

4.2. Directive Copy

An author uses the Directive Copy (`suit-directive-copy`) to decrypt the content of the component specified by `suit-parameter-source-component` using `suit-parameter-encryption-info`. This directive is used to copy data from one component to another.

Figure 4 illustrates the Directive Copy. In this example the encrypted payload is found at the URI indicated by the `parameter-uri`, i.e., `"coaps://example.com/encrypted.bin"` in L3. The encrypted payload will be downloaded and stored in component #1, as indicated by `directive-set-component-index` in L1.

Then, the information in the `SUIT_Encryption_Info` structure referred to by `parameter-encryption-info`, i.e., `h'D86...1F0'` in L9, will be used to decrypt the content in component #1 and the resulting plaintext payload will be stored into component #0 (as set in L7). The command in L12 invokes the operation.

```

/ 1/ / directive-set-component-index / 12, 1,
/ 2/ / directive-override-parameters / 20, {
/ 3/ / parameter-uri / 21: "coaps://example.com/encrypted.bin",
/ 4/ },
/ 5/ / directive-fetch / 21, 15,
/ 6/
/ 7/ / directive-set-component-index / 12, 0,
/ 8/ / directive-override-parameters / 20, {
/ 9/ / parameter-encryption-info / TBD19: h'D86...1F0',
/ 10/ / parameter-source-component / 22: 1
/ 11/ },
/ 12/ / directive-copy / 22, 15

```

Figure 4: Example showing the extended `suit-directive-copy`.

RFC Editor's Note (TBD19): The value for the `suit-parameter-encryption-info` parameter is set to 19, as the proposed value.

4.3. Authenticating the Payload

The payload to be encrypted MAY be detached and, in that case, it is not covered by the digital signature or the MAC protecting the manifest. (To be more precise, the `suit-authentication-wrapper` found in the envelope contains a digest of the manifest in the `SUIT Digest Container`.)

The lack of authentication and integrity protection of the payload is particularly a concern when a cipher without integrity protection is used.

To provide authentication and integrity protection of the payload in the detached case a SUIF Digest Container with the hash of the encrypted and/or plaintext payload MUST be included in the manifest. See `suit-parameter-image-digest` parameter in Section 8.4.8.6 of [I-D.ietf-suit-manifest].

Once a CEK is available, the steps described in Section 6 are applicable. These steps apply to both content key distribution methods.

More detailed examples for the two directives can be found in Section 9.1.

5. Content Key Distribution

The following sub-sections describe two content key distribution methods: AES Key Wrap (AES-KW) and Ephemeral-Static Diffie-Hellman (ES-DH). While many other methods are specified in the literature and supported by COSE, AES-KW and ES-DH were chosen for their widespread use in the market today. They were selected for their maturity, differing security properties, and strong interoperability.

Interoperability requirements for content key distribution methods differ: since a device typically supports only one of the two specified methods, the distribution system must be aware of the supported method. Restricting a constrained device to a single content key distribution method also helps minimize code size.

Both content key distribution methods require the CEKs to be randomly generated. The guidelines for random number generation in [RFC8937] MUST be followed.

When sending an encrypted payload to multiple recipients, various deployment options are available. The following notation is used to explain these options:

- KEK[R1, S] refers to a KEK shared between recipient R1 and the sender S.
- CEK[R1, S] refers to a CEK shared between R1 and S.
- CEK[* , S] or KEK[* , S] are used when a single CEK or a single KEK is shared with all authorized recipients by a given sender S in a certain context.
- ENC(plaintext, k) refers to the encryption of plaintext with a key k.

5.1. Content Key Distribution with AES Key Wrap

5.1.1. Introduction

The AES Key Wrap (AES-KW) algorithm, as described in [RFC3394], is used to encrypt a randomly generated content-encryption key (CEK) with a pre-shared key-encryption key (KEK). The COSE conventions for using AES-KW are specified in Section 8.5.2 of [RFC9052] and in Section 6.2.1 of [RFC9053]. The encrypted CEK is carried within the COSE_recipient structure, which includes the necessary information for AES-KW. The COSE_recipient structure, a substructure of COSE_Encrypt, contains the CEK encrypted by the KEK.

To ensure high security when using AES Key Wrap, it is important that the KEK is of high entropy and that implementations protect the KEK from disclosure. A compromised KEK could expose all data encrypted with it, including binaries and configuration data.

The COSE_Encrypt structure conveys the information needed to encrypt the payload, including details such as the algorithm and IV. Even though the payload may be conveyed as detached content, the encryption information is still embedded in the COSE_Encrypt.ciphertext structure.

5.1.2. Deployment Options

There are three deployment options for use with AES Key Wrap for payload encryption:

- * If all recipients (typically of the same product family) share the same KEK, a single COSE_recipient structure contains the encrypted CEK. The sender executes the following steps:

1. Fetch KEK[* , S]
2. Generate CEK
3. ENC(CEK, KEK)
4. ENC(payload, CEK)

This deployment option is strongly discouraged. An attacker gaining access to the KEK will be able to encrypt and send payloads to all recipients configured to use this KEK.

- * If recipients have different KEKs, then multiple COSE_recipient structures are included but only a single CEK is used. Each COSE_recipient structure contains the CEK encrypted with the KEKs appropriate for a given recipient. The benefit of this approach is that the payload is encrypted only once with a CEK while there is no sharing of the KEK across recipients. Hence, authorized

recipients still use their individual KEK to decrypt the CEK and to subsequently obtain the plaintext. The steps taken by the sender are:

1. Generate CEK
2. for i=1 to n
 - {
 - 2a. Fetch KEK[Ri, S]
 - 2b. ENC(CEK, KEK[Ri, S])
 - }
3. ENC(payload, CEK)

- * The third option is to use different CEKs encrypted with KEKs of authorized recipients. This approach is appropriate when no benefits can be gained from encrypting and transmitting payloads only once. Assume there are n recipients with their unique KEKs - KEK[R1, S], ..., KEK[Rn, S] and unique CEKs. The sender needs to execute the following steps:

1. for i=1 to n
 - {
 - 1a. Fetch KEK[Ri, S]
 - 1b. Generate CEK[Ri, S]
 - 1c. ENC(CEK[Ri, S], KEK[Ri, S])
 - 1d. ENC(payload, CEK[Ri, S])
 - 2. }

5.1.3. The CDDL of SUIE_Encryption_Info for AES-KW binary

The CDDL for the AES-KW binary is shown in Figure 5. `empty_or_serialized_map` and `header_map` are structures defined in [RFC9052].

```

SUIT_Encryption_Info_AESKW = #6.96([
  protected   : outer_header_map_protected,
  unprotected : outer_header_map_unprotected,
  ciphertext  : bstr / nil,
  recipients  : [ + COSE_recipient_AESKW ]
])

outer_header_map_protected = empty_or_serialized_map
outer_header_map_unprotected = header_map

COSE_recipient_AESKW = [
  protected   : bstr .size 0 / bstr .cbor empty_map,
  unprotected : recipient_header_unpr_map_aeskw,
  ciphertext  : bstr           ; CEK encrypted with KEK
]

empty_map = {}

recipient_header_unpr_map_aeskw =
{
  ? 1 => int / tstr,   ; content encryption algorithm identifier
  ? 4 => bstr,         ; identifier of the KEK
                        ; pre-shared with the recipient
  * label => values    ; extension point
}

```

Figure 5: CDDL for AES-KW-based Content Key Distribution

Note that the AES-KW algorithm, as defined in Section 2.2.3.1 of [RFC3394], does not have public parameters that vary on a per-invocation basis. Hence, the protected header in the COSE_recipient structure is a byte string of zero length.

5.2. Content Key Distribution with Ephemeral-Static Diffie-Hellman

5.2.1. Introduction

Ephemeral-Static Diffie-Hellman (ES-DH) is a public key encryption scheme that enables encryption using the recipient's public key. There are several variations of this scheme; this document adopts the version specified in Section 8.5.5 of [RFC9052].

The structure is composed of two layers:

- * Layer 0: Contains content encrypted with a Content Encryption Key (CEK). The content may be provided separately.

- * Layer 1: Uses the AES Key Wrap (AES-KW) algorithm to encrypt the randomly generated CEK with a Key Encryption Key (KEK) derived via ES-DH. The resulting symmetric key is processed through an HKDF-based key derivation function [RFC5869].

This two-layer structure combines ES-DH with AES-KW and HKDF, referred to as ECDH-ES + AES-KW. An example can be found in Figure 10.

Another variant of the ES-DH algorithm, called ECDH-ES + HKDF, does not utilize AES Key Wrap. However, this version is not covered in this document.

5.2.2. Deployment Options

This approach supports only two deployment options, as it assumes that each recipient is always equipped with a device-specific public/private key pair.

- * When a sender transmits a payload to multiple recipients, all recipients receive the same encrypted payload, meaning the same CEK is used to encrypt the content. For each recipient, a separate COSE_recipient structure is used, which contains the CEK encrypted with the recipient-specific KEK. To derive the KEK, each COSE_recipient structure includes a COSE_recipient_inner structure that carries the sender's ephemeral key and an identifier for the recipient's public key.

The steps taken by the sender are:

1. Generate CEK
 2. for i=1 to n
 - {
 - 2a. Generate KEK[R_i, S] using ES-DH
 - 2b. ENC(CEK, KEK[R_i, S])
 - }
 3. ENC(payload, CEK)
- * The alternative is to encrypt each device specific payload with a unique content encryption key (CEK), resulting in a manifest per device specific payload. This approach is useful when payloads contain device-specific information or when the optimization in previous approach are not applicable or not valuable enough. In this case, the encryption operation becomes ENC(payload_i, CEK[R_i, S]) where each recipient R_i receives a unique CEK. Assume that KEK[R₁, S], ..., KEK[R_n, S] have been generated for the recipients using ES-DH. The sender must then follow these steps:

1. for i=1 to n
 - {
 - 1a. Generate KEK[Ri, S] using ES-DH
 - 1b. Generate CEK[Ri, S]
 - 1c. ENC(CEK[Ri, S], KEK[Ri, S])
 - 1d. ENC(payload, CEK[Ri, S])
 - }

5.2.3. The CDDL of SUIF_Encryption_Info for ES-DH binary

The CDDL for the ECDH-ES+AES-KW binary is provided in Figure 6. Only the essential parameters are included. The structures `empty_or_serialized_map` and `header_map` are defined in [RFC9052].

```
SUIF_Encryption_Info_ESDH = #6.96([
  protected   : outer_header_map_protected,
  unprotected : outer_header_map_unprotected,
  ciphertext   : bstr / nil,
  recipients   : [ + COSE_recipient_ESDH ]
])

outer_header_map_protected = empty_or_serialized_map
outer_header_map_unprotected = header_map

COSE_recipient_ESDH = [
  protected   : bstr .cbor recipient_header_map_esdh,
  unprotected : recipient_header_unpr_map_esdh,
  ciphertext   : bstr           ; CEK encrypted with KEK
]

recipient_header_map_esdh =
{
  ? 1 => int / tstr, ; content encryption algorithm identifier
  * label => values   ; extension point
}

recipient_header_unpr_map_esdh =
{
  ? 4 => bstr,           ; identifier of the recipient public key
  -1 => COSE_Key,       ; ephemeral public key for the sender
  * label => values     ; extension point
}
```

Figure 6: CDDL for ES-DH-based Content Key Distribution

See Section 6 for a description on how to encrypt the payload.

5.2.4. Context Information Structure

The context information structure ensures that the derived keying material is "bound" to the specific context of the transaction. This specification reuses the structure defined in Section 5.2 of [RFC9053], with modifications to fit the current use case.

The following elements are bound to the context:

- * the protocol employing the key-derivation method,
- * information about the utilized AES Key Wrap algorithm, and the key length.
- * the protected header field, which contains the content key encryption algorithm.

The sender and recipient identities are left empty.

The following fields in Figure 7 require an explanation:

- * The COSE_KDF_Context.AlgorithmID field MUST contain the identifier for the AES Key Wrap algorithm being used. This specification uses the following values: A128KW (value -3), A192KW (value -4), or A256KW (value -5)
- * The COSE_KDF_Context.SuppPubInfo.keyDataLength field MUST specify the key length, in bits, corresponding to the algorithm in the AlgorithmID field. For A128KW the value is 128, for A192KW the value is 192, and for A256KW the value 256.
- * The COSE_KDF_Context.SuppPubInfo.other field captures the protocol that uses the ES-DH content key distribution algorithm. It MUST be set to the constant string "SUIIT Payload Encryption".
- * The COSE_KDF_Context.SuppPubInfo.protected field MUST contain the serialized content of the recipient_header_map_esdh field, which contains (among other elements) the identifier of the content key distribution method.

```
COSE_KDF_Context = [  
  AlgorithmID : int,  
  PartyUInfo : [ PartyInfoSender ],  
  PartyVInfo : [ PartyInfoRecipient ],  
  SuppPubInfo : [  
    keyDataLength : uint,  
    protected : bstr,  
    other: 'SUIT Payload Encryption'  
  ],  
  ? SuppPrivInfo : bstr  
]  
  
PartyInfoSender = (  
  identity : nil,  
  nonce : nil,  
  other : nil  
)  
  
PartyInfoRecipient = (  
  identity : nil,  
  nonce : nil,  
  other : nil  
)
```

Figure 7: CDDL for COSE_KDF_Context Structure

The HKDF-based key derivation function MAY contain a salt value, as described in Section 5.1 of [RFC9053]. This optional value influences the key generation process, though this specification does not require the use of a salt. If the salt is public and included in the message, the "salt" algorithm header parameter MUST be used. The salt adds extra randomness to the KDF context. When the salt is transmitted via the "salt" algorithm header parameter, the receiver MUST be capable of processing it and MUST pass it into the key derivation function. For more details on salt usage, refer to [RFC5869] and NIST SP800-56 [SP800-56].

Profiles of this specification MAY define an extended version of the context information structure or MAY employ a different context information structure.

6. Content Encryption

This section summarizes the steps involved in content encryption, applicable to both content key distribution methods.

When using AEAD ciphers, such as AES-GCM or ChaCha20/Poly1305, the COSE specification requires a consistent byte stream to create the authenticated data structure. This structure is illustrated in Figure 8 and defined in Section 5.3 of [RFC9052].

```
Enc_structure = [  
  context : "Encrypt",  
  protected : empty_or_serialized_map,  
  external_aad : bstr  
]
```

Figure 8: CDDL for Enc_structure Data Structure

This Enc_structure must be populated as follows:

- * The protected field in the Enc_structure from Figure 8 refers to the content of the protected field in the COSE_Encrypt structure.
- * The value of external_aad MUST be set to a zero-length byte string, represented as h'' in diagnostic notation and encoded as 0x40.

Some ciphers, such as AES-CTR, provide confidentiality without integrity protection (see [RFC9459]). For these ciphers, the Enc_structure shown in Figure 8 cannot be used, as the Additional Authenticated Data (AAD) byte string is only applicable to AEAD ciphers. Therefore, the AAD structure is not passed to the API for these ciphers, and the protected header in the SUIT_Encryption_Info structure MUST be a zero-length byte string.

6.1. AES-GCM

6.1.1. Introduction

AES-GCM is an AEAD cipher and provides confidentiality and integrity protection.

Examples in this section use the following parameters:

- * Algorithm for payload encryption: AES-GCM-128
 - k: h'15F785B5C931414411B4B71373A9C0F7'
 - IV: h'F14AAB9D81D51F7AD943FE87AF4F70CD'
- * Plaintext: "This is a real firmware image."

- in hex:
546869732069732061207265616C206669726D7761726520696D6167652E

6.1.2. AES-KW + AES-GCM Example

This example uses the following parameters:

- * Algorithm id for key wrap: A128KW
- * KEK COSE_Key (Secret Key):
 - kty: Symmetric
 - k: 'aaaaaaaaaaaaaaaa'
 - kid: 'kid-1'

The COSE_Encrypt structure, in hex format, is (with a line break inserted):

```
D8608443A10101A1054CF14AAB9D81D51F7AD943FE87F6818340A2012204
456B69642D31581875603FFC9518D794713C8CA8A115A7FB32565A6D5953
4D62
```

The resulting COSE_Encrypt structure in a diagnostic format is shown in Figure 9.

```
96([
  / protected: / << {
    / alg / 1: 1 / A128GCM /
  } >>,
  / unprotected: / {
    / IV / 5: h'F14AAB9D81D51F7AD943FE87'
  },
  / ciphertext: / null / detached ciphertext /,
  / recipients: / [
    [
      / protected: / h'',
      / unprotected: / {
        / alg / 1: -3 / A128KW /,
        / kid / 4: 'kid-1'
      },
      / ciphertext: /
        h'75603FFC9518D794713C8CA8A115A7FB32565A6D59534D62'
      / CEK encrypted with KEK /
    ]
  ]
])
```

Figure 9: COSE_Encrypt Example for AES Key Wrap

The encrypted payload (with a line feed added) was:

```
758C4B7BBAE2C4C1D462423E0F0DC3164FFA7B85BB94D4BD6D7ED26AB32F
EB063385D4D3465927EC82CB5E198A59
```

6.1.3. ECDH-ES+AES-KW + AES-GCM Example

This example uses the following parameters:

- * Algorithm for content key distribution: ECDH-ES + A128KW
- * KEK COSE_Key (Receiver's Private Key):
 - kty: EC2
 - crv: P-256
 - x: h' 5886CD61DD875862E5AAA820E7A15274C968A9BC96048DDCACE32F50C3651BA3'
 - y: h' 9EED8125E932CD60C0EAD3650D0A485CF726D378D1B016ED4298B2961E258F1B'
 - d: h' 60FE6DD6D85D5740A5349B6F91267EEAC5BA81B8CB53EE249E4B4EB102C476B3'
 - kid: 'kid-2'
- * KDF Context
 - Algorithm ID: -3 (A128KW)
 - SuppPubInfo
 - o keyDataLength: 128
 - o protected: << { / alg / 1: -29 / ECDH-ES+A128KW / } >>
 - o other: 'SUIE Payload Encryption'

The COSE_Encrypt structure, in hex format, is (with a line break inserted):

```
D8608443A10101A1054CF14AAB9D81D51F7AD943FE87F6818344A101381C
A120A40102200121582073024F415AA51529A66CCEFD88F3F62A734492FF
45F6AD37FD2888E73EAF19DA2258204005B48A6FD091AA6ABFE3CFBEEDE8
8B347E521D43405FDBD7D2CFF0EBC21B265818A06B8E6550F308712B1DF0
44B21B7D11D9B22792F1DE0997
```

The resulting COSE_Encrypt structure in a diagnostic format is shown in Figure 10.

```
96([
  / protected: / << {
    / alg / 1: 1 / A128GCM /
  } >>,
  / unprotected: / {
    / IV / 5: h'F14AAB9D81D51F7AD943FE87'
  },
  / ciphertext: / null / detached ciphertext /,
  / recipients: / [
    [
      / protected: / << {
        / alg / 1: -29 / ECDH-ES + A128KW /
      } >>,
      / unprotected: / {
        / ephemeral key / -1: {
          / kty / 1: 2 / EC2 /,
          / crv / -1: 1 / P-256 /,
          / x / -2: h'73024F415AA51529A66CCEFD88F3F62A
            734492FF45F6AD37FD2888E73EAF19DA',
          / y / -3: h'4005B48A6FD091AA6ABFE3CFBEEDE88B
            347E521D43405FDBD7D2CFF0EBC21B26'
        }
      },
      / ciphertext: /
        h'A06B8E6550F308712B1DF044B21B7D11D9B22792F1DE0997'
      / CEK encrypted with KEK /
    ]
  ]
])
```

Figure 10: COSE_Encrypt Example for ES-DH

The encrypted payload (with a line feed added) was:

```
758C4B7BBAE2C4C1D462423E0F0DC3164FFA7B85BB94D4BD6D7ED26AB32F
EB063385D4D3465927EC82CB5E198A59
```

6.2. AES-CTR

6.2.1. Introduction

AES-CTR is a non-AEAD cipher that provides confidentiality but lacks integrity protection. Unlike AES-CBC, AES-CTR uses an IV per block, as shown in Figure 11. Hence, when an image is encrypted using AES-CTR-128 or AES-CTR-256, the counter value MUST start with the IV value and incremented by one for each 16-byte plaintext block. The IV value MAY be provided by the COSE header field or is communicated via out-of-band means, for example by setting it to a given value (e.g. the value of zero). Firmware authors MUST make sure that the same IV and AES content key encryption combination is not used more than once. Communicating the IV value inside the COSE header is RECOMMENDED.

The following abbreviations are used in Figure 11:

- * P_i = Plaintext blocks
- * C_i = Ciphertext blocks
- * E = Encryption function
- * k = Symmetric key
- * $\hat{\wedge}212\225$ = XOR operation

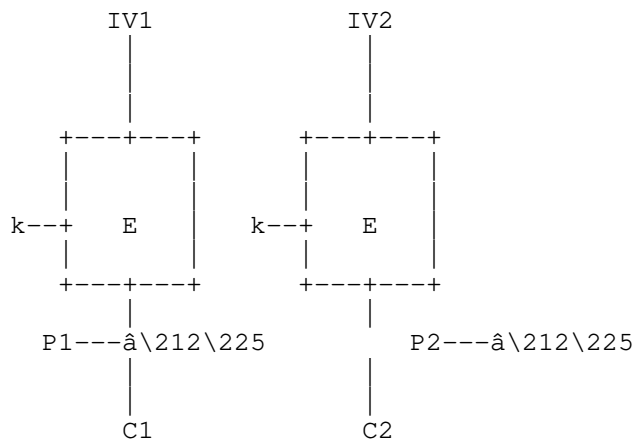


Figure 11: AES-CTR Operation

Examples in this section use the following parameters:

- * Algorithm for payload encryption: AES-CTR-128

- k: h'261DE6165070FB8951EC5D7B92A065FE'
- IV: h'DAE613B2E0DC55F4322BE38BDBA9DC68'
- * Plaintext: "This is a real firmware image."
- in hex:
546869732069732061207265616C206669726D7761726520696D6167652E

6.2.2. AES-KW + AES-CTR Example

This example uses the following parameters:

- * Algorithm id for key wrap: A128KW
- * KEK COSE_Key (Secret Key):
 - kty: Symmetric
 - k: 'aaaaaaaaaaaaaaaa'
 - kid: 'kid-1'

The COSE_Encrypt structure, in hex format, is (with a line break inserted):

```
D8608440A20139FFFD0550DAE613B2E0DC55F4322BE38BDBA9DC68F68183
40A2012204456B69642D315818CE34035CE5C2E2666E46D4C131FC561DD1
90A6D26CFA1990
```

The resulting COSE_Encrypt structure in a diagnostic format is shown in Figure 12.

```

96([
  / protected: / h'',
  / unprotected: / {
    / alg / 1: -65534 / A128CTR /,
    / IV / 5: h'DAE613B2E0DC55F4322BE38BDBA9DC68'
  },
  / ciphertext: / null / detached ciphertext /,
  / recipients: / [
    [
      / protected: / h'',
      / unprotected: / {
        / alg / 1: -3 / A128KW /,
        / kid / 4: 'kid-1'
      },
      / ciphertext: /
        h'CE34035CE5C2E2666E46D4C131FC561DD190A6D26CFA1990'
        / CEK encrypted with KEK /
    ]
  ]
])

```

Figure 12: COSE_Encrypt Example for AES Key Wrap

The encrypted payload (with a line feed added) was:

```
2BB8DB522AE978246CC775C3B0241BD4B0333FFDD2DB70C7EE7A4966E3B7
```

6.2.3. ECDH-ES+AES-KW + AES-CTR Example

This example uses the following parameters:

- * Algorithm for content key distribution: ECDH-ES + A128KW
- * KEK COSE_Key (Receiver's Private Key):
 - kty: EC2
 - crv: P-256
 - x: h'5886CD61DD875862E5AAA820E7A15274C968A9BC96048DDCACE32F50C3651BA3'
 - y: h'9EED8125E932CD60C0EAD3650D0A485CF726D378D1B016ED4298B2961E258F1B'
 - d: h'60FE6DD6D85D5740A5349B6F91267EEAC5BA81B8CB53EE249E4B4EB102C476B3'

- kid: 'kid-2'
- * KDF Context
 - Algorithm ID: -3 (A128KW)
 - SuppPubInfo
 - o keyDataLength: 128
 - o protected: << { / alg / 1: -29 / ECDH-ES+A128KW / } >>
 - o other: 'SUIE Payload Encryption'

The COSE_Encrypt structure, in hex format, is (with a line break inserted):

```
D8608440A20139FFFD0550DAE613B2E0DC55F4322BE38BDBA9DC68F68183
44A101381CA120A401022001215820EE0718F6B019C29CC611C18CEDE221
4066DDCEDC2F0DBEF873CB224C715C1174225820279F2A88E4AB9E2ED30C
0FCB69515B31B5D36725BFDB9AE02032ED4D5AB52CB85818E28B4502E4F5
151884A995405579006E9465C3E94E3E0808
```

The resulting COSE_Encrypt structure in a diagnostic format is shown in Figure 13.

```

96([
  / protected: / h'',
  / unprotected: / {
    / alg / 1: -65534 / A128CTR /,
    / IV / 5: h'DAE613B2E0DC55F4322BE38BDBA9DC68'
  },
  / ciphertext: / null / detached ciphertext /,
  / recipients: / [
    [
      / protected: / << {
        / alg / 1: -29 / ECDH-ES + A128KW /
      } >>,
      / unprotected: / {
        / ephemeral key / -1: {
          / kty / 1: 2 / EC2 /,
          / crv / -1: 1 / P-256 /,
          / x / -2: h'EE0718F6B019C29CC611C18CEDE22140
                    66DDCEDC2F0DBEF873CB224C715C1174',
          / y / -3: h'279F2A88E4AB9E2ED30C0FCB69515B31
                    B5D36725BFDB9AE02032ED4D5AB52CB8'
        }
      },
      / ciphertext: /
        h'E28B4502E4F5151884A995405579006E9465C3E94E3E0808'
        / CEK encrypted with KEK /
    ]
  ]
])

```

Figure 13: COSE_Encrypt Example for ES-DH

The encrypted payload (with a line feed added) was:

```
2BB8DB522AE978246CC775C3B0241BD4B0333FFDD2DB70C7EE7A4966E3B7
```

7. Integrity Check on Encrypted and Decrypted Payloads

In addition to `suit-condition-image-match` (see Section 8.4.9.2 of [I-D.ietf-suit-manifest]), AEAD algorithms used for content encryption provides another way to validate the integrity of components. This section provides a guideline to construct secure but not redundant SUIT Manifest for encrypted payloads.

7.1. Validating Payload Integrity

This sub-section explains three ways to validate the integrity of payloads.

7.1.1. Image Match after Decryption

The `suit-condition-image-match` on the plaintext payload is used after decryption. An example command sequence is shown in Figure 14.

```
/ directive-set-component-index / 12, 1,  
/ directive-override-parameters / 20, {  
  / parameter-uri / 21: "coaps://example.com/encrypted.bin"  
},  
/ directive-fetch / 21, 15,  
  
/ directive-set-component-index / 12, 0,  
/ directive-override-parameters / 20, {  
  / parameter-image-digest / 3: << {  
    / algorithm-id: / -16 / SHA256 /,  
    / digest-bytes: / h'3B1...92A' / digest of plaintext payload /  
  } >>,  
  / parameter-image-size / 14: 30 / size of plaintext payload /,  
  / parameter-encryption-info / TBD19: h'369...50F',  
  / parameter-source-component / 22: 1  
},  
/ directive-copy / 22, 15,  
/ condition-image-match / 3, 15 / check decrypted payload integrity /
```

Figure 14: Check Image Match After Decryption

RFC Editor's Note (TBD19): The value for the `suit-parameter-encryption-info` parameter is set to 19, as the proposed value.

7.1.2. Image Match before Decryption

The `suit-condition-image-match` can also be applied on encrypted payloads before decryption takes place. An example command sequence is shown in Figure 15.

This option mitigates battery exhaustion attacks discussed in Section 11.

```
/ directive-set-component-index / 12, 1,  
/ directive-override-parameters / 20, {  
  / parameter-image-digest / 3: << {  
    / algorithm-id: / -16 / SHA256 /,  
    / digest-bytes: / h'8B4...D34' / digest of encrypted payload /  
  } >>,  
  / parameter-image-size / 14: 30 / size of encrypted payload /,  
  / parameter-uri / 21: "coaps://example.com/encrypted.bin"  
},  
  
/ directive-fetch / 21, 15,  
/ condition-image-match / 3, 15 / check decrypted payload integrity /,  
  
/ directive-set-component-index / 12, 0,  
/ directive-override-parameters / 20, {  
  / parameter-encryption-info / TBD19: h'D86...1F0',  
  / parameter-source-component / 22: 1  
},  
/ directive-copy / 22, 15
```

Figure 15: Check Image Match Before Decryption

RFC Editor's Note (TBD19): The value for the `suit-parameter-encryption-info` parameter is set to 19, as the proposed value.

7.1.3. Checking Authentication Tag while Decrypting

AEAD algorithms, such as AES-GCM and ChaCha20/Poly1305, verify the integrity of the encrypted content.

7.2. Payload Integrity Validation

This subsection offers guidelines for validating the integrity of payloads within the SUIT manifest. The decision tree in Figure 16 illustrates the process for establishing payload integrity.

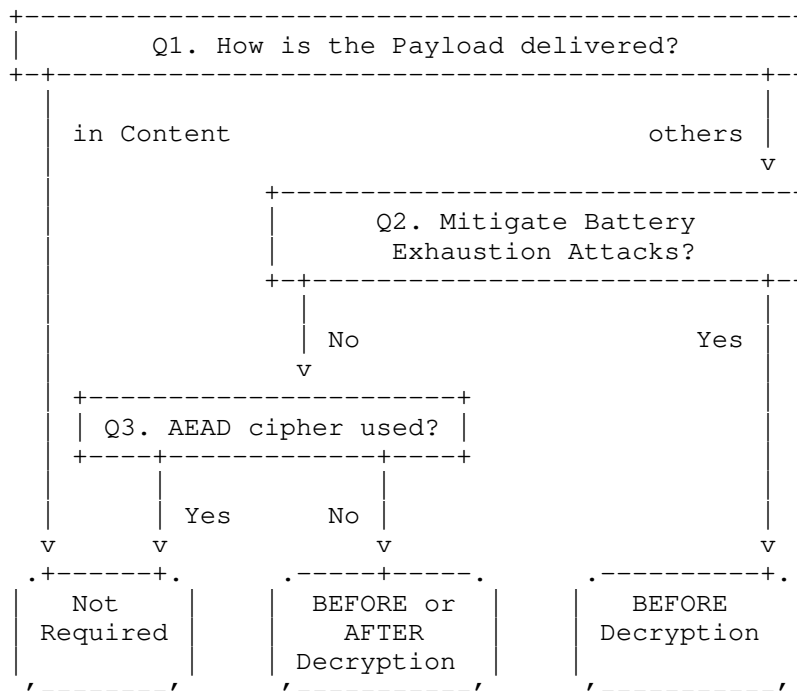


Figure 16: Decision Tree: Validating the Payload

There are three questions to ask:

- * Q1. How does the recipient receive the encrypted payload? If the encrypted payload is used as the value of the `suit-parameter-content`, its integrity is already verified by the `suit-authentication-wrapper`. Therefore, no additional integrity check is required. However, if the encrypted payload is delivered via `suit-directive-fetch` from an integrated payload or from outside the SUIT envelope, for example `"coaps://example.com/encrypted.bin"`, additional considerations must be addressed.
- * Q2. Are battery exhaustion attacks a concern? If yes, the integrity of the encrypted payload must be checked before the payload is decrypted. If no, then other questions need to be asked.

- * Q3. Is the payload encrypted with an AEAD cipher? If yes, no additional integrity check is required, as the recipient verifies the payload's integrity during decryption. If no, integrity validation can occur either before or after decryption. However, validating integrity before decryption is RECOMMENDED especially for the AES-CTR mode (see Section 8 of [RFC9459]).

8. Firmware Updates on IoT Devices with Flash Memory

Embedded devices come in many forms, and the market is both large and fragmented. As a result, some implementations and deployments may adopt firmware update procedures that differ from the descriptions provided here. On a positive note, the SUIT manifest accommodates various deployment scenarios, thanks to the "scripting" functionality offered by its commands.

This section specifically addresses firmware images on microcontrollers and does not pertain to generic software, configuration data, or machine learning models. The differences arise from two main aspects:

- * **Use of Flash Memory:** Flash memory in microcontrollers is a type of non-volatile memory that typically erases data in larger units called blocks, pages, or sectors, and rewrites data at the byte level (often 4 bytes) or larger units. Furthermore, flash memory is segmented into different regions, storing the bootloader, various versions of firmware images (in designated slots), and configuration data. An example layout of a microcontroller flash area is illustrated in Figure 17.
- * **Microcontroller Design:** Code on microcontrollers typically cannot be executed from arbitrary locations in flash memory without additional software development and design efforts. Consequently, developers often compile firmware so that the bootloader can execute code from a specific location in flash memory, commonly referred to as the "primary slot."

Once the encrypted firmware image is transferred to the device, it is usually stored in a dedicated area known as the "secondary slot."

During the next boot, the bootloader detects the new firmware image and begins decrypting it sector by sector, swapping it with the image located in the primary slot. This method of swapping the newly downloaded image with the previously valid one requires two slots, allowing for a rollback if the new firmware fails to boot, thereby enhancing the robustness of the firmware update process.

The swap occurs only after verifying the signature on the plaintext. It is important to note that the plaintext firmware image is available in the primary slot only after the swap is completed, unless "dummy decrypt" is used to compute the hash over the plaintext prior to executing the decryption during the swap. In this context, dummy decryption refers to decrypting the firmware image in the secondary slot sector by sector while computing a rolling hash over the resulting plaintext (also sector by sector) without performing the swap operation. Although performance optimizations, such as conveying hashes for each sector in the manifest rather than a hash of the entire firmware image, are possible, these optimizations are not detailed in this specification.

Without hardware-based, on-the-fly decryption, the image in the primary slot is available in cleartext and may need to be re-encrypted before copying it to the secondary slot. This step might be necessary if the secondary slot has different access permissions or is located in off-chip flash memory, which tends to be more vulnerable to physical attacks.

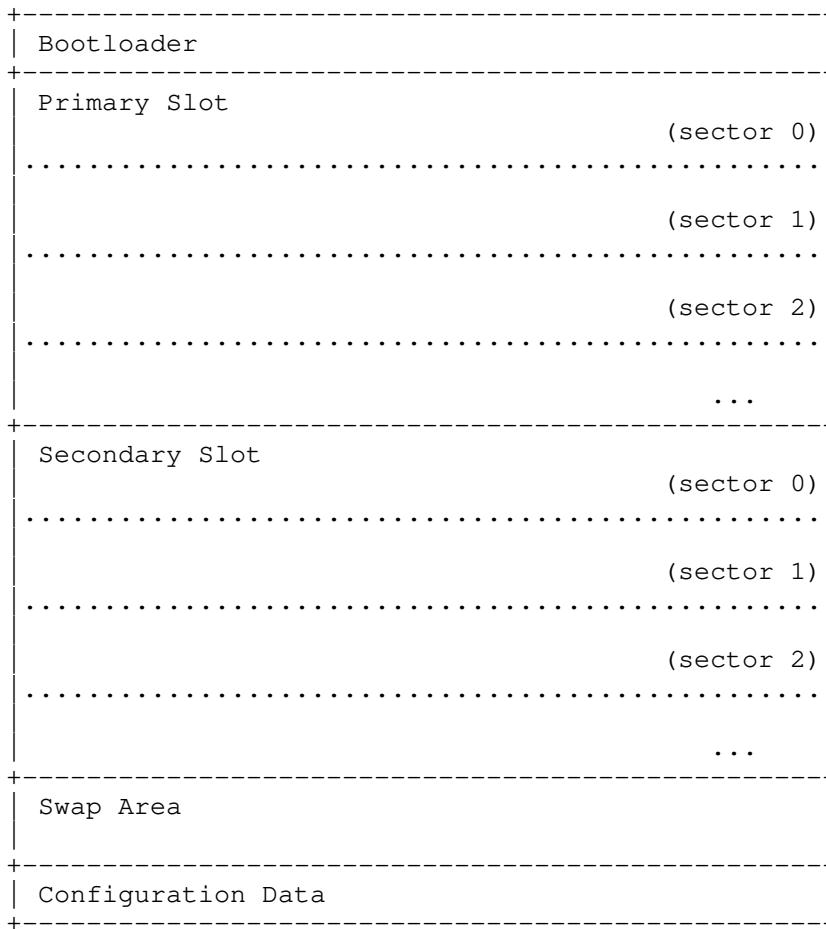


Figure 17: Example Flash Area Layout

The ability to resume an interrupted firmware update is often essential for unattended devices, including low-end, constrained IoT devices. To meet this requirement, a firmware image must be divided into sectors, with each sector encrypted individually using a cipher that does not increase the size of the resulting ciphertext (i.e., by avoiding the addition of an authentication tag after each encrypted block).

If an update is aborted while the bootloader is decrypting the newly received image and swapping the sectors, the bootloader can restart from where it left off. This technique enhances robustness and performance.

For this purpose, ciphers without integrity protection are employed to encrypt the firmware image. It is crucial that integrity protection for the firmware image is provided, and the `suit-parameter-image-digest`, defined in Section 8.4.8.6 of [I-D.ietf-suit-manifest], MUST be utilized.

[RFC9459] specifies the AES Counter (AES-CTR) mode and AES Cipher Block Chaining (AES-CBC) ciphers, both of which do not provide integrity protection. These ciphers are suitable for firmware encryption in IoT devices. However, for many other scenarios involving software packages, configuration information, or personalization data, the use of AEAD ciphers is RECOMMENDED.

The following subsections offer additional information on the selection of initialization vectors (IVs) for use with AES-CTR in the context of firmware encryption. A random CEK MUST be used with every plaintexts, as specified in Section 5, since the IVs are not random but are instead based on the slot/sector combination in flash memory. The discussion assumes that the block size of AES is significantly smaller than the sector size. Typically, flash memory sectors are measured in KiB, necessitating the decryption of multiple AES blocks to complete the decryption of an entire sector.

To offer a specific example, let us assume the slot size of a specific flash controller on an IoT device is 64 KiB, the sector size 4096 bytes (4 KiB) and an AES plaintext block size of 16 bytes. The counter values used with AES-CTR range from IV+0 to IV+255 in the first sector, and $16 * 256$ counter values are required for the slot. This

IV value is either communicated in the COSE header or via out-of-band means.

9. Complete Examples

The following manifests illustrate how to deliver an encrypted payload along with its encryption information to devices.

In the AES-KW examples, HMAC-256 MACs are included, utilizing the following secret key:

```
'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'  
(616161... in hex, and its length is 32)
```

ES-DH 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 SHA-256 as the digest function.

9.1. AES Key Wrap Example with Write Directive

The following SUIF manifest instructs a parser to authenticate the manifest using COSE_Mac0 with HMAC256. It also directs the parser to write and decrypt the encrypted payload into a component using the `suit-directive-write` directive.

The SUIF manifest in diagnostic notation (with line breaks added for clarity) is displayed below:

```

/ 1/ / SUIT_Envelope_Tagged / 107({
/ 2/   / authentication-wrapper / 2: << [
/ 3/     << [
/ 4/       / digest-algorithm-id: / -16 / SHA256 /,
/ 5/       / digest-bytes: / h'037A5C325CE14078A0AADF007428EAC6
/ 6/                                     59361AD9402A732410BDA542FAE94E2C'
/ 7/     ] >>,
/ 8/     << / COSE_Mac0_Tagged / 17([
/ 9/       / protected: / << {
/ 10/        / algorithm-id / 1: 5 / HMAC256 /
/ 11/       } >>,
/ 12/       / unprotected: / {},
/ 13/       / payload: / null,
/ 14/       / tag: / h'8D92599011C451A4C5FB69709FA6CA6C
/ 15/                                     0F846D692BDBB3F624EC91F82F9F620A'
/ 16/     ]) >>
/ 17/   ] >>,
/ 18/ / manifest / 3: << {
/ 19/   / manifest-version / 1: 1,
/ 20/   / manifest-sequence-number / 2: 1,
/ 21/   / common / 3: << {
/ 22/     / components / 2: [
/ 23/       ['plaintext-firmware']

```

```

/ 24/      ]
/ 25/      } >>,
/ 26/      / install / 20: << [
/ 27/          / fetch encrypted firmware /
/ 28/          / directive-override-parameters / 20, {
/ 29/              / parameter-content / 18:
/ 30/                  h'758C4B7BBAE2C4C1D462423E0F0DC3164FFA7B85BB94D4
/ 31/                  BD6D7ED26AB32FEB063385D4D3465927EC82CB5E198A59',
/ 32/          / parameter-encryption-info / 19: << 96([
/ 33/              / protected: / << {
/ 34/                  / alg / 1: 1 / A128GCM /
/ 35/              } >>,
/ 36/          / unprotected: / {
/ 37/              / IV / 5: h'F14AAB9D81D51F7AD943FE87'
/ 38/          },
/ 39/          / ciphertext: / null / detached ciphertext /,
/ 40/          / recipients: / [
/ 41/              [
/ 42/                  / protected: / h'',
/ 43/                  / unprotected: / {
/ 44/                      / alg / 1: -3 / A128KW /,
/ 45/                      / kid / 4: 'kid-1'
/ 46/                  },
/ 47/                  / ciphertext: /
/ 48/                      h'75603FFC9518D794713C8CA8
/ 49/                      A115A7FB32565A6D59534D62'
/ 50/                      / CEK encrypted with KEK /
/ 51/              ]
/ 52/          ]
/ 53/      ] ) >>
/ 54/      },
/ 55/
/ 56/          / decrypt encrypted firmware /
/ 57/          / directive-write / 18, 15
/ 58/          / consumes the SUIT_Encryption_Info above /
/ 59/      ] >>
/ 60/  } >>
/ 61/ })

```

In hex format, the SUIT manifest is:

```
D86BA2025853825824822F5820037A5C325CE14078A0AADF007428EAC659
361AD9402A732410BDA542FAE94E2C582AD18443A10105A0F658208D9259
9011C451A4C5FB69709FA6CA6C0F846D692BDBB3F624EC91F82F9F620A03
5898A4010102010357A102818152706C61696E746578742D6669726D7761
72651458778414A212582E758C4B7BBAE2C4C1D462423E0F0DC3164FFA7B
85BB94D4BD6D7ED26AB32FEB063385D4D3465927EC82CB5E198A5913583E
D8608443A10101A1054CF14AAB9D81D51F7AD943FE87F6818340A2012204
456B69642D31581875603FFC9518D794713C8CA8A115A7FB32565A6D5953
4D62120F
```

9.2. AES Key Wrap Example with Fetch + Copy Directives

The following SUIT manifest instructs a parser to fetch and store the encrypted payload. Subsequently, the payload is decrypted and copied into another component using the `suit-directive-copy` directive. This approach is particularly effective for constrained devices with `execute-in-place` (XIP) flash memory.

The SUIT manifest in diagnostic notation (with line breaks added for clarity) is displayed below:

```
/ 1/ / SUIT_Envelope_Tagged / 107({
/ 2/ / authentication-wrapper / 2: << [
/ 3/ << [
/ 4/ / digest-algorithm-id: / -16 / SHA256 /,
/ 5/ / digest-bytes: / h'3C92AECEAA7225DDD5129A83B2842BF2
/ 6/ 8CC53B2C9467C5BF256E7108F2DA7C9C'
/ 7/ ] >>,
/ 8/ << / COSE_Mac0_Tagged / 17([
/ 9/ / protected: / << {
/ 10/ / algorithm-id / 1: 5 / HMAC256 /
/ 11/ } >>,
/ 12/ / unprotected: / {},
/ 13/ / payload: / null,
/ 14/ / tag: / h'46CB34181A04B967023D4C9E136DC5DC
/ 15/ 591D8A9BE9365DE4D282C9D6168C01FB'
/ 16/ ]) >>
/ 17/ ] >>,
/ 18/ / manifest / 3: << {
/ 19/ / manifest-version / 1: 1,
/ 20/ / manifest-sequence-number / 2: 1,
/ 21/ / common / 3: << {
/ 22/ / components / 2: [
/ 23/ ['plaintext-firmware'],
/ 24/ ['encrypted-firmware']
/ 25/ ]
/ 26/ } >>,
/ 27/ / install / 20: << [
```

```
/ 28/      / fetch encrypted firmware /
/ 29/      / directive-set-component-index / 12,
/ 30/      / 1 / ['encrypted-firmware'] /,
/ 31/      / directive-override-parameters / 20, {
/ 32/      /   parameter-image-size / 14: 46,
/ 33/      /   parameter-uri / 21:
/ 34/      /     "coaps://example.com/encrypted-firmware"
/ 35/      / },
/ 36/      / directive-fetch / 21, 15,
/ 37/
/ 38/      / decrypt encrypted firmware /
/ 39/      / directive-set-component-index / 12,
/ 40/      / 0 / ['plaintext-firmware'] /,
/ 41/      / directive-override-parameters / 20, {
/ 42/      /   parameter-encryption-info / 19: << 96([
/ 43/      /     / protected: / << {
/ 44/      /       / alg / 1: 1 / A128GCM /
/ 45/      /     } >>,
/ 46/      /     / unprotected: / {
/ 47/      /       / IV / 5: h'F14AAB9D81D51F7AD943FE87'
/ 48/      /     },
/ 49/      /     / ciphertext: / null / detached ciphertext /,
/ 50/      /     / recipients: / [
/ 51/      /       [
/ 52/      /         / protected: / h'',
/ 53/      /         / unprotected: / {
/ 54/      /           / alg / 1: -3 / A128KW /,
/ 55/      /           / kid / 4: 'kid-1'
/ 56/      /         },
/ 57/      /         / ciphertext: /
/ 58/      /           h'75603FFC9518D794713C8CA8
/ 59/      /           A115A7FB32565A6D59534D62'
/ 60/      /         / CEK encrypted with KEK /
/ 61/      /       ]
/ 62/      /     ]
/ 63/      /   ] >>,
/ 64/      /   parameter-source-component / 22:
/ 65/      /     1 / ['encrypted-firmware'] /
/ 66/      /   },
/ 67/      /   directive-copy / 22,
/ 68/      /     15 / consumes the SUIT_Encryption_Info above /
/ 69/      /   ] >>
/ 70/      / } >>
/ 71/      / })
```

The default storage area is defined by the component identifier (see Section 8.4.5.1 of [I-D.ietf-suit-manifest]). In this example, the component identifier for component #0 is ['plaintext-firmware'] and the file path "/plaintext-firmware" is the expected location.

While parsing the manifest, the behavior of SUIT manifest processor would be

- * [L2-L17] authenticates the manifest part on [L18-L68]
- * [L22-L25] gets two component identifiers; ['plaintext-firmware'] for component #0, and ['encrypted-firmware'] for component # 1 respectively
- * [L29] sets current component index # 1 (the lasting directives target ['encrypted-firmware'])
- * [L33-L34] sets source uri parameter "coaps://example.com/encrypted-firmware"
- * [L36] fetches content from source uri into ['encrypted-firmware']
- * [L39] sets current component index # 0 (the lasting directives target ['plaintext-firmware'])
- * [L42-L62] sets SUIT encryption info parameter
- * [L63-L64] sets source component index parameter # 1
- * [L66] decrypts component # 1 (source component index) and stores the result into component # 0 (current component index)

Table 1 lists the features from the SUIT manifest specification, which are re-used by this specification.

Feature Name	Abbr.	Manifest Ref.
component identifier	CI	Sec. 8.4.5.1
(destination) component index	dst-CI	Sec. 8.4.10.1
(destination) component slot OPTIONAL param	dst-CS	Sec. 8.4.8.8
(source) uri OPTIONAL parameter	src-URI	Sec. 8.4.8.10
source component index OPTIONAL parameter	src-CI	Sec. 8.4.8.11

Table 1: Example Flash Area Layout

The resulting state of the SUIT manifest processor is shown in Table 2.

Abbreviation	Plaintext	Ciphertext
CI	['plaintext-firmware']	['encrypted-firmware']
dst-CI	0	1
dst-CS	N/A	N/A
src-URI	N/A	"coaps://example.com/ encrypted-firmware"
src-CI	1	N/A

Table 2: Manifest Processor State

In hex format, the SUIT manifest shown above is:

```
D86BA2025853825824822F58203C92AECEAA7225DDD5129A83B2842BF28C
C53B2C9467C5BF256E7108F2DA7C9C582AD18443A10105A0F6582046CB34
181A04B967023D4C9E136DC5DC591D8A9BE9365DE4D282C9D6168C01FB03
58B2A40101020103582BA102828152706C61696E746578742D6669726D77
6172658152656E637279707465642D6669726D7761726514587C8C0C0114
A20E182E157826636F6170733A2F2F6578616D706C652E636F6D2F656E63
7279707465642D6669726D77617265150F0C0014A213583ED8608443A101
01A1054CF14AAB9D81D51F7AD943FE87F6818340A2012204456B69642D31
581875603FFC9518D794713C8CA8A115A7FB32565A6D59534D621601160F
```

The encrypted payload (with a line feed added) to be fetched from "coaps://example.com/encrypted-firmware" is:

```
758C4B7BBAE2C4C1D462423E0F0DC3164FFA7B85BB94D4BD6D7ED26AB32F
EB063385D4D3465927EC82CB5E198A59
```

The previous example does not utilize storage slots. However, it is possible to implement this functionality for devices that support slots in flash memory. In the enhanced example below, we reference the slots using [h'00'] and [h'01']. In this context, the component identifier [h'00'] designates component slot #0.

```
/ 1/ / SUIT_Envelope_Tagged / 107({
/ 2/   / authentication-wrapper / 2: << [
/ 3/     << [
/ 4/       / digest-algorithm-id: / -16 / SHA256 /,
/ 5/       / digest-bytes: / h'6D74BD3110A2573236E03DD78693D5B2
/ 6/                                     1C299C917A4327D9939DDF3582A41DE3'
/ 7/     ] >>,
/ 8/     << / COSE_Mac0_Tagged / 17([
/ 9/       / protected: / << {
/ 10/        / algorithm-id / 1: 5 / HMAC256 /
/ 11/       } >>,
/ 12/       / unprotected: / {},
/ 13/       / payload: / null,
/ 14/       / tag: / h'E6837A54A9B5813F8D5EDAD48AB96D5D
/ 15/                                     7388D9D1C89AB29EC55AE964F67E01ED'
/ 16/     ]) >>
/ 17/   ] >>,
/ 18/   / manifest / 3: << {
/ 19/     / manifest-version / 1: 1,
/ 20/     / manifest-sequence-number / 2: 1,
/ 21/     / common / 3: << {
/ 22/       / components / 2: [
/ 23/         [h'00'],
/ 24/         [h'01']
/ 25/       ]
/ 26/     } >>,
```

```
/ 27/      / install / 20: << [  
/ 28/      /   fetch encrypted firmware /  
/ 29/      /   directive-set-component-index / 12, 1 / [h'01'] /,  
/ 30/      /   directive-override-parameters / 20, {  
/ 31/      /     parameter-image-size / 14: 46,  
/ 32/      /     parameter-uri / 21:  
/ 33/      /       "coaps://example.com/encrypted-firmware"  
/ 34/      /   },  
/ 35/      /   directive-fetch / 21, 15,  
/ 36/      /  
/ 37/      /   decrypt encrypted firmware /  
/ 38/      /   directive-set-component-index / 12, 0 / ['00'] /,  
/ 39/      /   directive-override-parameters / 20, {  
/ 40/      /     parameter-encryption-info / 19: << 96([  
/ 41/      /       / protected: / << {  
/ 42/      /         / alg / 1: 1 / A128GCM /  
/ 43/      /       } >>,  
/ 44/      /       / unprotected: / {  
/ 45/      /         / IV / 5: h'F14AAB9D81D51F7AD943FE87'  
/ 46/      /       },  
/ 47/      /       / ciphertext: / null / detached ciphertext /,  
/ 48/      /       / recipients: / [  
/ 49/      /         [  
/ 50/      /           / protected: / h'',  
/ 51/      /           / unprotected: / {  
/ 52/      /             / alg / 1: -3 / A128KW /,  
/ 53/      /             / kid / 4: 'kid-1'  
/ 54/      /           },  
/ 55/      /           / ciphertext: /  
/ 56/      /             h'75603FFC9518D794713C8CA8  
/ 57/      /             A115A7FB32565A6D59534D62'  
/ 58/      /             / CEK encrypted with KEK /  
/ 59/      /         ]  
/ 60/      /       ]  
/ 61/      /     ] >>,  
/ 62/      /     / parameter-source-component / 22: 1 / [h'01'] /  
/ 63/      /   },  
/ 64/      /   / directive-copy / 22, 15  
/ 65/      /   / consumes the SUIT_Encryption_Info above /  
/ 66/      / ] >>  
/ 67/      / } >>  
/ 68/      / })
```

9.3. ES-DH Example with Write + Copy Directives

The following SUIE manifest instructs a parser to authenticate the manifest using COSE_Sign1 with ES256. It also directs the parser to write and decrypt the encrypted payload into a component via the `suit-directive-write` directive.

The SUIE manifest in diagnostic notation (formatted with line breaks for clarity) is presented below:

```

/ 1/ / SUIE_Envelope_Tagged / 107({
/ 2/   / authentication-wrapper / 2: << [
/ 3/     << [
/ 4/       / digest-algorithm-id: / -16 / SHA256 /,
/ 5/       / digest-bytes: / h'1DB69EF1477E9942815F29F78E09957B
/ 6/                                     26B4ADD03902BDB3D1EDF3DA2075F593'
/ 7/     ] >>,
/ 8/     << / COSE_Sign1_Tagged / 18([
/ 9/       / protected: / << {
/ 10/        / algorithm-id / 1: -9 / ESP256 /
/ 11/       } >>,
/ 12/       / unprotected: / {},
/ 13/       / payload: / null,
/ 14/       / signature: / h'2B20A797AC7DBEBC53147592BB110AEC
/ 15/                                     43A2489AC19A169BB59FF6BD429300A9
/ 16/                                     719FEB7DF277E4B8D1D821C816854229
/ 17/                                     F266AC62AFD9DB52114F608EE66B187B'
/ 18/     ]) >>
/ 19/   ] >>,
/ 20/ / manifest / 3: << {
/ 21/   / manifest-version / 1: 1,
/ 22/   / manifest-sequence-number / 2: 1,
/ 23/   / common / 3: << {
/ 24/     / components / 2: [
/ 25/       ['decrypted-firmware']
/ 26/     ]
/ 27/   } >>,
/ 28/   / install / 20: << [
/ 29/     / directive-set-component-index / 12,
/ 30/     0 / ['plaintext-firmware'] /,
/ 31/     / directive-override-parameters / 20, {
/ 32/       / parameter-content / 18:
/ 33/         h'758C4B7BBAE2C4C1D462423E0F0DC3164FFA7B85BB94D4
/ 34/         BD6D7ED26AB32FEB063385D4D3465927EC82CB5E198A59',
/ 35/       / parameter-encryption-info / 19: << 96([
/ 36/         / protected: / << {
/ 37/           / alg / 1: 1 / A128GCM /
/ 38/         } >>,

```

```

/ 39/          / unprotected: / {
/ 40/          /   IV / 5: h'F14AAB9D81D51F7AD943FE87'
/ 41/          },
/ 42/          / ciphertext: / null / detached ciphertext /,
/ 43/          / recipients: / [
/ 44/          [
/ 45/            / protected: / << {
/ 46/              / alg / 1: -29 / ECDH-ES + A128KW /
/ 47/            } >>,
/ 48/            / unprotected: / {
/ 49/              / ephemeral key / -1: {
/ 50/                / kty / 1: 2 / EC2 /,
/ 51/                / crv / -1: 1 / P-256 /,
/ 52/                / x / -2:
/ 53/                  h'73024F415AA51529A66CCEFD88F3F62A
/ 54/                    734492FF45F6AD37FD2888E73EAF19DA',
/ 55/                / y / -3:
/ 56/                  h'4005B48A6FD091AA6ABFE3CFBEEDE88B
/ 57/                    347E521D43405FDBD7D2CFF0EBC21B26'
/ 58/              },
/ 59/              / kid / 4: 'kid-2'
/ 60/            },
/ 61/            / ciphertext: /
/ 62/              h'A06B8E6550F308712B1DF044
/ 63/                B21B7D11D9B22792F1DE0997'
/ 64/              / CEK encrypted with KEK /
/ 65/            ]
/ 66/          ]
/ 67/        ]) >>
/ 68/      },
/ 69/      / directive-write / 18,
/ 70/      15 / consumes the SUIT_Encryption_Info above /
/ 71/    ] >>
/ 72/  } >>
/ 73/ })

```

In hex format, the SUIT manifest is this:

```
D86BA2025873825824822F58201DB69EF1477E9942815F29F78E09957B26
B4ADD03902BDB3D1EDF3DA2075F593584AD28443A10128A0F658402B20A7
97AC7DBEBC53147592BB110AEC43A2489AC19A169BB59FF6BD429300A971
9FEB7DF277E4B8D1D821C816854229F266AC62AFD9DB52114F608EE66B18
7B0358E8A4010102010357A1028181526465637279707465642D6669726D
776172651458C7860C0014A212582E758C4B7BBAE2C4C1D462423E0F0DC3
164FFA7B85BB94D4BD6D7ED26AB32FEB063385D4D3465927EC82CB5E198A
5913588CD8608443A10101A1054CF14AAB9D81D51F7AD943FE87F6818344
A101381CA220A40102200121582073024F415AA51529A66CCEFD88F3F62A
734492FF45F6AD37FD2888E73EAF19DA2258204005B48A6FD091AA6ABFE3
CFBEDE88B347E521D43405FDBD7D2CFF0EBC21B2604456B69642D325818
A06B8E6550F308712B1DF044B21B7D11D9B22792F1DE0997120F
```

9.4. ES-DH Example with Dependency

The following SUIIT manifest requests a parser to resolve the dependency.

The dependent manifest is signed with another key:

```
-----BEGIN EC PRIVATE KEY-----
MHcCAQEIIQa67e56m8CYL5zVaJFiLl30j0qxb8ray2DeUMqH+qYoAoGCCqGSM49
AwEHoUQDQgAEDpCKqPBm2x8ITgw2UsY5Ur2Z8qW9si+eATZ6rQOrpot32hvYrE8M
tJC6IQZiV3mrFk1JrTVR1x0xSydJ7kLSmg==
-----END EC PRIVATE KEY-----
```

The dependency manifest is embedded as an integrated-dependency and referred to by the "#dependency-manifest" URI.

The SUIIT manifest in diagnostic notation (with line breaks added for readability) is shown here:

```
/ 1/ / SUIIT_Envelope_Tagged / 107({
/ 2/ / authentication-wrapper / 2: << [
/ 3/ << [
/ 4/ / digest-algorithm-id: / -16 / SHA256 /,
/ 5/ / digest-bytes: / h'A00CB6C85515C1EF471B50B542FACDD8
/ 6/ / 8B71B3C7EA2A43DE13D32C4A99056FE9'
/ 7/ ] >>,
/ 8/ << / COSE_Sign1_Tagged / 18([
/ 9/ / protected: / << {
/ 10/ / algorithm-id / 1: -9 / ESP256 /
/ 11/ } >>,
/ 12/ / unprotected: / {},
/ 13/ / payload: / null,
/ 14/ / signature: / h'3000301B7C54B3383CC4723C4B7BE667
/ 15/ / C6760C504213A105DD38401BED5EEF8E
/ 16/ / B915F8313420104F59467D76790A0EA2
```

```
/ 17/                                     20B6021B4ED87051B3B4A8D05F7E0254'
/ 18/      ]) >>
/ 19/      ] >>,
/ 20/      / manifest / 3: << {
/ 21/        / manifest-version / 1: 1,
/ 22/        / manifest-sequence-number / 2: 1,
/ 23/        / common / 3: << {
/ 24/          / dependencies / 1: {
/ 25/            / component-index / 1: {
/ 26/              / dependency-prefix / 1: [
/ 27/                'dependency-manifest.suit'
/ 28/              ]
/ 29/            }
/ 30/          },
/ 31/        / components / 2: [
/ 32/          ['decrypted-firmware']
/ 33/        ]
/ 34/      } >>,
/ 35/      / manifest-component-id / 5: [
/ 36/        'dependent-manifest.suit'
/ 37/      ],
/ 38/      / install / 20: << [
/ 39/        / NOTE: set SUIT_Encryption_Info /
/ 40/        / directive-set-component-index / 12,
/ 41/        0 / ['decrypted-firmware'] /,
/ 42/        / directive-override-parameters / 20, {
/ 43/          / parameter-content / 18:
/ 44/            h'758C4B7BBAE2C4C1D462423E0F0DC3164FFA7B85BB94D4
/ 45/              BD6D7ED26AB32FEB063385D4D3465927EC82CB5E198A59',
/ 46/          / parameter-encryption-info / 19: << 96([
/ 47/            / protected: / << {
/ 48/              / alg / 1: 1 / A128GCM /
/ 49/            } >>,
/ 50/          / unprotected: / {
/ 51/            / IV / 5: h'F14AAB9D81D51F7AD943FE87'
/ 52/          },
/ 53/          / ciphertext: / null / detached ciphertext /,
/ 54/          / recipients: / [
/ 55/            [
/ 56/              / protected: / << {
/ 57/                / alg / 1: -29 / ECDH-ES + A128KW /
/ 58/              } >>,
/ 59/            / unprotected: / {
/ 60/              / ephemeral key / -1: {
/ 61/                / kty / 1: 2 / EC2 /,
/ 62/                / crv / -1: 1 / P-256 /,
/ 63/                / x / -2:
/ 64/                  h'73024F415AA51529A66CCEFD88F3F62A
```

```

/ 65/           734492FF45F6AD37FD2888E73EAF19DA',
/ 66/           / y / -3:
/ 67/           h'4005B48A6FD091AA6ABFE3CFBEEDE88B
/ 68/           347E521D43405FDBD7D2CFF0EBC21B26'
/ 69/           },
/ 70/           / kid / 4: 'kid-2'
/ 71/           },
/ 72/           / ciphertext: /
/ 73/           h'A06B8E6550F308712B1DF044
/ 74/           B21B7D11D9B22792F1DE0997'
/ 75/           / CEK encrypted with KEK /
/ 76/           ]
/ 77/           ]
/ 78/           ]) >>
/ 79/           },
/ 80/
/ 81/           / NOTE: call dependency-manifest /
/ 82/           / directive-set-component-index / 12,
/ 83/           1 / ['dependency-manifest.suit'] /,
/ 84/           / directive-override-parameters / 20, {
/ 85/           / parameter-image-digest / 3: << [
/ 86/           / algorithm-id / -16 / SHA256 /,
/ 87/           / digest-bytes / h'4B15C90FBD776A820E7E733DF040D90B
/ 88/           356B5C75982ECAECE8673818179BDF16'
/ 89/           ] >>,
/ 90/           / parameter-image-size / 14: 247,
/ 91/           / parameter-uri / 21: "#dependency-manifest"
/ 92/           },
/ 93/           / directive-fetch / 21, 15,
/ 94/           / condition-dependency-integrity / 7, 15,
/ 95/           / directive-process-dependency / 11, 15
/ 96/           ] >>
/ 97/           } >>,
/ 98/           "#dependency-manifest": <<
/ 99/           / SUIT_Envelope_Tagged / 107({
/100/           / authentication-wrapper / 2: << [
/101/           << [
/102/           / digest-algorithm-id: / -16 / SHA256 /,
/103/           / digest-bytes: /
/104/           h'4B15C90FBD776A820E7E733DF040D90B
/105/           356B5C75982ECAECE8673818179BDF16'
/106/           ] >>,
/107/           << / COSE_Sign1_Tagged / 18([
/108/           / protected: / << {
/109/           / algorithm-id / 1: -9 / ESP256 /
/110/           } >>,
/111/           / unprotected: / {},
/112/           / payload: / null,

```

```

/113/          / signature: / h'BF95C29295B45470EF819E7F4E3C9084
/114/                               F4534E26469C0A0F2B8B9664881A5359
/115/                               D500F81BD3A6436A025C3E92E51CD714
/116/                               8F83DF47D29FF253F8D41B4D350A2B75'
/117/          ]) >>
/118/          ] >>,
/119/          / manifest / 3: << {
/120/            / manifest-version / 1: 1,
/121/            / manifest-sequence-number / 2: 1,
/122/            / common / 3: << {
/123/              / components / 2: [
/124/                ['decrypted-firmware']
/125/              ],
/126/            / shared-sequence / 4: << [
/127/              / directive-set-componnt-index / 12,
/128/                0 / ['decrypted-firmware'] /,
/129/              / directive-override-parameters / 20, {
/130/                / parameter-image-digest / 3: << [
/131/                  / algorithm-id / -16 / SHA256 /,
/132/                  / digest-bytes /
/133/                    h'36921488FE6680712F734E11F58D87EE
/134/                    B66D4B21A8A1AD3441060814DA16D50F'
/135/                ] >>,
/136/              / parameter-image-size / 14: 30
/137/            }
/138/          ] >>
/139/          } >>,
/140/          / manifest-component-id / 5: [
/141/            'dependency-manifest.suit'
/142/          ],
/143/          / validate / 7: << [
/144/            / condition-image-match / 3, 15
/145/          ] >>,
/146/          / install / 20: << [
/147/            / directive-set-component-index / 12,
/148/              0 / ['decrypted-firmware'] /,
/149/            / directive-write / 18, 15
/150/            / consumes the SUIT_Encryption_Info /
/151/            / set by the dependent /,
/152/            / condition-image-match / 3, 15
/153/            / check the integrity of the decrypted payload /
/154/          ] >>
/155/        } >>
/156/      })
/157/    >>
/158/  })

```

In hex format, the SUIT manifest is this:

```
D86BA3025873825824822F5820A00CB6C85515C1EF471B50B542FACDD88B
71B3C7EA2A43DE13D32C4A99056FE9584AD28443A10128A0F65840300030
1B7C54B3383CC4723C4B7BE667C6760C504213A105DD38401BED5EEF8EB9
15F8313420104F59467D76790A0EA220B6021B4ED87051B3B4A8D05F7E02
540359016CA501010201035837A201A101A101815818646570656E64656E
63792D6D616E69666573742E73756974028181526465637279707465642D
6669726D77617265058157646570656E64656E742D6D616E69666573742E
737569741459010F8E0C0014A212582E758C4B7BBAE2C4C1D462423E0F0D
C3164FFA7B85BB94D4BD6D7ED26AB32FEB063385D4D3465927EC82CB5E19
8A5913588CD8608443A10101A1054CF14AAB9D81D51F7AD943FE87F68183
44A101381CA220A40102200121582073024F415AA51529A66CCEFD88F3F6
2A734492FF45F6AD37FD2888E73EAF19DA2258204005B48A6FD091AA6ABF
E3CFBEEDE88B347E521D43405FDBD7D2CFF0EBC21B2604456B69642D3258
18A06B8E6550F308712B1DF044B21B7D11D9B22792F1DE09970C0114A303
5824822F58204B15C90FBD776A820E7E733DF040D90B356B5C75982ECAEC
E8673818179BDF160E18F7157423646570656E64656E63792D6D616E6966
657374150F070F0B0F7423646570656E64656E63792D6D616E6966657374
58F7D86BA2025873825824822F58204B15C90FBD776A820E7E733DF040D9
0B356B5C75982ECAECE8673818179BDF16584AD28443A10128A0F65840BF
95C29295B45470EF819E7F4E3C9084F4534E26469C0A0F2B8B9664881A53
59D500F81BD3A6436A025C3E92E51CD7148F83DF47D29FF253F8D41B4D35
0A2B7503587BA601010201035849A2028181526465637279707465642D66
69726D7761726504582F840C0014A2035824822F582036921488FE668071
2F734E11F58D87EEB66D4B21A8A1AD3441060814DA16D50F0E181E058158
18646570656E64656E63792D6D616E69666573742E73756974074382030F
1447860C00120F030F
```

10. Operational Considerations

The algorithms outlined in this document assume that the party responsible for payload encryption:

- * shares a key-encryption key (KEK) with the recipient (for use with the AES Key Wrap scheme), or
- * possesses the recipient's public key (for use with ES-DH).

Both scenarios necessitate initial communication to distribute these keys among the involved parties. This interaction can be facilitated by a device management protocol, as described in [RFC9019], or may occur earlier in the device lifecycle, such as during manufacturing or commissioning. In addition to the keying material, key identifiers and algorithm information must also be provisioned. This specification does not impose any requirements on the structure of the key identifier.

In certain situations, third-party companies analyze binaries for known security vulnerabilities. However, encrypted payloads hinder this type of analysis. Consequently, these third-party companies must either be granted access to the plaintext binary before encryption or be authorized recipients of the encrypted payloads.

11. Security Considerations

This entire document focuses on security.

It is considered best security practice to use different keys for different purposes. For instance, the key-encryption key (KEK) utilized in an AES-KW-based content key distribution method for encryption should be distinct from the long-term symmetric key employed for authentication in a communication security protocol.

The recipient MUST verify that the algorithm identifiers carried in both the outer header of the COSE_Encrypt structure and within each recipient structure correspond to the algorithm configuration expected at the recipient, as defined by local policy. If any mismatch is detected between the algorithm identifiers and the configured algorithms, the recipient MUST reject the message. This prevents an attacker from modifying the algorithm field to perform a downgrade attack (for example, changing an AEAD cipher to a non-AEAD cipher) or to cause the recipient to use a key for an unintended purpose.

To further minimize the attack surface, it may be advantageous to use different long-term keys for encrypting various types of payloads. For example, KEK_1 could be used with an AES-KW content key distribution method to encrypt a firmware image, while KEK_2 would encrypt configuration data.

A substantial part of this document focuses on content key distribution, utilizing two primary methods: AES Key Wrap (AES-KW) and Ephemeral-Static Diffie-Hellman (ES-DH). The key properties associated with their deployment are summarized in Table 3.

Number of Long-Term Keys	Number of Content Encryption Keys (CEKs)	Use Case	Recommended?
Same key for all devices	Single CEK per payload shared with all devices	Legacy Usage	No, bad practice
One key per device	Single CEK per payload shared with all devices	Efficient Payload Distribution	Yes
One Key per device	One CEK per payload encryption transaction per device	Point-to-Point Payload Distribution	Yes

Table 3: Content Key Distribution: Comparison

The use of firmware encryption in battery-powered IoT devices introduces the risk of a battery exhaustion attack. This attack exploits the high energy cost of flash memory operations. To execute this attack, the adversary must be able to swap detached payloads and trick the device into processing an incorrect payload. Payload swapping is feasible only if there is no communication security protocol between the device and the distribution system or if the distribution system itself has been compromised.

While the security features provided by the manifest can detect this attack and prevent the device from booting with an incorrectly supplied payload, the energy-intensive flash operations will have already occurred. As a result, these operations can diminish the lifespan of the devices, making battery-powered IoT devices particularly susceptible to such attacks. For further discussion on IoT devices using flash memory, see Section 8.

Including the digest of the encrypted firmware in the manifest enables the device to detect a battery exhaustion attack before energy-consuming decryption and flash memory copy or swap operations take place.

As specified in Section 8 of [RFC9459], recipients must perform integrity checks before decryption to mitigate padding oracle vulnerabilities, particularly when using the AES-CTR mode. This practice not only prevents padding oracle attacks but also protects

against format and decryption oracles, as decryption is skipped if the integrity check fails. For further details on payload integrity validation, see Section 7.2.

The same combination of IV and AES key MUST NOT be reused. This requirement applies not only to AES-CTR mode, as specified in Section 4 of [RFC9459], but also to other content encryption algorithms, including AEAD ciphers like AES-GCM.

Although the examples in this document use the coaps scheme for payload retrieval, alternative URI schemes like coap and http can also be used. This flexibility is possible because the SUIT manifest and this extension do not rely on the TLS layer for security.

Confidentiality, integrity, and authentication are ensured by the SUIT manifest and the extensions defined in this document. For details on how the SUIT manifest meets the security requirements outlined in [RFC9124], refer to Section 12 of [I-D.ietf-suit-manifest]. Additional security considerations for the cryptographic primitives used in these extensions are discussed in Section 11 of [RFC9053] and Section 8 of [RFC9459].

12. IANA Considerations

IANA is asked to add the following value to the SUIT Parameters registry at [iana-suit], which is established by Section 11.5 of [I-D.ietf-suit-manifest]:

Label	Name	Reference
TBD19	Encryption Info	Section 4

RFC Editor's Note (TBD19): The value for the Encryption Info parameter is set to 19, as the proposed value.

13. References

13.1. Normative References

[I-D.ietf-suit-manifest]
Moran, B., Tschofenig, H., Birkholz, H., Zandberg, K., and O. R 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>>.

[I-D.ietf-suit-mti]

Moran, B., R nningstad, O., and A. Tsukamoto,
"Cryptographic Algorithms for Internet of Things (IoT)
Devices", Work in Progress, Internet-Draft, draft-ietf-
suit-mti-23, 22 July 2025,
<[https://datatracker.ietf.org/doc/html/draft-ietf-suit-
mti-23](https://datatracker.ietf.org/doc/html/draft-ietf-suit-mti-23)>.

[I-D.ietf-suit-trust-domains]

Moran, B. and K. Takayama, "Software Update for the
Internet of Things (SUIT) Manifest Extensions for Multiple
Trust Domain", Work in Progress, Internet-Draft, draft-
ietf-suit-trust-domains-12, 22 July 2025,
<[https://datatracker.ietf.org/doc/html/draft-ietf-suit-
trust-domains-12](https://datatracker.ietf.org/doc/html/draft-ietf-suit-trust-domains-12)>.

[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>>.

[RFC3394] Schaad, J. and R. Housley, "Advanced Encryption Standard
(AES) Key Wrap Algorithm", RFC 3394, DOI 10.17487/RFC3394,
September 2002, <<https://www.rfc-editor.org/rfc/rfc3394>>.

[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>>.

[RFC9052] Schaad, J., "CBOR Object Signing and Encryption (COSE):
Structures and Process", STD 96, RFC 9052,
DOI 10.17487/RFC9052, August 2022,
<<https://www.rfc-editor.org/rfc/rfc9052>>.

[RFC9053] Schaad, J., "CBOR Object Signing and Encryption (COSE):
Initial Algorithms", RFC 9053, DOI 10.17487/RFC9053,
August 2022, <<https://www.rfc-editor.org/rfc/rfc9053>>.

[RFC9459] Housley, R. and H. Tschofenig, "CBOR Object Signing and
Encryption (COSE): AES-CTR and AES-CBC", RFC 9459,
DOI 10.17487/RFC9459, September 2023,
<<https://www.rfc-editor.org/rfc/rfc9459>>.

13.2. Informative References

[I-D.ietf-teep-usecase-for-cc-in-network]

Chen, M., Yang, P., Su, L., and T. Pang, "TEEP Usecase for
Confidential Computing in Network", Work in Progress,

Internet-Draft, draft-ietf-teep-usecase-for-cc-in-network-11, 30 June 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-teep-usecase-for-cc-in-network-11>>.

[iana-suit]

Internet Assigned Numbers Authority, "IANA SUIF Manifest Registry", 2023, <TBD>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/rfc/rfc5280>>.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <<https://www.rfc-editor.org/rfc/rfc5652>>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <<https://www.rfc-editor.org/rfc/rfc5869>>.

[RFC8937] Cremers, C., Garratt, L., Smyshlyaev, S., Sullivan, N., and C. Wood, "Randomness Improvements for Security Protocols", RFC 8937, DOI 10.17487/RFC8937, October 2020, <<https://www.rfc-editor.org/rfc/rfc8937>>.

[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>>.

[ROP] Wikipedia, "Return-Oriented Programming", March 2023, <https://en.wikipedia.org/wiki/Return-oriented_programming>.

[SP800-56] NIST, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography, NIST Special Publication 800-56A Revision 3", April 2018, <<http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar3.pdf>>.

Appendix A. Full CDDL

The following CDDL must be appended to the SUIE Manifest CDDL. The SUIE CDDL is defined in Appendix A of [I-D.ietf-suit-manifest]

```
SUIT_Encryption_Info = #6.96(COSE_Encrypt)

$$SUIT_Parameters // = (suit-parameter-encryption-info =>
    bstr .cbor SUIT_Encryption_Info)

suit-parameter-encryption-info = 19
```

Acknowledgements

We would like to thank Henk Birkholz for his feedback on the CDDL description in this document. Additionally, we would like to thank Michael Richardson, Dick Brooks, Å\230yvind RÅ, nningstad, Dave Thaler, Laurence Lundblade, Christian AmsÅ¼ss, Ruud Derwig, Martin Thomson, Kris Kwiatkowski, Suresh Krishnan and Carsten Bormann for their review feedback.

We would like to thank the IESG, in particular Deb Cooley, Å\211ric Vyncke and Roman Danyliw, for their help to improve the quality of this document.

Authors' Addresses

Hannes Tschofenig
University of Applied Sciences Bonn-Rhein-Sieg
Email: Hannes.Tschofenig@gmx.net

Russ Housley
Vigil Security, LLC
Email: housley@vigilsec.com

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

David Brown
Linaro
Email: david.brown@linaro.org

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

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 29 November 2025

B. Moran
Arm Limited
H. Tschofenig
H-BRS
H. Birkholz
Fraunhofer SIT
K. Zandberg
Inria
Å\230. RÅ, nningstad
Nordic Semiconductor
28 May 2025

A Concise Binary Object Representation (CBOR)-based Serialization Format
for the Software Updates for Internet of Things (SUIT) Manifest
draft-ietf-suit-manifest-34

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 29 November 2025.

Copyright Notice

Copyright (c) 2025 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1.	Introduction	4
2.	Conventions and Terminology	6
3.	How to use this Document	8
4.	Background	9
4.1.	IoT Firmware Update Constraints	9
4.2.	SUIT Workflow Model	10
5.	Metadata Structure Overview	11
5.1.	Envelope	12
5.2.	Authentication Block	13
5.3.	Manifest	13
5.3.1.	Critical Metadata	13
5.3.2.	Common	13
5.3.3.	Command Sequences	14
5.3.4.	Integrity Check Values	14
5.3.5.	Human-Readable Text	15
5.4.	Severable Elements	15
5.5.	Integrated Payloads	15
6.	Manifest Processor Behavior	16
6.1.	Manifest Processor Setup	16
6.2.	Required Checks	17
6.3.	Interpreter Fundamental Properties	18
6.3.1.	Resilience to Disruption	18
6.4.	Abstract Machine Description	19
6.5.	Special Cases of Component Index	21
6.6.	Serialized Processing Interpreter	22
6.7.	Parallel Processing Interpreter	23
7.	Creating Manifests	24
7.1.	Compatibility Check Template	25
7.2.	Trusted Invocation Template	25
7.3.	Component Download Template	26
7.4.	Install Template	26
7.5.	Integrated Payload Template	27

7.6.	Load from Nonvolatile Storage Template	27
7.7.	A/B Image Template	28
8.	Metadata Structure	29
8.1.	Encoding Considerations	30
8.2.	Envelope	30
8.3.	Authenticated Manifests	30
8.4.	Manifest	31
8.4.1.	suit-manifest-version	32
8.4.2.	suit-manifest-sequence-number	32
8.4.3.	suit-reference-uri	32
8.4.4.	suit-text	32
8.4.5.	suit-common	34
8.4.6.	SUIT_Command_Sequence	35
8.4.7.	Reporting Policy	37
8.4.8.	SUIT_Parameters	38
8.4.9.	SUIT_Condition	46
8.4.10.	SUIT_Directive	49
8.4.11.	suit-command-custom	54
8.4.12.	Integrity Check Values	55
8.5.	Implementation Conformance Matrix	55
8.6.	Severable Elements	57
9.	Access Control Lists	58
10.	SUIT Digest Container	59
11.	IANA Considerations	59
11.1.	SUIT Envelope Elements	60
11.2.	SUIT Manifest Elements	61
11.3.	SUIT Common Elements	62
11.4.	SUIT Commands	63
11.5.	SUIT Parameters	65
11.6.	SUIT Text Values	66
11.7.	SUIT Component Text Values	66
11.8.	Expert Review Instructions	67
11.9.	Media Type Registration	68
12.	Security Considerations	70
13.	Acknowledgements	72
14.	References	73
14.1.	Normative References	73
14.2.	Informative References	75
Appendix A.	Full CDDL	75
Appendix B.	Examples	82
B.1.	Example 0: Secure Boot	83
B.2.	Example 1: Simultaneous Download and Installation of Payload	85
B.3.	Example 2: Simultaneous Download, Installation, Secure Boot, Severed Fields	87
B.4.	Example 3: A/B images	90
B.5.	Example 4: Load from External Storage	93
B.6.	Example 5: Two Images	96

Appendix C. Design Rationale	99
C.1. C.1 Design Rationale: Envelope	100
C.2. C.2 Byte String Wrappers	101
Authors' Addresses	102

1. Introduction

A firmware update mechanism is an essential security feature for IoT devices to deal with vulnerabilities. The transport of firmware images to the devices themselves is important security aspect. Luckily, there are already various device management solutions available offering the distribution of firmware images to IoT devices. Equally important is the inclusion of metadata about the conveyed firmware image (in the form of a manifest) and the use of a security wrapper to provide end-to-end security protection to detect modifications and (optionally) to make reverse engineering more difficult. Firmware signing allows the author, who builds the firmware image, to be sure that no other party (including potential adversaries) can install firmware updates on IoT devices without adequate privileges. For confidentiality protected firmware images it is additionally required to encrypt the firmware image and to distribute the content encryption key securely. The support for firmware and payload encryption via the SUIT manifest format is described in a companion document

[I-D.ietf-suit-firmware-encryption]. Starting security protection at the author is a risk mitigation technique so firmware images and manifests can be stored on untrusted repositories; it also reduces the scope of a compromise of any repository or intermediate system to be no worse than a denial of service.

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

This specification defines the SUIT manifest format. It is intended to meet several goals:

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

- * Extensible.

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

- * a Network Operator to reason about compatibility of a firmware, such as timing and acceptance of firmware updates.
- * a Device Operator to reason about the impact of a firmware.
- * a device to evaluate the authenticity of a firmware and the authority of the firmware author prior to installation.
- * a device to evaluate the applicability of a firmware.
- * a device to determine the installation process of a firmware.
- * a device to evaluate the authenticity of a firmware pre-boot
- * a device to determine the encoding and boot process of a firmware.

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

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

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

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

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

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

While the SUIT manifest is informed by and optimized for firmware update and Trusted Invocation use cases, there is nothing in the SUIT Information Model [RFC9124] that restricts its use to only those use cases. Other use cases include the management of trusted applications (TAs) in a Trusted Execution Environment (TEE), as discussed in [RFC9397].

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. Severable elements can be removed from the manifest without impacting its security, see Section 8.6.
- * **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.
- * **Pull parser:** A parser that traverses the data and extracts information on an as-needed basis.
- * **Severable element:** An element of the manifest that supports elision of hashed data. If a hash of the data is included in the manifest and the data is included in the envelope, then that data may be elided.

3. How to use this Document

This specification covers five aspects of firmware update:

- * Section 4 describes the device constraints, use cases, and design principles that informed the structure of the manifest.
- * Section 5 gives a general overview of the metadata structure to inform the following sections
- * Section 6 describes what actions a Manifest processor should take.
- * Section 7 describes the process of creating a Manifest.
- * Section 8 specifies the content of the Envelope and the Manifest.

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

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

The complete CDDL ([RFC8610]) definition is provided in Appendix A, examples are given in Appendix B and a design rationale is offered in Appendix C. Finally, Section 8.5 summarizes the mandatory-to-implement features of this specification.

Additional specifications describe functionality needed to implement all of the requirements of [RFC9124], such as:

- * Firmware encryption [I-D.ietf-suit-firmware-encryption]
- * Update management [I-D.ietf-suit-update-management]
- * Dependency manifests [I-D.ietf-suit-trust-domains]
- * Secure reporting of the update status [I-D.ietf-suit-report]

A technique to compress firmware images may be standardized in the future.

4. Background

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

4.1. IoT Firmware Update Constraints

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

- * limited processing power and storage may require a simple representation of metadata.
- * bandwidth constraints may require firmware compression or partial update support.
- * intermittent or unstable connectivity.
- * intermittent power, for example due to energy harvesting.

- * bootloader complexity constraints may require simple selection between two bootable images.
- * small internal storage may require external storage support.
- * multiple microcontrollers may require coordinated update of all applications.
- * large storage and complex functionality may require parallel update of many software components.
- * extra information may need to be conveyed in the manifest in the earlier stages of the device lifecycle before those data items are stripped when the manifest is delivered to a constrained device.

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

4.2. SUIT Workflow Model

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

- * Compatibility must be checked before any other operation is performed.
- * In some applications, payloads must be fetched and validated prior to installation.

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

- * Compatibility must be checked before any other operation is performed.
- * All payloads must be validated prior to loading.
- * All loaded images must be validated prior to execution.

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

1. Verify the signature of the manifest.

2. Verify the applicability of the manifest.
3. Fetch payload(s).
4. Install payload(s).
5. Verify image(s).

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

1. Verify image(s).
2. Load image(s).
3. Invoke image(s).

If verification and invocation is implemented in a bootloader, then the bootloader **MUST** also verify the signature of the manifest and the applicability of the manifest in order to implement secure boot workflows. Because signature verifications can be costly in constrained applications, the bootloader may add its own authentication, e.g., a Message Authentication Code (MAC), to the manifest in order to prevent further signature verifications and save energy, provided that the bootloader can protect its authentication key.

5. Metadata Structure Overview

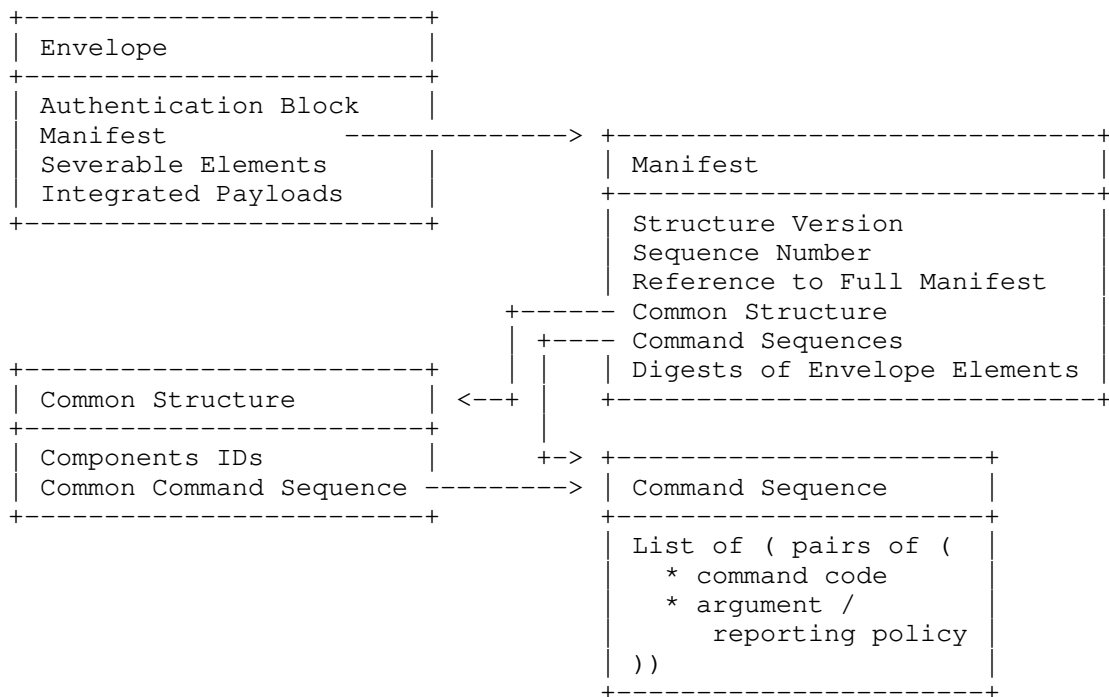
This section provides a high level overview of the manifest structure. The full description of the manifest structure is in Section 8.4

The manifest is structured from several key components:

1. The Envelope (see Section 5.1) contains the Authentication Block, the Manifest, any Severable Elements, and any Integrated Payloads.
2. The Authentication Block (see Section 5.2) contains a list of signatures or MACs of the manifest.
3. The Manifest (see Section 5.3) contains all critical, non-severable metadata that the Recipient requires. It is further broken down into:
 1. Critical metadata, such as sequence number.

2. Common metadata, such as affected components.
3. Command sequences, directing the Recipient how to install and use the payload(s).
4. Integrity check values for severable elements.
4. Severable elements (see Section 5.4).
5. Integrated payloads (see Section 5.5).

The diagram below illustrates the hierarchy of the Envelope.



5.1. Envelope

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

See Section 8.2 for more detail.

5.2. Authentication Block

The Authentication Block contains a bstr-wrapped SUIT Digest Container, see Section 10, and one or more [RFC9052] CBOR Object Signing and Encryption (COSE) authentication blocks. These blocks are one of:

- * COSE_Sign_Tagged
- * COSE_Sign1_Tagged
- * COSE_Mac_Tagged
- * COSE_Mac0_Tagged

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

See Section 8.3 for more detail.

5.3. Manifest

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

See Section 8.4 for more detail.

5.3.1. Critical Metadata

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

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

5.3.2. Common

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

have any side-effects outside of setting parameter values.

See Section 8.4.5 for more detail.

5.3.3. Command Sequences

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

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

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

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

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

See Section 8.4.6 for more detail.

5.3.4. Integrity Check Values

To enable severable elements Section 5.4, there needs to be a mechanism to verify the integrity of the severed data. While the severed data stays outside the manifest, for efficiency reasons, Integrity Check Values are used to include the digest of the data in the manifest. Note that Integrated Payloads, see Section 5.5, are integrity-checked using Command Sequences.

See Section 8.4.12 for more detail.

5.3.5. Human-Readable Text

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

See Section 8.4.4 for more detail.

5.4. Severable Elements

Severable Elements are elements of the Envelope (Section 5.1) that have Integrity Check Values (Section 5.3.4) in the Manifest (Section 5.3). This is a form of elision of hashed data. The elements in the envelope are verified by Integrity Check Values and therefore cannot be replaced with other elements even if they are authenticated elements.

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

- * A management system severs the Text sections before sending an Envelope to a constrained Recipient, which saves Recipient bandwidth.
- * A Recipient severs the Installation section after installing the Update, which saves storage space.

See Section 8.6 for more detail.

5.5. Integrated Payloads

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

- * When an update is delivered via a comparatively unconstrained medium, such as a removable mass storage device, it may be beneficial to bundle updates into single files.
- * When a manifest transports a small payload, such as an encrypted key, that payload may be placed in the manifest's envelope.

See Section 7.5 for more detail.

6. Manifest Processor Behavior

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

6.1. Manifest Processor Setup

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

- * Whenever the manifest processor can choose between several manifests, it MUST select the latest valid, authentic manifest.
- * If the latest valid, authentic manifest fails, it MAY select the next latest valid, authentic manifest, according to application-specific policy.

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

- * Test failed (e.g., Vendor ID/Class ID).
- * Unsupported command encountered.
- * Unsupported parameter encountered.
- * Unsupported Component Identifier encountered.
- * Payload not available.
- * Application crashed when executed.
- * Watchdog timeout occurred.
- * Payload verification failed.
- * Missing required component from a Component Set.

* Required parameter not supplied.

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

Selecting an older manifest in the event of failure of the latest valid manifest is one possible strategy to provide robustness of the firmware update process. It may not be appropriate for all applications. In particular Trusted Execution Environments MAY require a failure to invoke a new installation, rather than a rollback approach. See [RFC9124], Section 4.2.1 for more discussion on the security considerations that apply to rollback.

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

6.2. Required Checks

The manifest processor MUST verify the signature of the manifest prior to parsing/executing any section of the manifest. This guards the parser against arbitrary input by unauthenticated third parties. When validating authenticity of manifests, the manifest processor MAY use an ACL (see Section 9) to determine the extent of the rights conferred by that authenticity.

Once a valid, authentic manifest has been selected, the manifest processor MUST examine the component list and check that the number of components listed in the manifest is not larger than the number in the target system.

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

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

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

If the manifest contains more than one component, each command sequence MUST begin with a Set Component Index Section 8.4.10.1.

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:

1. the manifest Author has sufficient permissions for the requested operations (see Section 9) and
2. the manifest specifies a digest and a payload for every Component in the Component Set.

6.3. Interpreter Fundamental Properties

The interpreter has a small set of design goals:

1. Executing an update MUST either result in an error, or a correct system state that can be checked against known digests.
2. Executing a Trusted Invocation MUST either result in an error, or an invoked image.
3. Executing the same manifest on multiple Recipients MUST result in the same system state.

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

6.3.1. Resilience to Disruption

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

The manifest processor must be resilient to these faults. In order to enable this resilience, systems implementing the manifest processor MUST guarantee that manifests can be either resumed or reapplied.

This can be achieved in a variety of ways: 1. A fallback/recovery image is provided so that a disrupted system can apply the SUIT Manifest again. 2. Manifest Authors construct Manifests in such a way that repeated partial invocations of any Manifest always results in a correct system state. Typically this is done by using Try-Each and Conditions to bypass operations that have already been completed. 3. A journal of manifest operations is stored in nonvolatile memory. The journal enables the parser to re-create the state just prior to the disruption. This journal can, for example, be a SUIT Report or a journaling file system. 4. Where a command is not repeatable because

of the way in which it alters system state (e.g., swapping images or in-place delta) it is resumable or revertible. This applies primarily to commands that modify at least one source component as well as the destination component.

6.4. Abstract Machine Description

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

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

Conditions MUST NOT have any side-effects other than informing the interpreter of success or failure. The Interpreter does not Abort if the Soft Failure flag (Section 8.4.8.15) is set when a Condition reports failure.

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

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

```
current := components[component-index]
```

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

The following table describes the semantics of each operation. The pseudo-code semantics are inspired by the Python programming language.

pseudo-code operation	Semantics
assert(test)	When test is false, causes an error return
store(dest, source)	Writes source into dest
expression0 for-each e in l else expression1	Performs expression0 once for each element in iterable l; performs expression1 if no break is encountered
break	halt a for-each loop
now()	return the current UTC time
expression if test	performs expression if test is true

Table 1

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

Command Name	Semantic of the Operation
Check Vendor Identifier	assert(binary-match(current, current.params[vendor-id]))
Check Class Identifier	assert(binary-match(current, current.params[class-id]))
Verify Image	assert(binary-match(digest(current), current.params[digest]))
Check Content	assert(binary-match(current, current.params[content]))
Set Component Index	current := components[arg]
Override Parameters	current.params[k] := v for-each k,v in arg
Invoke	invoke(current)
Fetch	store(current,

	<code>fetch(current.params[uri])</code>
Write	<code>store(current, current.params[content])</code>
Use Before	<code>assert(now() < arg)</code>
Check Component Slot	<code>assert(current.slot-index == arg)</code>
Check Device Identifier	<code>assert(binary-match(current, current.params[device-id]))</code>
Abort	<code>assert(0)</code>
Try Each	<code>(break if (exec(seq) is not error)) for-each seq in arg else assert(0)</code>
Copy	<code>store(current, current.params[src-component])</code>
Swap	<code>swap(current, current.params[src-component])</code>
Run Sequence	<code>exec(arg)</code>
Invoke with Arguments	<code>invoke(current, arg)</code>

Table 2

6.5. Special Cases of Component Index

Component Index can take on one of three types:

1. Integer
2. Array of integers
3. True

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

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

When a command is executed, it

1. operates on the component identified by the component index if that index is an integer, or
2. it operates on each component identified by an array of indices, or
3. it operates on every component if the index is the boolean True.

This is described by the following pseudocode:

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

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

6.6. Serialized Processing Interpreter

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

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

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

6.7. Parallel Processing Interpreter

To enable parallel or out-of-order processing of Command Sequences, Recipients MAY make use of the Strict Order parameter. The Strict Order parameter indicates to the Manifest Processor that Commands MUST be executed strictly in order. When the Strict Order parameter is False, this indicates to the Manifest Processor that Commands MAY be executed in parallel and/or out of order.

To perform parallel processing, once the Strict Order parameter is set to False, the Recipient MAY add each command to an issue queue for parallel processing or an issue pool for out-of-order processing. The Manifest Processor then executes these pending commands in whatever order or parallelism it deems appropriate. Once there are no more commands to add to the issue queue/pool, the Manifest Processor drains the issue queue/pool by issuing all pending commands and waits for every issued command to complete. The Manifest Processor MAY issue commands before it has completed adding all remaining commands to the issue queue/pool.

While adding commands to the issue queue or pool, if the Manifest Processor encounters any of the following commands, it MUST treat the command as a barrier, draining the issue queue/pool and waiting for all issued commands to complete.

- * Override Parameters.
- * Set Strict Order = True.
- * Set Component Index.

Extensions MAY alter this list. Once all issued commands have completed, the Manifest Processor issues the barrier command, after which it may resume parallel processing if Strict Order is still False.

A Component MUST NOT be both a target of an operation and a source of data (for example, in Copy or Swap) in a Command Sequence where Strict Order is False. This would cause a race condition if the Component is written to, then later read from. The Manifest Processor MUST issue an Abort if it detects this exception.

To perform more useful parallel operations, a manifest author may collect sequences of commands in a Run Sequence command. Then, each of these sequences MAY be run in parallel. There are several invocation options for Run Sequence:

- * Component Index is a positive integer, Strict Order is False: Strict Order is set to True before the sequence argument is run. The sequence argument MUST begin with set-component-index.
- * Component Index is true or an array of positive integers, Strict Order is False: The sequence argument is run once for each component (or each component in the array); the Manifest Processor presets the component index and Strict Order = True before each iteration of the sequence argument.
- * Component Index is a positive integer, Strict Order is True: No special considerations
- * Component Index is True or an array of positive integers, Strict Order is True: The sequence argument is run once for each component (or each component in the array); the Manifest Processor presets the component index before each iteration of the sequence argument.

These rules isolate each sequence from each other sequence, ensuring that they operate as expected. When Strict Order = False, any further Set Component Index directives in the Run Sequence command sequence argument MUST cause an Abort. This allows the interpreter that issues Run Sequence commands to check that the first element is correct, then issue the sequence to a parallel execution context to handle the remainder of the sequence.

7. Creating Manifests

Manifests are created using tools for constructing COSE structures, calculating cryptographic values and compiling desired system state into a sequence of operations required to achieve that state. The process of constructing COSE structures and the calculation of cryptographic values is covered in [RFC9052].

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

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

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

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

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

7.1. Compatibility Check Template

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

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

- * Set Component Index directive (see Section 8.4.10.1)
- * Override Parameters directive (see Section 8.4.10.3) for Vendor ID and Class ID (see Section 8.4.8)
- * Check Vendor Identifier condition (see Section 8.4.8.2)
- * Check Class Identifier condition (see Section 8.4.8.2)

7.2. Trusted Invocation Template

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

The following commands are placed into the shared sequence:

- * Set Component Index directive (see Section 8.4.10.1)
- * Override Parameters directive (see Section 8.4.10.3) for Image Digest and Image Size (see Section 8.4.8)

The system validation sequence contains the following commands:

- * Set Component Index directive (see Section 8.4.10.1)
- * Check Image Match condition (see Section 8.4.9.2)

Then, the run sequence contains the following commands:

- * Set Component Index directive (see Section 8.4.10.1)
- * Invoke directive (see Section 8.4.10.7)

7.3. Component Download Template

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

The following commands are placed into the shared sequence:

- * Set Component Index directive (see Section 8.4.10.1)
- * Override Parameters directive (see Section 8.4.10.3) for Image Digest and Image Size (see Section 8.4.8)

Then, the install sequence contains the following commands:

- * Set Component Index directive (see Section 8.4.10.1)
- * Override Parameters directive (see Section 8.4.10.3) for URI (see Section 8.4.8.10)
- * Fetch directive (see Section 8.4.10.4)
- * Check Image Match condition (see Section 8.4.9.2)

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

7.4. Install Template

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

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

Then, the install sequence contains the following commands:

- * Set Component Index directive (see Section 8.4.10.1)
- * Override Parameters directive (see Section 8.4.10.3) for Source Component (see Section 8.4.8.11)
- * Copy directive (see Section 8.4.10.5)
- * Check Image Match condition (see Section 8.4.9.2)

7.5. Integrated Payload Template

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

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

The URI for a payload enclosed in this way MAY be expressed as a fragment-only reference, as defined in [RFC3986], Section 4.4, for example: "#device-model-v1.2.3.bin".

An intermediary, such as a Network Operator, MAY choose to pre-fetch a payload and add it to the manifest envelope, using the URI as the key.

7.6. Load from Nonvolatile Storage Template

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

The following commands are placed into the load sequence:

- * Set Component Index directive (see Section 8.4.10.1)
- * Override Parameters directive (see Section 8.4.10.3) for Source Component (see Section 8.4.8)
- * Copy directive (see Section 8.4.10.5)

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

7.7. A/B Image Template

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

The following commands are placed in the common block:

- * Set Component Index directive (see Section 8.4.10.1)
- * Try Each
 - First Sequence:
 - o Override Parameters directive (see Section 8.4.10.3, Section 8.4.8) for Slot A
 - o Check Slot Condition (see Section 8.4.9.4)
 - o Override Parameters directive (see Section 8.4.10.3) for Image Digest A and Image Size A (see Section 8.4.8)
 - Second Sequence:
 - o Override Parameters directive (see Section 8.4.10.3, Section 8.4.8) for Slot B
 - o Check Slot Condition (see Section 8.4.9.4)
 - o Override Parameters directive (see Section 8.4.10.3) for Image Digest B and Image Size B (see Section 8.4.8)

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

- * Set Component Index directive (see Section 8.4.10.1)
- * Try Each
 - First Sequence:
 - o Override Parameters directive (see Section 8.4.10.3, Section 8.4.8) for Slot A
 - o Check Slot Condition (see Section 8.4.9.4)

- o Set Parameters directive (see Section 8.4.10.3) for URI A (see Section 8.4.8)
- Second Sequence:
 - o Override Parameters directive (see Section 8.4.10.3, Section 8.4.8) for Slot B
 - o Check Slot Condition (see Section 8.4.9.4)
 - o Set Parameters directive (see Section 8.4.10.3) for URI B (see Section 8.4.8)
- * Fetch

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

- * Set Component Index directive (see Section 8.4.10.1)
- * Try Each
 - First Sequence:
 - o Override Parameters directive (see Section 8.4.10.3, Section 8.4.8) for Slot A
 - o Check Slot Condition (see Section 8.4.9.4)
 - Second Sequence:
 - o Override Parameters directive (see Section 8.4.10.3, Section 8.4.8) for Slot B
 - o Check Slot Condition (see Section 8.4.9.4)
- * Invoke

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

8. Metadata Structure

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

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

8.1. Encoding Considerations

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

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

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

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

8.2. Envelope

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

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

8.3. Authenticated Manifests

SUIT_Authentication contains a list of elements, which consist of a SUIT_Digest calculated over the manifest, and zero or more SUIT_Authentication_Block's calculated over the SUIT_Digest.

```
SUIT_Authentication = [  
    bstr .cbor SUIT_Digest,  
    * bstr .cbor SUIT_Authentication_Block  
]  
SUIT_Authentication_Block /= COSE_Mac_Tagged  
SUIT_Authentication_Block /= COSE_Sign_Tagged  
SUIT_Authentication_Block /= COSE_Mac0_Tagged  
SUIT_Authentication_Block /= COSE_Sign1_Tagged
```

The `SUIT_Digest` is computed over the bstr-wrapped `SUIT_Manifest` that is present in the `SUIT_Envelope` at the `suit-manifest` key. The `SUIT_Digest` MUST always be present. The Manifest Processor requires a `SUIT_Authentication_Block` to be present. The manifest MUST be protected from tampering between the time of creation and the time of signing/MACing.

The `SUIT_Authentication_Block` is computed using detached payloads, as described in RFC 9052 [RFC9052]. The detached payload in each case is the bstr-wrapped `SUIT_Digest` at the beginning of the list. Signers (or MAC calculators) MUST verify the `SUIT_Digest` prior to performing the cryptographic computation to avoid "Time-of-check to time-of-use" type of attack. When multiple `SUIT_Authentication_Blocks` are present, then each `SUIT_Authentication_Block` MUST be computed over the same `SUIT_Digest` but using a different algorithm or signing/MAC authority. This feature also allows to transition to new algorithms, such as post-quantum cryptography (PQC) algorithms.

The `SUIT_Authentication` structure MUST come before the `suit-manifest` element, regardless of canonical encoding of CBOR. The algorithms used in `SUIT_Authentication` are defined by the profiles declared in [I-D.ietf-suit-mti].

8.4. Manifest

The manifest contains:

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

The Text section, or any Command Sequence of the Update Procedure (Image Fetch, Image Installation and, System Validation) can be either a CBOR structure or a `SUIT_Digest`. In each of these cases,

the SUIE_Digest provides for a severable element. Severable elements are RECOMMENDED to implement. In particular, the human-readable text SHOULD be severable, since most useful text elements occupy more space than a SUIE_Digest, but are not needed by the Recipient. Because SUIE_Digest is a CBOR Array and each severable element is a CBOR bstr, it is straight-forward for a Recipient to determine whether an element has been severed. The key used for a severable element is the same in the SUIE_Manifest and in the SUIE_Envelope so that a Recipient can easily identify the correct data in the envelope. See Section 8.4.12 for more detail.

8.4.1. suit-manifest-version

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

8.4.2. suit-manifest-sequence-number

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

8.4.3. suit-reference-uri

suit-reference-uri is a URI where a full version of this manifest can be found. This is convenient for allowing management systems to show the severed elements of a manifest when this URI is reported by a Recipient after installation. This document is only concerned with the transport of a URI which is intended for machine readable uses, not human readable uses. The encoding is the same as CBOR Tag 32, however the tag is omitted because it is implied by context.

8.4.4. suit-text

suit-text SHOULD be a severable element. suit-text is a map of language identifiers (identical to Tag38 of RFC9290, Appendix A) to language-specific text maps. Each language-specific text map is a map containing two different types of pair:

- * integer => text
- * SUIE_Component_Identifier => map

The SUIE_Text_Map is defined in the following CDDL.

```

tag38-ltag = text .regexp "[a-zA-Z]{1,8}(-[a-zA-Z0-9]{1,8})*"

SUIT_Text_Map = {
  + tag38-ltag => SUIT_Text_LMap
}
SUIT_Text_LMap = {
  SUIT_Text_Keys,
  * SUIT_Component_Identifier => {
    SUIT_Text_Component_Keys
  }
}

```

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

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

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

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

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

Table 3

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

CDDL Structure	Description
<code>suit-text-vendor-name</code>	Free text vendor name
<code>suit-text-model-name</code>	Free text model name
<code>suit-text-vendor-domain</code>	The domain used to create the vendor-id condition (see Section 8.4.8.2)
<code>suit-text-model-info</code>	The information used to create the class-id condition (see Section 8.4.8.2)
<code>suit-text-component-description</code>	Free text description of each component in the manifest
<code>suit-text-component-version</code>	A free text representation of the component version

Table 4

`suit-text` is OPTIONAL to implement.

8.4.5. `suit-common`

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

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

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

8.4.5.1. SUIF_Component_Identifier

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

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

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

8.4.6. SUIF_Command_Sequence

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

1. Payload Fetch: suit-payload-fetch is a SUIF_Command_Sequence to execute in order to obtain a payload. Some manifests may include these actions in the suit-install section instead if they operate in a streaming installation mode. This is particularly relevant for constrained devices without any temporary storage for staging the update. suit-payload-fetch is OPTIONAL to implement because it is not relevant in all bootloaders.
2. Payload Installation: suit-install is a SUIF_Command_Sequence to execute in order to install a payload. Typical actions include verifying a payload stored in temporary storage, copying a staged payload from temporary storage, and unpacking a payload. suit-install is OPTIONAL to implement.

3. Image Validation: `suit-validate` is a `SUIT_Command_Sequence` to execute in order to validate that the result of applying the update is correct. Typical actions involve image validation. `suit-validate` is REQUIRED to implement.
4. Image Loading: `suit-load` is a `SUIT_Command_Sequence` to execute in order to prepare a payload for execution. Typical actions include copying an image from permanent storage into RAM, optionally including actions such as decryption or decompression. `suit-load` is OPTIONAL to implement.
5. Invoke or Boot: `suit-invoke` is a `SUIT_Command_Sequence` to execute in order to invoke an image. `suit-invoke` typically contains a single instruction: the "invoke" directive, but may also contain an image condition. `suit-invoke` is OPTIONAL to implement because it not needed for restart-based invocation.

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

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

Lists of commands are constructed from two kinds of element:

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

Each condition is composed of:

1. A command code identifier
2. A `SUIT_Reporting_Policy` (Section 8.4.7)

Each directive is composed of:

1. A command code identifier
2. An argument block or a `SUIT_Reporting_Policy` (Section 8.4.7)

Argument blocks are consumed only by flow-control directives:

- * Set Component Index
- * Set/Override Parameters

- * Try Each
- * Run Sequence

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

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

To facilitate optional conditions, a special directive, `suit-directive-try-each` (Section 8.4.10.2), is provided. It runs several new lists of conditions/directives, one after another, that are contained as an argument to the directive. By default, it assumes that a failure of a condition should not indicate a failure of the update/invocation, but a parameter is provided to override this behavior. See `suit-parameter-soft-failure` (Section 8.4.8.15).

8.4.7. Reporting Policy

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

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

Table 5

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

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

The information elements provided to the Reporting Engine are:

- * The reporting policy
- * The result of the command
- * The values of parameters consumed by the command
- * The system information consumed by the command

The Reporting Engine consumes these information elements and decides whether to generate an entry in its report output and which information elements to include based on its internal policy decisions. The Reporting Engine uses the reporting policy provided to it by the SUIT Manifest Processor as a set of hints but MAY choose to ignore these hints and apply its own policy instead.

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

This specification does not define a particular format of Records or Reports. This specification only defines hints to the Reporting Engine for which information elements it should aggregate into the Report.

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

8.4.8. SUIT_Parameters

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

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

The defined manifest parameters are described below.

Name	CDDL Structure	Reference
Vendor ID	suit-parameter-vendor-identifier	Section 8.4.8.3
Class ID	suit-parameter-class-identifier	Section 8.4.8.4
Device ID	suit-parameter-device-identifier	Section 8.4.8.5
Image Digest	suit-parameter-image-digest	Section 8.4.8.6
Image Size	suit-parameter-image-size	Section 8.4.8.7
Content	suit-parameter-content	Section 8.4.8.9
Component Slot	suit-parameter-component-slot	Section 8.4.8.8
URI	suit-parameter-uri	Section 8.4.8.10
Source Component	suit-parameter-source-component	Section 8.4.8.11
Invoke Args	suit-parameter-invoke-args	Section 8.4.8.12
Fetch Arguments	suit-parameter-fetch-arguments	Section 8.4.8.13
Strict Order	suit-parameter-strict-order	Section 8.4.8.14
Soft Failure	suit-parameter-soft-failure	Section 8.4.8.15
Custom	suit-parameter-custom	Section 8.4.8.16

Table 6

CBOR-encoded object parameters are still wrapped in a bstr. This is because it allows a parser that is aggregating parameters to reference the object with a single pointer and traverse it without understanding the contents. This is important for modularization and

division of responsibility within a pull parser. The same consideration does not apply to Directives because those elements are invoked with their arguments immediately.

8.4.8.1. CBOR PEN UUID Namespace Identifier

The CBOR PEN (Private Enterprise Number) UUID Namespace Identifier is constructed as follows:

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

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

Computing a version 5 UUID from these produces:

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

8.4.8.2. Constructing UUIDs

Several conditions use identifiers to determine whether a manifest matches a given Recipient or not. These identifiers are defined to be RFC 9562 [RFC9562] UUIDs. These UUIDs are not human-readable and are therefore used for machine-based processing only.

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

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

A more complete example: Imagine a device has the following physical components: 1. A host Microcontroller 2. A Wi-Fi module

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

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

1. Hardware model/revision
2. OS
3. Wi-Fi module model/revision
4. Application

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

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

UUIDs MUST be created according to versions 3, 4, or 5 of [RFC9562]. Versions 1 and 2 do not provide a tangible benefit over version 4 for this application.

The RECOMMENDED method to create a vendor ID is:

The "IANA UUID Namespace ID for DNS" is:
6ba7b810-9dad-11d1-80b4-00c04fd430c8

Vendor ID = UUID5(<IANA UUID Namespace ID DNS>, vendor domain name)

In this case, the vendor domain name is a UTF-8 encoded string. Since UUID version 5 applies a digest, internationalization considerations are not applied. The native UTF-8 domain name is used.

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

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

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

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

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

- * Model number.
- * Hardware revision.
- * Bootloader version (for immutable bootloaders).

8.4.8.3. `suit-parameter-vendor-identifier`

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

- * A Private Enterprise Number
- * A byte string containing a UUID [RFC9562]

Private Enterprise Numbers are encoded as a relative OID, according to the definition in [RFC9090]. All PENs are relative to the IANA PEN: 1.3.6.1.4.1.

8.4.8.4. `suit-parameter-class-identifier`

A RFC 9562 UUID representing the class of the device or component. The UUID is encoded as a 16 byte `bstr`, containing the raw bytes of the UUID. It MUST be constructed as described in Section 8.4.8.2

8.4.8.5. `suit-parameter-device-identifier`

A RFC 9562 UUID representing the specific device or component. The UUID is encoded as a 16 byte `bstr`, containing the raw bytes of the UUID. It MUST be constructed as described in Section 8.4.8.2

8.4.8.6. `suit-parameter-image-digest`

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

8.4.8.7. `suit-parameter-image-size`

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

8.4.8.8. suit-parameter-component-slot

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

8.4.8.9. suit-parameter-content

A block of raw data for use with Section 8.4.10.6. It contains a byte string of data to be written to a specified component ID in the same way as a fetch or a copy.

If data is encoded this way, it should be small, e.g., 10's of bytes. Large payloads, e.g., 1000's of bytes, written via this method might prevent the manifest from being held in memory during validation. Typical applications include small configuration parameters.

The size of payload embedded in suit-parameter-content impacts the security requirement defined in [RFC9124], Section 4.3.21 REQ.SEC.MFST.CONST: Manifest Kept Immutable between Check and Use. Actual limitations on payload size for suit-parameter-content depend on the application, in particular the available memory that satisfies REQ.SEC.MFST.CONST. If the availability of tamper resistant memory is less than the manifest size, then REQ.SEC.MFST.CONST cannot be satisfied.

If suit-parameter-content is instantiated in a severable command sequence, then this becomes functionally very similar to an integrated payload, which may be a better choice.

8.4.8.10. suit-parameter-uri

A URI Reference [RFC3986] from which to fetch a resource. The encoding is the same as CBOR Tag 32, however the tag is omitted because it is implied by the context. This document is only concerned with the transport of a URI which is intended for machine readable uses, not human readable uses.

8.4.8.11. suit-parameter-source-component

This parameter sets the source component to be used with either suit-directive-copy (Section 8.4.10.5) or with suit-directive-swap (Section 8.4.10.9). The current Component, as set by suit-directive-set-component-index defines the destination, and suit-parameter-source-component defines the source.

8.4.8.12. `suit-parameter-invoke-args`

This parameter contains an encoded set of arguments for `suit-directive-invoke` (Section 8.4.10.7). The arguments MUST be provided as an implementation-defined bstr.

8.4.8.13. `suit-parameter-fetch-arguments`

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

8.4.8.14. `suit-parameter-strict-order`

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

See Section 6.7 for behavioral description of Strict Order.

8.4.8.15. `suit-parameter-soft-failure`

When executing a command sequence inside `suit-directive-try-each` (Section 8.4.10.2) or `suit-directive-run-sequence` (Section 8.4.10.8) and a condition failure occurs, the manifest processor aborts the sequence. For `suit-directive-try-each`, if Soft Failure is True, the next sequence in Try Each is invoked, otherwise `suit-directive-try-each` fails with the condition failure code. In `suit-directive-run-sequence`, if Soft Failure is True the `suit-directive-run-sequence` simply halts with no side-effects and the Manifest Processor continues with the following command, otherwise, the `suit-directive-run-sequence` fails with the condition failure code.

suit-parameter-soft-failure is scoped to the enclosing SUIT_Command_Sequence. Its value is discarded when the enclosing SUIT_Command_Sequence terminates and suit-parameter-soft-failure reverts to the value it had prior to the invocation of the SUIT_Command_Sequence. Nested SUIT_Command_Sequences do not inherit the enclosing sequence's suit-parameter-soft-failure. It MUST NOT be set outside of suit-directive-try-each or suit-directive-run-sequence, modifying suit-parameter-soft-failure outside of these circumstances causes an Abort.

When suit-directive-try-each is invoked, Soft Failure defaults to True in every SUIT_Command_Sequence in the suit-directive-try-each argument. An Update Author may choose to set Soft Failure to False if they require a failed condition in a sequence to force an Abort. When the enclosing SUIT_Command_Sequence terminates, suit-parameter-soft-failure reverts to the value it held before the SUIT_Command_Sequence was invoked.

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

8.4.8.16. suit-parameter-custom

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

suit-parameter-custom MAY be consumed by any command, in an application-specific way, however if a suit-parameter-custom is absent, then all standardised suit-commands MUST execute correctly. In this respect, suit-parameter-custom MUST be treated as a hint by any standardised suit-command that consumes it.

8.4.9. SUIT_Condition

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

Name	CDDL Structure	Reference
Vendor Identifier	suit-condition-vendor-identifier	Section 8.4.9.1
Class Identifier	suit-condition-class-identifier	Section 8.4.9.1
Device Identifier	suit-condition-device-identifier	Section 8.4.9.1
Image Match	suit-condition-image-match	Section 8.4.9.2
Check Content	suit-condition-check-content	Section 8.4.9.3
Component Slot	suit-condition-component-slot	Section 8.4.9.4
Abort	suit-condition-abort	Section 8.4.9.5
Custom Condition	suit-command-custom	Section 8.4.11

Table 7

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

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

Each condition MUST report a result code on completion. If a condition reports failure, then the current sequence of commands MUST terminate. A subsequent command or command sequence MAY continue

executing if `suit-parameter-soft-failure` (Section 8.4.8.15) is set. If a condition requires additional information, this MUST be specified in one or more parameters before the condition is executed. If a Recipient attempts to process a condition that expects additional information and that information has not been set, it MUST report a failure. If a Recipient encounters an unknown condition, it MUST report a failure.

Condition labels greater than or equal to -256 are reserved for IANA registration while those lesser than -256 are custom conditions reserved for proprietary definition by the author of a manifest processor. See Section 11 for more details.

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

There are three identifier-based conditions: `suit-condition-vendor-identifier`, `suit-condition-class-identifier`, and `suit-condition-device-identifier`. Each of these conditions match a UUID [RFC9562] that MUST have already been set as a parameter. The installing Recipient MUST match the specified UUID in order to consider the manifest valid. These identifiers are scoped by component in the manifest. Each component MAY match more than one identifier. Care is needed to ensure that manifests correctly identify their targets using these conditions. Using only a generic class ID for a device-specific firmware could result in matching devices that are not compatible.

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

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

8.4.9.2. `suit-condition-image-match`

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

8.4.9.3. suit-condition-check-content

This directive compares the specified component identifier to the data indicated by suit-parameter-content. This functions similarly to suit-condition-image-match, however it does a direct, byte-by-byte comparison rather than a digest-based comparison. Because it is possible that an early stop to check-content could reveal information through timing, suit-condition-check-content MUST be constant time: no early exits.

The following pseudo-code described an example content checking algorithm:

```
// content & component must be same length
// returns 0 for match
int check_content(content, component, length) {
    int residual = 0;
    for (i = 0; i < length; i++) {
        residual |= content[i] ^ component[i];
    }
    return residual;
}
```

8.4.9.4. suit-condition-component-slot

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

8.4.9.5. suit-condition-abort

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

8.4.10. SUIT_Directive

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

Name	CDDL Structure	Reference
Set Component Index	suit-directive-set-component-index	Section 8.4.10.1
Try Each	suit-directive-try-each	Section 8.4.10.2
Override Parameters	suit-directive-override-parameters	Section 8.4.10.3
Fetch	suit-directive-fetch	Section 8.4.10.4
Copy	suit-directive-copy	Section 8.4.10.5
Write	suit-directive-write	Section 8.4.10.6
Invoke	suit-directive-invoke	Section 8.4.10.7
Run Sequence	suit-directive-run-sequence	Section 8.4.10.8
Swap	suit-directive-swap	Section 8.4.10.9
Custom Directive	suit-command-custom	Section 8.4.11

Table 8

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

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

8.4.10.1. suit-directive-set-component-index

Set Component Index defines the component to which successive directives and conditions will apply. The Set Component Index arguments are described in Section 6.5.

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.

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

8.4.10.2. `suit-directive-try-each`

This command runs several `SUIT_Command_Sequence` instances, one after another, in a strict order, until one succeeds or the list is exhausted. Use this command to implement a "try/catch-try/catch" sequence. Manifest processors MAY implement this command.

`suit-parameter-soft-failure` (Section 8.4.8.15) is initialized to True at the beginning of each sequence. If one sequence aborts due to a condition failure, the next is started. If no sequence completes without condition failure, then `suit-directive-try-each` returns an error. If a particular application calls for all sequences to fail and still continue, then an empty sequence (nil) can be added to the Try Each Argument.

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

8.4.10.3. `suit-directive-override-parameters`

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

Available parameters are defined in Section 8.4.8.

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

8.4.10.4. `suit-directive-fetch`

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

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

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

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

The size and digest of the payload to be fetched are typically set prior to the invocation of `suit-directive-fetch`. If both `suit-parameter-image-digest` and `suit-parameter-image-size` are set for the current component when `suit-directive-fetch` is invoked, the Manifest Processor MAY choose to optimize the fetch by:

- * Checking if the target component matches the digest supplied before fetching.
- * Checking if another component matches the digest supplied before fetching.

The exact mechanisms of these optimizations are implementation defined.

8.4.10.5. `suit-directive-copy`

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

`suit-directive-copy` reads its source from `suit-parameter-source-component` (Section 8.4.8.11).

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

The size and digest of the payload to be fetched are typically set prior to the invocation of `suit-directive-copy`. If both `suit-parameter-image-digest` and `suit-parameter-image-size` are set for the current component when `suit-directive-copy` is invoked, the Manifest Processor MAY choose to optimize the copy by:

- * Checking if the target component matches the digest supplied before copying.
- * Checking if the source component matches the digest supplied before copying.

The first optimization avoids a copy operation when the data is the same. The second optimization avoids a copy of a corrupted image. The exact mechanisms of these optimizations are implementation defined.

8.4.10.6. `suit-directive-write`

This directive writes a small block of data, specified in Section 8.4.8.9, to a component.

Encoding Considerations: Careful consideration must be taken to determine whether it is more appropriate to use an integrated payload or to use Section 8.4.8.9 for a particular application. While the encoding of `suit-directive-write` is smaller than an integrated payload, a large `suit-parameter-content` payload may prevent the manifest processor from holding the command sequence in memory while executing it.

8.4.10.7. `suit-directive-invoke`

`suit-directive-invoke` directs the manifest processor to transfer execution to the current Component Index. When this is invoked, the manifest processor MAY be unloaded and execution continues in the Component Index. Arguments are provided to `suit-directive-invoke` through `suit-parameter-invoke-arguments` (Section 8.4.8.12) and are forwarded to the executable code located in Component Index in an application-specific way. For example, this could form the Linux Kernel Command Line if booting a Linux device.

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

8.4.10.8. `suit-directive-run-sequence`

To enable conditional commands, and to allow several strictly ordered sequences to be executed out-of-order, `suit-directive-run-sequence` allows the manifest processor to execute its argument as a `SUIIT_Command_Sequence`. The argument must be wrapped in a `bstr`. This also allows a sequence of instructions to be iterated over, once for each current component index, when `component-index = true` or `component-index = list`. See Section 6.5.

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

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

`suit-parameter-soft-failure` (Section 8.4.8.15) defaults to `False` when `suit-directive-run-sequence` begins. Its value is discarded when `suit-directive-run-sequence` terminates.

8.4.10.9. `suit-directive-swap`

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

8.4.11. `suit-command-custom`

`suit-command-custom` identifies an experimental, proprietary, or application-specific condition or directive. The associated value is an integer less than `22023256`, selected by the firmware developer from the Private Use address range defined for the respective registry. If additional information must be provided, it should be encoded in a custom parameter (as described in Section 8.4.8). Any number of custom commands is permitted. `SUIIT_Command_Custom` is OPTIONAL to implement.

8.4.12. Integrity Check Values

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

Each Integrity Check Value covers the corresponding Envelope Element as described in Section 8.6.

8.5. Implementation Conformance Matrix

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

The subsequent table shows the conditions.

Name	Reference	Implementation
Vendor Identifier	Section 8.4.8.2	REQUIRED
Class Identifier	Section 8.4.8.2	REQUIRED
Device Identifier	Section 8.4.8.2	OPTIONAL
Image Match	Section 8.4.9.2	REQUIRED
Check Content	Section 8.4.9.3	OPTIONAL
Component Slot	Section 8.4.9.4	OPTIONAL
Abort	Section 8.4.9.5	OPTIONAL
Custom Condition	Section 8.4.11	OPTIONAL

Table 9

The subsequent table shows the directives.

Name	Reference	Implementation
Set Component Index	Section 8.4.10.1	REQUIRED if more than one component
Write Content	Section 8.4.10.6	OPTIONAL
Try Each	Section 8.4.10.2	OPTIONAL
Override Parameters	Section 8.4.10.3	REQUIRED
Fetch	Section 8.4.10.4	REQUIRED for Updater
Copy	Section 8.4.10.5	OPTIONAL
Invoke	Section 8.4.10.7	REQUIRED for Bootloader
Run Sequence	Section 8.4.10.8	OPTIONAL
Swap	Section 8.4.10.9	OPTIONAL

Table 10

The subsequent table shows the parameters.

Name	Reference	Implementation
Vendor ID	Section 8.4.8.3	REQUIRED
Class ID	Section 8.4.8.4	REQUIRED
Image Digest	Section 8.4.8.6	REQUIRED
Image Size	Section 8.4.8.7	REQUIRED
Component Slot	Section 8.4.8.8	OPTIONAL
Content	Section 8.4.8.9	OPTIONAL
URI	Section 8.4.8.10	REQUIRED for Updater
Source Component	Section 8.4.8.11	OPTIONAL
Invoke Args	Section 8.4.8.12	OPTIONAL
Device ID	Section 8.4.8.5	OPTIONAL
Strict Order	Section 8.4.8.14	OPTIONAL
Soft Failure	Section 8.4.8.15	OPTIONAL
Custom	Section 8.4.8.16	OPTIONAL

Table 11

8.6. Severable Elements

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

At time of manifest creation, the Author MAY chose to make a manifest element severable by removing it from the manifest, encoding it in a bstr, and placing a SUIE_Digest of the bstr in the manifest so that it can still be authenticated. Making an element severable changes the digest of the manifest, so the signature MUST be computed after manifest elements are made severable. Only Manifest Elements with corresponding elements in the SUIE_Envelope can be made severable (see Section 11.1 for SUIE_Envelope elements). The SUIE_Digest typically consumes 4 bytes more than the size of the raw digest, therefore elements smaller than $(\text{Digest Bits})/8 + 4$ SHOULD NOT be severable. Elements larger than $(\text{Digest Bits})/8 + 4$ MAY be severable, while elements that are much larger than $(\text{Digest Bits})/8 + 4$ SHOULD be severable.

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

9. Access Control Lists

SUIE Manifest Processors are RECOMMENDED to use one of the following models for managing permissions in the manifest.

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

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

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

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

10. SUIT Digest Container

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

The values of the algorithm identifier are found in the IANA "COSE Algorithms" registry [COSE-Alg], which was created by [RFC9054]. SHA-256 (-16) MUST be implemented by all Manifest Processors.

Any other algorithm defined in the IANA "COSE Algorithms" registry, such as SHA-512 (-44), MAY be implemented in a Manifest Processor.

11. IANA Considerations

IANA is requested to register the following CBOR Tags:

- * Tag: 107
- * Data Item: map
- * Semantics: SUIT_Envelope as defined in Appendix A
- * Reference: [this RFC]
- * Tag: 1070
- * Data Item: map
- * Semantics: SUIT_Manifest as defined in Appendix A
- * Reference: [this RFC]

Additionally, IANA is requested to register:

- * allocate CBOR tag 107 (suggested) in the "CBOR Tags" registry for the SUIT Envelope. The CBOR Tag's Data Item is a SUIT_Envelope as defined in Appendix A
- * allocate CBOR tag 1070 (suggested) in the "CBOR Tags" registry for the SUIT Manifest. The CBOR Tag's Data Item is a SUIT_Manifest as defined in Appendix A
- * allocate media type application/suit-envelope+cose in the "Media Types" registry, see below.

- * allocate Namespace CBOR PEN in the "UUID Namespace IDs" registry with value 47fbdabb-f2e4-55f0-bb39-3620c2f6df4e, as defined in Section 8.4.8.1
- * setup several registries as described below.

IANA is requested to create a new category for Software Update for the Internet of Things (SUIT) and a page within this category for SUIT manifests.

IANA is also requested to create several registries defined in the subsections below.

For each registry, the number space is partitioned, with each range governed by a different allocation policy:

- * Values 256 and above are subject to Specification Required,
- * Values in the range 0 to 255 follow a Standards Action policy,
- * Values from 223255 to 0 are also governed by Standards Action, and
- * Values 223256 and below are designated for Private Use (also referred to as custom values).

New entries to those registries need to provide a label, a name and a reference to a specification that describes the functionality. More guidance on the expert review can be found below.

11.1. SUIT Envelope Elements

IANA is requested to create a new registry for SUIT envelope elements.

Label	Name	Reference
-255 to -1	Unassigned	
0	Unset Detection	Section 8.1 of [TBD: this document]
1	Reserved (Delegation)	Appendix C.1 of [TBD: this document]
2	Authentication Wrapper	Section 8.3 of [TBD: this document]
3	Manifest	Section 8.4 of [TBD: this document]
4 to 15	Unassigned	
16	Payload Fetch	Section 8.4.6 of [TBD: this document]
17 to 19	Unassigned	
20	Payload Installation	Section 8.4.6 of [TBD: this document]
21 to 22	Unassigned	
23	Text Description	Section 8.4.4 of [TBD: this document]

Table 12

11.2. SUIT Manifest Elements

IANA is requested to create a new registry for SUIT manifest elements.

Label	Name	Reference
-255 to -1	Unassigned	
0	Unset Detection	Section 8.1 of [TBD: this document]

1	Encoding Version	Section 8.4.1 of [TBD: this document]
2	Sequence Number	Section 8.4.2 of [TBD: this document]
3	Common Data	Section 8.4.5 of [TBD: this document]
4	Reference URI	Section 8.4.3 of [TBD: this document]
5 to 6	Unassigned	
7	Image Validation	Section 8.4.6 of [TBD: this document]
8	Image Loading	Section 8.4.6 of [TBD: this document]
9	Image Invocation	Section 8.4.6 of [TBD: this document]
10 to 15	Unassigned	
16	Payload Fetch	Section 8.4.6 of [TBD: this document]
17 to 19	Unassigned	
20	Payload Installation	Section 8.4.6 of [TBD: this document]
21 to 22	Unassigned	
23	Text Description	Section 8.4.4 of [TBD: this document]

Table 13

11.3. SUIT Common Elements

IANA is requested to create a new registry for SUIT common elements.

Label	Name	Reference
-255 to -1	Unassigned	
0	Unset Detection	Section 8.1 of [TBD: this document]
1	Unassigned	
2	Component Identifiers	Section 8.4.5 of [TBD: this document]
3	Unassigned	
4	Common Command Sequence	Section 8.4.5 of [TBD: this document]

Table 14

11.4. SUIT Commands

IANA is requested to create a new registry for SUIT commands.

Label	Name	Reference
-255 to -1	Unassigned	
0	Unset Detection	Section 8.1 of [TBD: this document]
1	Vendor Identifier	Section 8.4.9.1 of [TBD: this document]
2	Class Identifier	Section 8.4.9.1 of [TBD: this document]
3	Image Match	Section 8.4.9.2 of [TBD: this document]
4	Unassigned	
5	Component Slot	Section 8.4.9.4 of [TBD: this document]
6	Check Content	Section 8.4.9.3 of

		[TBD: this document]
7 to 11	Unassigned	
12	Set Component Index	Section 8.4.10.1 of [TBD: this document]
13	Unassigned	
14	Abort	Section 8.4.9.5 of [TBD: this document]
15	Try Each	Section 8.4.10.2 of [TBD: this document]
16 to 17	Unassigned	
18	Write Content	Section 8.4.10.6 of [TBD: this document]
19	Unassigned	
20	Override Parameters	Section 8.4.10.3 of [TBD: this document]
21	Fetch	Section 8.4.10.4 of [TBD: this document]
22	Copy	Section 8.4.10.5 of [TBD: this document]
23	Invoke	Section 8.4.10.7 of [TBD: this document]
24	Device Identifier	Section 8.4.9.1 of [TBD: this document]
25 to 30	Unassigned	
31	Swap	Section 8.4.10.9 of [TBD: this document]
32	Run Sequence	Section 8.4.10.8 of [TBD: this document]

Table 15

11.5. SUIT Parameters

IANA is requested to create a new registry for SUIT parameters.

Label	Name	Reference
-255 to -1	Unassigned	
0	Unset Detection	Section 8.1 of [TBD: this document]
1	Vendor ID	Section 8.4.8.3 of [TBD: this document]
2	Class ID	Section 8.4.8.4 of [TBD: this document]
3	Image Digest	Section 8.4.8.6 of [TBD: this document]
4	Unassigned	
5	Component Slot	Section 8.4.8.8 of [TBD: this document]
6 to 11	Unassigned	
12	Strict Order	Section 8.4.8.14 of [TBD: this document]
13	Soft Failure	Section 8.4.8.15 of [TBD: this document]
14	Image Size	Section 8.4.8.7 of [TBD: this document]
15 to 17	Unassigned	
18	Content	Section 8.4.8.9 of [TBD: this document]
19 to 20	Unassigned	
21	URI	Section 8.4.8.10 of [TBD: this document]
22	Source	Section 8.4.8.11 of [TBD: this document]

	Component	[TBD: this document]
23	Invoke Args	Section 8.4.8.12 of [TBD: this document]
24	Device ID	Section 8.4.8.5 of [TBD: this document]

Table 16

11.6. SUIT Text Values

IANA is requested to create a new registry for SUIT text values.

Label	Name	Reference
-255 to -1	Unassigned	
0	Unset Detection	Section 8.1 of [TBD: this document]
1	Manifest Description	Section 8.4.4 of [TBD: this document]
2	Update Description	Section 8.4.4 of [TBD: this document]
3	Manifest JSON Source	Section 8.4.4 of [TBD: this document]
4	Manifest YAML Source	Section 8.4.4 of [TBD: this document]

Table 17

11.7. SUIT Component Text Values

IANA is requested to create a new registry for SUIT component text values.

Label	Name	Reference
-255 to -1	Unassigned	
0	Unset Detection	Section 8.1 of [TBD: this document]
1	Vendor Name	Section 8.4.4 of [TBD: this document]
2	Model Name	Section 8.4.4 of [TBD: this document]
3	Vendor Domain	Section 8.4.4 of [TBD: this document]
4	Model Info	Section 8.4.4 of [TBD: this document]
5	Component Description	Section 8.4.4 of [TBD: this document]
6	Component Version	Section 8.4.4 of [TBD: this document]

Table 18

11.8. Expert Review Instructions

The IANA registries established in this document allow values to be added based on expert review. This section gives some general guidelines for what the experts should be looking for, but they are being designated as experts for a reason, so they should be given substantial latitude.

Expert reviewers should take into consideration the following points:

- * Point squatting should be discouraged. Reviewers are encouraged to get sufficient information for registration requests to ensure that the usage is not going to duplicate one that is already registered, and that the point is likely to be used in deployments. The zones tagged as private use are intended for testing purposes and closed environments; code points in other ranges should not be assigned for testing.

- * Specifications are required for the standards track range of point assignment. Specifications should exist for all other ranges, but early assignment before a specification is available is considered to be permissible. When specifications are not provided, the description provided needs to have sufficient information to identify what the point is being used for.
- * Experts should take into account the expected usage of fields when approving point assignment. The fact that there is a range for standards track documents does not mean that a standards track document cannot have points assigned outside of that range. The length of the encoded value should be weighed against how many code points of that length are left, the size of device it will be used on, and the number of code points left that encode to that size.
- * Key assignments in the SUIT Parameters table, particularly those that encode to 1 CBOR byte (-24 to 23) should be reserved for SUIT Directives that match the same key value.

11.9. Media Type Registration

This section registers the 'application/suit-envelope+cose' media type in the "Media Types" registry. This media type are used to indicate that the content is a SUIT envelope.

Type name: application

Subtype name: suit-envelope+cose

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: See the Security Considerations section of [[This RFC]].

Interoperability considerations: N/A

Published specification: [[This RFC]]

Applications that use this media type: Primarily used for Firmware and software updates although the content may also contain configuration data and other information related to software and firmware.

Fragment identifier considerations: N/A

Additional information:

* Deprecated alias names for this type: N/A

* Magic number(s): N/A

* File extension(s): cbor, suit

* Macintosh file type code(s): N/A

Person & email address to contact for further information:
iesg@ietf.org

Intended usage: COMMON

Restrictions on usage: N/A

Author: Brendan Moran, <brendan.moran.ietf@gmail.com>

Change Controller: IETF

Provisional registration? No

12. Security Considerations

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

The security requirements outlined in [RFC9124] are addressed by this draft and its extensions. The specific mapping of requirements and information elements in [RFC9124] to manifest data structures is outlined in the table below:

Security Requirement	Information Element	Implementation
REQ.SEC.SEQUENCE	Monotonic Sequence Number	Section 8.4.2
REQ.SEC.COMPATIBLE	Vendor ID Condition, Class ID Condition	Section 8.4.9.1
REQ.SEC.EXP	Expiration Time	[I-D.ietf-suit-update-management]
REQ.SEC.AUTHENTIC	Signature, Payload Digests	Section 8.3, Section 8.4.9.2
REQ.SEC.AUTH.IMG_TYPE	Payload Format	[I-D.ietf-suit-update-management]
REQ.SEC.AUTH.IMG_LOC	Storage Location	Section 8.4.5.1
REQ.SEC.AUTH.REMOTE_LOC	Payload Indicator	Section 8.4.8.10
REQ.SEC.AUTH.EXEC	Payload Digests, Size	Section 8.4.8.6, Section 8.4.8.7

	REQ.SEC.AUTH.PRECURSOR	Precursor Image	Section 8.4.8.6
		Digest	
-----+			
.8.4	REQ.SEC.AUTH.COMPATIBILITY	Authenticated	Section 8.4.8.3, Section 8.4
		Vendor and Class	
		IDs	
-----+			
	REQ.SEC.RIGHTS	Signature	Section 8.3, Section 9
-----+			
	REQ.SEC.IMG.CONFIDENTIALITY	Encryption Wrapper	[I-D.ietf-suit-firmware-encryption]

REQ.SEC.ACCESS_CONTROL:	None	Section 9
Access Control		
REQ.SEC.MFST.CONFIDENTIALITY	Manifest Encryption	[I-D.ietf-suit-firmware-encryption]
	Wrapper / Transport	
	Security	
REQ.SEC.IMG.COMPLETE_DIGEST	Payload Digests	Implementation Consideration
REQ.SEC.REPORTING	None	[I-D.ietf-suit-report], [RFC 9334]
REQ.SEC.KEY.PROTECTION	None	Implementation Consideration
REQ.SEC.KEY.ROTATION	None	[I-D.tschofenig-cose-cwt-chain], Implementation Consideration
REQ.SEC.MFST.CHECK	None	Deployment Consideration
REQ.SEC.MFST.TRUSTED	None	Deployment Consideration
REQ.SEC.MFST.CONST	None	Implementation Consideration
REQ.USE.MFST.PRE_CHECK	Additional	[I-D.ietf-suit-update-management]
	Installation	
	Instructions	
REQ.USE.MFST.TEXT	Manifest Text	Section 8.4.4
	Information	
REQ.USE.MFST.OVERRIDE_REMOTE	Aliases	[RFC3986] Relative URIs,

]			[I-D.ietf-suit-trust-domains
+-----+			
	REQ.USE.MFST.COMPONENT	Dependencies,	SUIT_Component_Identifier
		StorageIdentifier,	(Section 8.4.5.1),
		ComponentIdentifier	[I-D.ietf-suit-trust-domains
]			
+-----+			
	REQ.USE.MFST.MULTI_AUTH	Signature	Section 8.3
+-----+			
	REQ.USE.IMG.FORMAT	Payload Format	[I-D.ietf-suit-update-manage
ment]			
+-----+			
	REQ.USE.IMG.NESTED	Processing Steps	[I-D.ietf-suit-firmware-encr
yption]			(Encryption Wrapper),
			[I-D.ietf-suit-update-manage
ment]			(Payload Format)
+-----+			
	REQ.USE.IMG.VERSIONS	Required Image	[I-D.ietf-suit-update-manage
ment]		Version List	

REQ.USE.IMG.SELECT	XIP Address	Section 8.4.9.4
REQ.USE.EXEC	Runtime Metadata	Section 8.4.6 (suit-invoke)
REQ.USE.LOAD	Load-Time Metadata	Section 8.4.6 (suit-load)
REQ.USE.PAYLOAD	Payload	Section 7.5
REQ.USE.PARSE	Simple Parsing	Section 6.4
REQ.USE.DELEGATION	Delegation Chain	[I-D.tschofenig-cose-cwt-chain]

Table 19

13. Acknowledgements

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

- * Milosch Meriac
- * Geraint Luff
- * Dan Ros
- * John-Paul Stanford
- * Hugo Vincent
- * Carsten Bormann
- * Frank Audun Kvamtr ,
- * Krzysztof Chru \233ci \204ski
- * Andrzej Puzdrowski
- * Michael Richardson
- * David Brown
- * Emmanuel Baccelli

We would like to thank our responsible area director, Roman Danyliw, for his detailed review. Finally, we would like to thank our SUIT working group chairs (Dave Thaler, David Waltermire, Russ Housley) for their feedback and support.

14. References

14.1. Normative References

[COSE-Alg] IANA, "CBOR Object Signing and Encryption (COSE) Algorithms", <<https://www.iana.org/assignments/cose/cose.xhtml#algorithms>>.

[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-24, 19 March 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-suit-firmware-encryption-24>>.

[I-D.ietf-suit-mti]
Moran, B., Rønningstad, O., and A. Tsukamoto, "Mandatory-to-Implement Algorithms for Authors and Recipients of Software Update for the Internet of Things manifests", Work in Progress, Internet-Draft, draft-ietf-suit-mti-15, 26 May 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-suit-mti-15>>.

[I-D.ietf-suit-report]
Moran, B. and H. Birkholz, "Secure Reporting of Update Status", Work in Progress, Internet-Draft, draft-ietf-suit-report-11, 3 March 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-suit-report-11>>.

[I-D.ietf-suit-trust-domains]
Moran, B. and K. Takayama, "SUIT Manifest Extensions for Multiple Trust Domains", Work in Progress, Internet-Draft, draft-ietf-suit-trust-domains-10, 3 March 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-suit-trust-domains-10>>.

- [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>>.
- [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>>.
- [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/rfc/rfc3986>>.
- [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>>.
- [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>>.
- [RFC9052] Schaad, J., "CBOR Object Signing and Encryption (COSE): Structures and Process", STD 96, RFC 9052, DOI 10.17487/RFC9052, August 2022, <<https://www.rfc-editor.org/rfc/rfc9052>>.
- [RFC9054] Schaad, J., "CBOR Object Signing and Encryption (COSE): Hash Algorithms", RFC 9054, DOI 10.17487/RFC9054, August 2022, <<https://www.rfc-editor.org/rfc/rfc9054>>.

- [RFC9090] Bormann, C., "Concise Binary Object Representation (CBOR) Tags for Object Identifiers", RFC 9090, DOI 10.17487/RFC9090, July 2021, <<https://www.rfc-editor.org/rfc/rfc9090>>.
- [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>>.
- [RFC9562] Davis, K., Peabody, B., and P. Leach, "Universally Unique Identifiers (UUIDs)", RFC 9562, DOI 10.17487/RFC9562, May 2024, <<https://www.rfc-editor.org/rfc/rfc9562>>.

14.2. Informative References

- [I-D.tschofenig-cose-cwt-chain] Tschofenig, H., Moran, B., and H. Birkholz, "CBOR Object Signing and Encryption (COSE): Header Parameters for Carrying and Referencing Chains of CBOR Web Tokens (CWTs)", Work in Progress, Internet-Draft, draft-tschofenig-cose-cwt-chain-02, 2 March 2025, <<https://datatracker.ietf.org/doc/html/draft-tschofenig-cose-cwt-chain-02>>.
- [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/rfc/rfc7228>>.
- [RFC9334] Birkholz, H., Thaler, D., Richardson, M., Smith, N., and W. Pan, "Remote Attestation procedures (RATS) Architecture", RFC 9334, DOI 10.17487/RFC9334, January 2023, <<https://www.rfc-editor.org/rfc/rfc9334>>.
- [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>>.
- [YAML] "YAML Ain't Markup Language (YAML) version 1.2", 2021, <<https://yaml.org/spec/1.2.2/>>.

Appendix A. Full CDDL

In order to create a valid SUIT Manifest document the structure of the corresponding CBOR message MUST adhere to the following CDDL ([RFC8610]) data definition.

To be valid, the following CDDL MUST have the COSE CDDL appended to it. The COSE CDDL can be obtained by following the directions in [RFC9052], Section 1.4.

```

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

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

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

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

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

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

SUIT_Manifest_Tagged = #6.1070(SUIT_Manifest)

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

```

```

    SUIE_Severable_Members_Choice,
    * $$SUIE_Manifest_Extensions,
}

SUIE_Unseverable_Members = (
  ? suit-validate => bstr .cbor SUIE_Command_Sequence,
  ? suit-load => bstr .cbor SUIE_Command_Sequence,
  ? suit-invoke => bstr .cbor SUIE_Command_Sequence,
  * $$unseverable-manifest-member-extensions,
)

SUIE_Severable_Members_Choice = (
  ? suit-payload-fetch => SUIE_Digest /
    bstr .cbor SUIE_Command_Sequence,
  ? suit-install => SUIE_Digest / bstr .cbor SUIE_Command_Sequence,
  ? suit-text => SUIE_Digest / bstr .cbor SUIE_Text_Map,
  * $$severable-manifest-members-choice-extensions
)

SUIE_Common = {
  ? suit-components           => SUIE_Components,
  ? suit-shared-sequence     => bstr .cbor SUIE_Shared_Sequence,
  * $$SUIE_Common-extensions,
}

SUIE_Components              = [ + SUIE_Component_Identifier ]

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

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

SUIE_Component_Identifier = [* bstr]

SUIE_Shared_Sequence = [
  + ( SUIE_Condition // SUIE_Shared_Commands )
]

SUIE_Shared_Commands // = (suit-directive-set-component-index,  IndexArg)
SUIE_Shared_Commands // = (suit-directive-run-sequence,
  bstr .cbor SUIE_Shared_Sequence)
SUIE_Shared_Commands // = (suit-directive-try-each,
  SUIE_Directive_Try_Each_Argument_Shared)
SUIE_Shared_Commands // = (suit-directive-override-parameters,

```

```

    (+ $$SUIE_Parameters})

IndexArg /= uint
IndexArg /= true
IndexArg /= [+uint]

SUIE_Directive_Try_Each_Argument_Shared = [
    2* bstr .cbor SUIE_Shared_Sequence,
    ?nil
]

SUIE_Command_Sequence = [ + (
    SUIE_Condition // SUIE_Directive // SUIE_Command_Custom
) ]

SUIE_Command_Custom = (suit-command-custom, bstr/tstr/int/nil)
SUIE_Condition // = (suit-condition-vendor-identifier, SUIE_Rep_Policy)
SUIE_Condition // = (suit-condition-class-identifier, SUIE_Rep_Policy)
SUIE_Condition // = (suit-condition-device-identifier, SUIE_Rep_Policy)
SUIE_Condition // = (suit-condition-image-match, SUIE_Rep_Policy)
SUIE_Condition // = (suit-condition-component-slot, SUIE_Rep_Policy)
SUIE_Condition // = (suit-condition-check-content, SUIE_Rep_Policy)
SUIE_Condition // = (suit-condition-abort, SUIE_Rep_Policy)

SUIE_Directive // = (suit-directive-write, SUIE_Rep_Policy)
SUIE_Directive // = (suit-directive-set-component-index, IndexArg)
SUIE_Directive // = (suit-directive-run-sequence,
    bstr .cbor SUIE_Command_Sequence)
SUIE_Directive // = (suit-directive-try-each,
    SUIE_Directive_Try_Each_Argument)
SUIE_Directive // = (suit-directive-override-parameters,
    (+ $$SUIE_Parameters})
SUIE_Directive // = (suit-directive-fetch, SUIE_Rep_Policy)
SUIE_Directive // = (suit-directive-copy, SUIE_Rep_Policy)
SUIE_Directive // = (suit-directive-swap, SUIE_Rep_Policy)
SUIE_Directive // = (suit-directive-invoke, SUIE_Rep_Policy)

SUIE_Directive_Try_Each_Argument = [
    2* bstr .cbor SUIE_Command_Sequence,
    ?nil
]

SUIE_Rep_Policy = uint .bits suit-reporting-bits

suit-reporting-bits = &(
    suit-send-record-success : 0,
    suit-send-record-failure : 1,

```

```

    suit-send-sysinfo-success : 2,
    suit-send-sysinfo-failure : 3
)

$$SUIE_Parameters // = (suit-parameter-vendor-identifier =>
    (RFC4122_UUID / cbor-pen))

cbor-pen = #6.112(bstr)

$$SUIE_Parameters // = (suit-parameter-class-identifier => RFC4122_UUID)
$$SUIE_Parameters // = (suit-parameter-image-digest
    => bstr .cbor SUIE_Digest)
$$SUIE_Parameters // = (suit-parameter-image-size => uint)
$$SUIE_Parameters // = (suit-parameter-component-slot => uint)

$$SUIE_Parameters // = (suit-parameter-uri => tstr)
$$SUIE_Parameters // = (suit-parameter-fetch-arguments => bstr)
$$SUIE_Parameters // = (suit-parameter-source-component => uint)
$$SUIE_Parameters // = (suit-parameter-invoke-args => bstr)

$$SUIE_Parameters // = (suit-parameter-device-identifier => RFC4122_UUID)

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

$$SUIE_Parameters // = (suit-parameter-content => bstr)
$$SUIE_Parameters // = (suit-parameter-strict-order => bool)
$$SUIE_Parameters // = (suit-parameter-soft-failure => bool)

RFC4122_UUID = bstr .size 16

tag38-ltag = text .regexp "[a-zA-Z]{1,8}(-[a-zA-Z0-9]{1,8})*"
SUIE_Text_Map = {
    + tag38-ltag => SUIE_Text_LMap
}
SUIE_Text_LMap = {
    SUIE_Text_Keys,
    * SUIE_Component_Identifier => {
        SUIE_Text_Component_Keys
    }
}

SUIE_Text_Component_Keys = (
    ? suit-text-vendor-name           => tstr,
    ? suit-text-model-name            => tstr,
    ? suit-text-vendor-domain         => tstr,
    ? suit-text-model-info            => tstr,
    ? suit-text-component-description => tstr,
    ? suit-text-component-version     => tstr,

```

```
    * $$suit-text-component-key-extensions
  )

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

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

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

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

;Unseverable, recipient-necessary
suit-manifest-version = 1
suit-manifest-sequence-number = 2
suit-common = 3
suit-reference-uri = 4
suit-validate = 7
suit-load = 8
suit-invoke = 9
;Severable, recipient-necessary
suit-payload-fetch = 16
suit-install = 20
;Severable, recipient-unnecessary
suit-text = 23

suit-components = 2
suit-shared-sequence = 4

suit-command-custom = nint

suit-condition-vendor-identifier = 1
suit-condition-class-identifier = 2
suit-condition-image-match = 3
suit-condition-component-slot = 5
suit-condition-check-content = 6
```

suit-condition-abort	= 14
suit-condition-device-identifier	= 24
suit-directive-set-component-index	= 12
suit-directive-try-each	= 15
suit-directive-write	= 18
suit-directive-override-parameters	= 20
suit-directive-fetch	= 21
suit-directive-copy	= 22
suit-directive-invoke	= 23
suit-directive-swap	= 31
suit-directive-run-sequence	= 32
suit-parameter-vendor-identifier	= 1
suit-parameter-class-identifier	= 2
suit-parameter-image-digest	= 3
suit-parameter-component-slot	= 5
suit-parameter-strict-order	= 12
suit-parameter-soft-failure	= 13
suit-parameter-image-size	= 14
suit-parameter-content	= 18
suit-parameter-uri	= 21
suit-parameter-source-component	= 22
suit-parameter-invoke-args	= 23
suit-parameter-device-identifier	= 24
suit-parameter-fetch-arguments	= 25
suit-parameter-custom	= nint
suit-text-manifest-description	= 1
suit-text-update-description	= 2
suit-text-manifest-json-source	= 3
suit-text-manifest-yaml-source	= 4
suit-text-vendor-name	= 1
suit-text-model-name	= 2
suit-text-vendor-domain	= 3
suit-text-model-info	= 4
suit-text-component-description	= 5
suit-text-component-version	= 6

Appendix B. Examples

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

The examples are signed using the following ECDSA secp256r1 key:

```
-----BEGIN PRIVATE KEY-----
MIGHAgEAMBMGBYqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgApZYjZCUGLM50VBC
CjYStX+09jGmnyJPrpDLTz/hiXOhRANCAASEloEarguqq9JhVxie7NomvqqL8Rtv
P+bitWWchdvArTsfKktsCYExwKNtrNHXi9OB3N+wnAUtszmR23M4tKiW
-----END PRIVATE KEY-----
```

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

```
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEhJaBGq4LqqvSYVcYnuzaJr6qi/Eb
bz/m4rVlnIXbwK07HypLbAmBMCCjbazR14vTgdzfsJwFLbM5kdtzOLSolg==
-----END PUBLIC KEY-----
```

Each example uses SHA256 as the digest function.

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

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

Table 20

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

Table 21

B.1. Example 0: Secure Boot

This example covers the following templates:

- * Compatibility Check (Section 7.1)
- * Secure Boot (Section 7.2)

It also serves as the minimum example.

```
107({
  / authentication-wrapper / 2:<< [
    / digest: / << [
      / algorithm-id / -16 / "sha256" /,
      / digest-bytes /
      h' 6658ea560262696dd1f13b782239a064da7c6c5cbaf52fded428a6fc83c7e5af'
    ] >>,
    / signature: / << 18([
      / protected / << {
        / alg / 1:-7 / "ES256" /
      } >>,
  ] >>,
})
```

```

        / unprotected / {
        },
        / payload / null / nil /,
        / signature / h'408d0816f9b510749bf6a51b066951e08a4438
f849eb092a1ac768eed9de696c1b1dd35d82ef149e6a73a61976ad2cfe78444b806429
3350a122f332cb49f0da'
    ]) >>
] >>,
/ manifest / 3:<< {
  / manifest-version / 1:1,
  / manifest-sequence-number / 2:0,
  / common / 3:<< {
    / components / 2:[
      [h'00']
    ],
    / shared-sequence / 4:<< [
      / directive-override-parameters / 20,{
        / vendor-id /
1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
        / class-id /
2:h'1492af1425695e48bf429b2d51f2ab45' /
1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
        / image-digest / 3:<< [
          / algorithm-id / -16 / "sha256" /,
          / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
        ] >>,
        / image-size / 14:34768
      },
      / condition-vendor-identifier / 1,15,
      / condition-class-identifier / 2,15
    ] >>
  } >>,
  / validate / 7:<< [
    / condition-image-match / 3,15
  ] >>,
  / invoke / 9:<< [
    / directive-invoke / 23,2
  ] >>
} >>
})

```

Total size of Envelope without COSE authentication object: 161

Envelope:

```
d86ba2025827815824822f58206658ea560262696dd1f13b782239a064da
7c6c5cbaf52fded428a6fc83c7e5af035871a50101020003585fa2028181
41000458568614a40150fa6b4a53d5ad5fdfe9de663e4d41ffe02501492
af1425695e48bf429b2d51f2ab45035824822f5820001122334455667788
99aabbccddeeff0123456789abcdeffedcba98765432100e1987d0010f02
0f074382030f0943821702
```

Total size of Envelope with COSE authentication object: 237

Envelope with COSE authentication object:

```
d86ba2025873825824822f58206658ea560262696dd1f13b782239a064da
7c6c5cbaf52fded428a6fc83c7e5af584ad28443a10126a0f65840408d08
16f9b510749bf6a51b066951e08a4438f849eb092a1ac768eed9de696c1b
1dd35d82ef149e6a73a61976ad2cfe78444b8064293350a122f332cb49f0
da035871a50101020003585fa202818141000458568614a40150fa6b4a53
d5ad5fdfe9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab45
035824822f582000112233445566778899aabbccddeeff0123456789abcd
effedcba98765432100e1987d0010f020f074382030f0943821702
```

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

This example covers the following templates:

- * Compatibility Check (Section 7.1)
- * Firmware Download (Section 7.3)

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

```
107({
  / authentication-wrapper / 2:<< [
    / digest: / << [
      / algorithm-id / -16 / "sha256" /,
      / digest-bytes /
h'1f2e7acca0dc2786f2fe4eb947f50873a6a3cfaa98866c5b02e621f42074daf2'
    ] >>,
    / signature: / << 18([
      / protected / << {
        / alg / 1:-7 / "ES256" /
      } >>,
      / unprotected / {
      },
      / payload / null / nil /,
      / signature / h'27a3d7986eddc1bee04e1436746408c308ed3
c15ac590alca0cf96f85671ccac216cb9a1497fc59e21c15f33c95cf75203e25c287b3
1a57d6cd2ef950b27a7a'
```

```

    ]) >>
  ] >>,
  / manifest / 3:<< {
    / manifest-version / 1:1,
    / manifest-sequence-number / 2:1,
    / common / 3:<< {
      / components / 2:[
        [h'00']
      ],
      / shared-sequence / 4:<< [
        / directive-override-parameters / 20,{
          / vendor-id /
1:h'fa6b4a53d5ad5fd9de663e4d41ffe' / fa6b4a53-d5ad-5fd9-
be9d-e663e4d41ffe /,
          / class-id /
2:h'1492af1425695e48bf429b2d51f2ab45' /
1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
          / image-digest / 3:<< [
            / algorithm-id / -16 / "sha256" /,
            / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
          ] >>,
          / image-size / 14:34768
        },
        / condition-vendor-identifier / 1,15,
        / condition-class-identifier / 2,15
      ] >>
    } >>,
    / validate / 7:<< [
      / condition-image-match / 3,15
    ] >>,
    / install / 20:<< [
      / directive-override-parameters / 20,{
        / uri / 21:"http://example.com/file.bin"
      },
      / directive-fetch / 21,2,
      / condition-image-match / 3,15
    ] >>
  } >>
})

```

Total size of Envelope without COSE authentication object: 196

Envelope:

```
d86ba2025827815824822f58201f2e7acca0dc2786f2fe4eb947f50873a6
a3cfaa98866c5b02e621f42074daf2035894a50101020103585fa2028181
41000458568614a40150fa6b4a53d5ad5fd9be9de663e4d41ffe02501492
af1425695e48bf429b2d51f2ab45035824822f5820001122334455667788
99aabbccddeeff0123456789abcdeffedcba98765432100e1987d0010f02
0f074382030f1458258614a115781b687474703a2f2f6578616d706c652e
636f6d2f66696c652e62696e1502030f
```

Total size of Envelope with COSE authentication object: 272

Envelope with COSE authentication object:

```
d86ba2025873825824822f58201f2e7acca0dc2786f2fe4eb947f50873a6
a3cfaa98866c5b02e621f42074daf2584ad28443a10126a0f6584027a3d7
986eddc1bee04e1436746408c308ed3c15ac590a1ca0cf96f85671ccac2
16cb9a1497fc59e21c15f33c95cf75203e25c287b31a57d6cd2ef950b27a
7a035894a50101020103585fa202818141000458568614a40150fa6b4a53
d5ad5fd9be9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab45
035824822f582000112233445566778899aabbccddeeff0123456789abcd
effedcba98765432100e1987d0010f020f074382030f1458258614a11578
1b687474703a2f2f6578616d706c652e636f6d2f66696c652e62696e1502
030f
```

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

This example covers the following templates:

- * Compatibility Check (Section 7.1)
- * Secure Boot (Section 7.2)
- * Firmware Download (Section 7.3)

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

```
107({
  / authentication-wrapper / 2:<< [
    / digest: / << [
      / algorithm-id / -16 / "sha256" / ,
      / digest-bytes /
h' 6a5197ed8f9dccb733d1c89a359441708e070b4c6dcb9a1c2c82c6165f609b90'
    ] >>,
  / signature: / << 18([
    / protected / << {
      / alg / 1:-7 / "ES256" /
    } >>,

```

```

        / unprotected / {
        },
        / payload / null / nil /,
        / signature / h'073d8d80ca67d61cdf04d813c748b2de98fe78
6fc67b764431307c8dbcbe91dc6f762c2c4d7bb998ff9ead4798e03c8ee26b89ef7a9a
d4569f6e187ce89e16c5'
    ]) >>
] >>,
/ manifest / 3:<< {
    / manifest-version / 1:1,
    / manifest-sequence-number / 2:2,
    / common / 3:<< {
        / components / 2:[
            [h'00']
        ],
        / shared-sequence / 4:<< [
            / directive-override-parameters / 20,{
                / vendor-id /
1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
                / class-id /
2:h'1492af1425695e48bf429b2d51f2ab45' /
1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
                / image-digest / 3:<< [
                    / algorithm-id / -16 / "sha256" /,
                    / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
                ] >>,
                / image-size / 14:34768
            },
            / condition-vendor-identifier / 1,15,
            / condition-class-identifier / 2,15
        ] >>
    } >>,
    / reference-uri / 4:"https://git.io/JJYoj",
    / validate / 7:<< [
        / condition-image-match / 3,15
    ] >>,
    / invoke / 9:<< [
        / directive-invoke / 23,2
    ] >>,
    / install / 20:[
        / algorithm-id / -16 / "sha256" /,
        / digest-bytes /
h'cfa90c5c58595e7f5119a72f803fd0370b3e6abbec6315cd38f63135281bc498'
    ],
    / text / 23:[
        / algorithm-id / -16 / "sha256" /,

```

```

        / digest-bytes /
h' 302196d452bce5e8bfeaf71e395645ede6d365e63507a081379721eeecf00007'
    ]
  } >>
})

```

Total size of the Envelope without COSE authentication object or Severable Elements: 257

Envelope:

```

d86ba2025827815824822f58206a5197ed8f9dccf733d1c89a359441708e
070b4c6dcb9a1c2c82c6165f609b900358d1a80101020203585fa2028181
41000458568614a40150fa6b4a53d5ad5fdfe9de663e4d41ffe02501492
af1425695e48bf429b2d51f2ab45035824822f5820001122334455667788
99aabbccddeeff0123456789abcdeffedcba98765432100e1987d0010f02
0f047468747470733a2f2f6769742e696f2f4a4a596f6a074382030f0943
82170214822f5820cfa90c5c58595e7f5119a72f803fd0370b3e6abbec63
15cd38f63135281bc49817822f5820302196d452bce5e8bfeaf71e395645
ede6d365e63507a081379721eeecf00007

```

Total size of the Envelope with COSE authentication object but without Severable Elements: 333

Envelope:

```

d86ba2025873825824822f58206a5197ed8f9dccf733d1c89a359441708e
070b4c6dcb9a1c2c82c6165f609b90584ad28443a10126a0f65840073d8d
80ca67d61cdf04d813c748b2de98fe786fc67b764431307c8dbcbe91dc6f
762c2c4d7bb998ff9ead4798e03c8ee26b89ef7a9ad4569f6e187ce89e16
c50358d1a80101020203585fa202818141000458568614a40150fa6b4a53
d5ad5fdfe9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab45
035824822f582000112233445566778899aabbccddeeff0123456789abcd
effedcba98765432100e1987d0010f020f047468747470733a2f2f676974
2e696f2f4a4a596f6a074382030f094382170214822f5820cfa90c5c5859
5e7f5119a72f803fd0370b3e6abbec6315cd38f63135281bc49817822f58
20302196d452bce5e8bfeaf71e395645ede6d365e63507a081379721eeec
f00007

```

Total size of Envelope with COSE authentication object and Severable Elements: 923

Envelope with COSE authentication object:

```

d86ba4025873825824822f58206a5197ed8f9dccb733d1c89a359441708e
070b4c6dcb9a1c2c82c6165f609b90584ad28443a10126a0f65840073d8d
80ca67d61cdf04d813c748b2de98fe786fc67b764431307c8dbcbe91dc6f
762c2c4d7bb998ff9ead4798e03c8ee26b89ef7a9ad4569f6e187ce89e16
c50358d1a80101020203585fa202818141000458568614a40150fa6b4a53
d5ad5fdfbe9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab45
035824822f582000112233445566778899aabbccddeeff0123456789abcd
effedcba98765432100e1987d0010f020f047468747470733a2f2f676974
2e696f2f4a4a596f6a074382030f094382170214822f5820cfa90c5c5859
5e7f5119a72f803fd0370b3e6abbec6315cd38f63135281bc49817822f58
20302196d452bce5e8bfeaf71e395645ede6d365e63507a081379721eiec
f0000714583c8614a1157832687474703a2f2f6578616d706c652e636f6d
2f766572792f6c6f6e672f706174682f746f2f66696c652f66696c652e62
696e1502030f1759020ba165656e2d5553a20179019d2323204578616d70
6c6520323a2053696d756c74616e656f757320446f776e6c6f61642c2049
6e7374616c6c6174696f6e2c2053656375726520426f6f742c2053657665
726564204669656c64730a0a2020202054686973206578616d706c652063
6f766572732074686520666f6c6c6f77696e672074656d706c617465733a
0a202020200a202020202a20436f6d7061746962696c6974792043686563
6b20287b7b74656d706c6174652d636f6d7061746962696c6974792d6368
65636b7d7d290a202020202a2053656375726520426f6f7420287b7b7465
6d706c6174652d7365637572652d626f6f747d7d290a202020202a204669
726d7761726520446f776e6c6f616420287b7b6669726d776172652d646f
776e6c6f61642d74656d706c6174657d7d290a202020200a202020205468
6973206578616d706c6520616c736f2064656d6f6e737472617465732073
6576657261626c6520656c656d656e747320287b7b6f76722d7365766572
61626c657d7d292c20616e64207465787420287b7b6d616e69666573742d
6469676573742d746578747d7d292e814100a2036761726d2e636f6d0578
525468697320636f6d706f6e656e7420697320612064656d6f6e73747261
74696f6e2e205468652064696765737420697320612073616d706c652070
61747465726e2c206e6f742061207265616c206f6e652e

```

B.4. Example 3: A/B images

This example covers the following templates:

- * Compatibility Check (Section 7.1)
- * Secure Boot (Section 7.2)
- * Firmware Download (Section 7.3)
- * A/B Image Template (Section 7.7)

```

107({
  / authentication-wrapper / 2:<< [
    / digest: / << [
      / algorithm-id / -16 / "sha256" /,
      / digest-bytes /
h' f6d44a62ec906b392500c242e78e908e9cc5057f3f04104a06a8566200da2ee0'
    ] >>,
    / signature: / << 18([
      / protected / << {
        / alg / 1:-7 / "ES256" /
      } >>,
      / unprotected / {
        / payload / null / nil /,
        / signature / h'0bbf7058c1a79dff23c7755d36aae5c6cc1aac
b818f456e2e03f2664c369b9c6700931a52f1f8d808aa4a8e5220d479c9661d2bce0a4
4974004325001e3b1abb'
      }
    ] >>,
  / manifest / 3:<< {
    / manifest-version / 1:1,
    / manifest-sequence-number / 2:3,
    / common / 3:<< {
      / components / 2:[
        [h'00']
      ],
      / shared-sequence / 4:<< [
        / directive-override-parameters / 20,{
          / vendor-id /
1:h' fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
          / class-id /
2:h' 1492af1425695e48bf429b2d51f2ab45' /
1492af14-2569-5e48-bf42-9b2d51f2ab45 /
        },
        / directive-try-each / 15,[
          << [
            / directive-override-parameters / 20,{
              / slot / 5:0
            },
            / condition-component-slot / 5,5,
            / directive-override-parameters / 20,{
              / image-digest / 3:<< [
                / algorithm-id / -16 / "sha256" /,
                / digest-bytes /
h' 00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
              ] >>,
              / image-size / 14:34768
            }
          ]
        ]
      }
    }
  ] >>,
}

```

```

    }
  ] >>,
  << [
    / directive-override-parameters / 20,{
      / slot / 5:1
    },
    / condition-component-slot / 5,5,
    / directive-override-parameters / 20,{
      / image-digest / 3:<< [
        / algorithm-id / -16 / "sha256" /,
        / digest-bytes /
h'0123456789abcdeffedcba987654321000112233445566778899aabbccddeeff'
      ] >>,
      / image-size / 14:76834
    }
  ] >>
],
/ condition-vendor-identifier / 1,15,
/ condition-class-identifier / 2,15
] >>
} >>,
/ validate / 7:<< [
  / condition-image-match / 3,15
] >>,
/ install / 20:<< [
  / directive-try-each / 15,[
    << [
      / directive-override-parameters / 20,{
        / slot / 5:0
      },
      / condition-component-slot / 5,5,
      / directive-override-parameters / 20,{
        / uri / 21:"http://example.com/file1.bin"
      }
    ] >>,
    << [
      / directive-override-parameters / 20,{
        / slot / 5:1
      },
      / condition-component-slot / 5,5,
      / directive-override-parameters / 20,{
        / uri / 21:"http://example.com/file2.bin"
      }
    ] >>
  ],
  / directive-fetch / 21,2,
  / condition-image-match / 3,15
] >>

```

```
    } >>
  })
```

Total size of Envelope without COSE authentication object: 320

Envelope:

```
d86ba2025827815824822f5820f6d44a62ec906b392500c242e78e908e9c
c5057f3f04104a06a8566200da2ee00359010fa5010102030358a4a20281
81410004589b8814a20150fa6b4a53d5ad5fd9de663e4d41ffe025014
92af1425695e48bf429b2d51f2ab450f8258348614a10500050514a20358
24822f582000112233445566778899aabbccddeeff0123456789abcdeffe
dcba98765432100e1987d058368614a10501050514a2035824822f582001
23456789abcdeffedcba987654321000112233445566778899aabbccdde
ff0e1a00012c22010f020f074382030f14585b860f8258288614a1050005
0514a115781c687474703a2f2f6578616d706c652e636f6d2f66696c6531
2e62696e58288614a10501050514a115781c687474703a2f2f6578616d70
6c652e636f6d2f66696c65322e62696e1502030f
```

Total size of Envelope with COSE authentication object: 396

Envelope with COSE authentication object:

```
d86ba2025873825824822f5820f6d44a62ec906b392500c242e78e908e9c
c5057f3f04104a06a8566200da2ee0584ad28443a10126a0f658400bbf70
58c1a79dff23c7755d36aae5c6cc1aacb818f456e2e03f2664c369b9c670
0931a52f1f8d808aa4a8e5220d479c9661d2bce0a44974004325001e3b1a
bb0359010fa5010102030358a4a2028181410004589b8814a20150fa6b4a
53d5ad5fd9de663e4d41ffe02501492af1425695e48bf429b2d51f2ab
450f8258348614a10500050514a2035824822f5820001122334455667788
99aabbccddeeff0123456789abcdeffedcba98765432100e1987d0583686
14a10501050514a2035824822f58200123456789abcdeffedcba98765432
1000112233445566778899aabbccddeeff0e1a00012c22010f020f074382
030f14585b860f8258288614a10500050514a115781c687474703a2f2f65
78616d706c652e636f6d2f66696c65312e62696e58288614a10501050514
a115781c687474703a2f2f6578616d706c652e636f6d2f66696c65322e62
696e1502030f
```

B.5. Example 4: Load from External Storage

This example covers the following templates:

- * Compatibility Check (Section 7.1)
- * Secure Boot (Section 7.2)
- * Firmware Download (Section 7.3)

* Install (Section 7.4)

* Load (Section 7.6)

```

107({
  / authentication-wrapper / 2:<< [
    / digest: / << [
      / algorithm-id / -16 / "sha256" /,
      / digest-bytes /
h' 5b5f6586b1e6cdf19ee479a5adabf206581000bd584b0832a9bdaf4f72cddb6'
    ] >>,
    / signature: / << 18([
      / protected / << {
        / alg / 1:-7 / "ES256" /
      } >>,
      / unprotected / {
      },
      / payload / null / nil /,
      / signature / h'c53c2826b042384e95c646cbcd4308b181f1ed
2bfbeb4e70b93cac9fbdc82e382d877e2c2bcfaf975ffcd36941f2f4db89f68d3c77d6
a3506e9b1509a49dec46'
    ]) >>
  ] >>,
  / manifest / 3:<< {
    / manifest-version / 1:1,
    / manifest-sequence-number / 2:4,
    / common / 3:<< {
      / components / 2:[
        [h'00'],
        [h'02'],
        [h'01']
      ],
    / shared-sequence / 4:<< [
      / directive-set-component-index / 12,0,
      / directive-override-parameters / 20,{
        / vendor-id /
1:h' fa6b4a53d5ad5fdbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
        / class-id /
2:h' 1492af1425695e48bf429b2d51f2ab45' /
1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
        / image-digest / 3:<< [
          / algorithm-id / -16 / "sha256" /,
          / digest-bytes /
h' 00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
        ] >>,
        / image-size / 14:34768
      },
    },
  },

```

```

        / condition-vendor-identifier / 1,15,
        / condition-class-identifier / 2,15
    ] >>
} >>,
/ validate / 7:<< [
    / directive-set-component-index / 12,0,
    / condition-image-match / 3,15
] >>,
/ load / 8:<< [
    / directive-set-component-index / 12,2,
    / directive-override-parameters / 20,{
        / image-digest / 3:<< [
            / algorithm-id / -16 / "sha256" /,
            / digest-bytes /
h'0123456789abcdeffedcba987654321000112233445566778899aabbccddeeff'
        ] >>,
        / image-size / 14:76834,
        / source-component / 22:0 / [h'00'] /
    },
    / directive-copy / 22,2,
    / condition-image-match / 3,15
] >>,
/ invoke / 9:<< [
    / directive-set-component-index / 12,2,
    / directive-invoke / 23,2
] >>,
/ payload-fetch / 16:<< [
    / directive-set-component-index / 12,1,
    / directive-override-parameters / 20,{
        / image-digest / 3:<< [
            / algorithm-id / -16 / "sha256" /,
            / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
        ] >>,
        / uri / 21:"http://example.com/file.bin"
    },
    / directive-fetch / 21,2,
    / condition-image-match / 3,15
] >>,
/ install / 20:<< [
    / directive-set-component-index / 12,0,
    / directive-override-parameters / 20,{
        / source-component / 22:1 / [h'02'] /
    },
    / directive-copy / 22,2,
    / condition-image-match / 3,15
] >>
} >>

```

})

Total size of Envelope without COSE authentication object: 327

Envelope:

d86ba2025827815824822f58205b5f6586b1e6cdf19ee479a5adabf20658
1000bd584b0832a9bdaf4f72cdbdd603590116a801010204035867a20283
814100814102814101045858880c0014a40150fa6b4a53d5ad5fdfe9de6
63e4d41ffe02501492af1425695e48bf429b2d51f2ab45035824822f5820
00112233445566778899aabbccddee0123456789abcdeffedcba987654
32100e1987d0010f020f0745840c00030f085838880c0214a3035824822f
58200123456789abcdeffedcba987654321000112233445566778899aabb
ccddee0e1a00012c2216001602030f0945840c02170210584e880c0114
a2035824822f582000112233445566778899aabbccddee0123456789ab
cdeffedcba987654321015781b687474703a2f2f6578616d706c652e636f
6d2f66696c652e62696e1502030f144b880c0014a116011602030f

Total size of Envelope with COSE authentication object: 403

Envelope with COSE authentication object:

d86ba2025873825824822f58205b5f6586b1e6cdf19ee479a5adabf20658
1000bd584b0832a9bdaf4f72cdbdd6584ad28443a10126a0f65840c53c28
26b042384e95c646cbcd4308b181f1ed2bfbeb4e70b93cac9fbdc82e382d
877e2c2bcfaf975ffcd36941f2f4db89f68d3c77d6a3506e9b1509a49dec
4603590116a801010204035867a20283814100814102814101045858880c
0014a40150fa6b4a53d5ad5fdfe9de663e4d41ffe02501492af1425695e
48bf429b2d51f2ab45035824822f582000112233445566778899aabbccdd
eeff0123456789abcdeffedcba98765432100e1987d0010f020f0745840c
00030f085838880c0214a3035824822f58200123456789abcdeffedcba98
7654321000112233445566778899aabbccddee0e1a00012c2216001602
030f0945840c02170210584e880c0114a2035824822f5820001122334455
66778899aabbccddee0123456789abcdeffedcba987654321015781b68
7474703a2f2f6578616d706c652e636f6d2f66696c652e62696e1502030f
144b880c0014a116011602030f

B.6. Example 5: Two Images

This example covers the following templates:

- * Compatibility Check (Section 7.1)
- * Secure Boot (Section 7.2)
- * Firmware Download (Section 7.3)

Furthermore, it shows using these templates with two images.

```

107({
  / authentication-wrapper / 2:<< [
    / digest: / << [
      / algorithm-id / -16 / "sha256" /,
      / digest-bytes /
h'15ce60f77657e4531dc329155f8b0ed78f94bdc6d165b2665473693dcc34f470'
    ] >>,
    / signature: / << 18([
      / protected / << {
        / alg / 1:-7 / "ES256" /
      } >>,
      / unprotected / {
      },
      / payload / null / nil /,
      / signature / h'53505bf2b1aba7f3c3e142d6c02350daf95331
a8942e77d7378c6670285638e0fe460fe7cebcbe242b14e7ac1a4482cf500136a2568a
92a803f614d5f87ef7a7'
    ]) >>
  ] >>,
  / manifest / 3:<< {
    / manifest-version / 1:1,
    / manifest-sequence-number / 2:5,
    / common / 3:<< {
      / components / 2:[
        [h'00'],
        [h'01']
      ],
      / shared-sequence / 4:<< [
        / directive-set-component-index / 12,0,
        / directive-override-parameters / 20,{
          / vendor-id /
1:h'fa6b4a53d5ad5fdfbe9de663e4d41ffe' / fa6b4a53-d5ad-5fdf-
be9d-e663e4d41ffe /,
          / class-id /
2:h'1492af1425695e48bf429b2d51f2ab45' /
1492af14-2569-5e48-bf42-9b2d51f2ab45 /,
          / image-digest / 3:<< [
            / algorithm-id / -16 / "sha256" /,
            / digest-bytes /
h'00112233445566778899aabbccddeeff0123456789abcdeffedcba9876543210'
          ] >>,
          / image-size / 14:34768
        },
        / condition-vendor-identifier / 1,15,
        / condition-class-identifier / 2,15,
        / directive-set-component-index / 12,1,
        / directive-override-parameters / 20,{
          / image-digest / 3:<< [

```

```

        / algorithm-id / -16 / "sha256" /,
        / digest-bytes /
h'0123456789abcdeffedcba987654321000112233445566778899aabbccddeeff'
    ] >>,
    / image-size / 14:76834
  }
] >>
} >>,
/ validate / 7:<< [
  / directive-set-component-index / 12,0,
  / condition-image-match / 3,15,
  / directive-set-component-index / 12,1,
  / condition-image-match / 3,15
] >>,
/ invoke / 9:<< [
  / directive-set-component-index / 12,0,
  / directive-invoke / 23,2
] >>,
/ install / 20:<< [
  / directive-set-component-index / 12,0,
  / directive-override-parameters / 20,{
    / uri / 21:"http://example.com/file1.bin"
  },
  / directive-fetch / 21,2,
  / condition-image-match / 3,15,
  / directive-set-component-index / 12,1,
  / directive-override-parameters / 20,{
    / uri / 21:"http://example.com/file2.bin"
  },
  / directive-fetch / 21,2,
  / condition-image-match / 3,15
] >>
} >>
})

```

Total size of Envelope without COSE authentication object: 306

Envelope:

```
d86ba2025827815824822f582015ce60f77657e4531dc329155f8b0ed78f
94bdc6d165b2665473693dcc34f47003590101a601010205035895a20282
8141008141010458898c0c0014a40150fa6b4a53d5ad5fdbe9de663e4d4
1ffe02501492af1425695e48bf429b2d51f2ab45035824822f5820001122
33445566778899aabbccddee0123456789abcdeffedcba98765432100e
1987d0010f020f0c0114a2035824822f58200123456789abcdeffedcba98
7654321000112233445566778899aabbccddee0e1a00012c220749880c
00030f0c01030f0945840c00170214584f900c0014a115781c687474703a
2f2f6578616d706c652e636f6d2f66696c65312e62696e1502030f0c0114
a115781c687474703a2f2f6578616d706c652e636f6d2f66696c65322e62
696e1502030f
```

Total size of Envelope with COSE authentication object: 382

Envelope with COSE authentication object:

```
d86ba2025873825824822f582015ce60f77657e4531dc329155f8b0ed78f
94bdc6d165b2665473693dcc34f470584ad28443a10126a0f6584053505b
f2blaba7f3c3e142d6c02350daf95331a8942e77d7378c6670285638e0fe
460fe7cebcbe242b14e7ac1a4482cf500136a2568a92a803f614d5f87ef7
a703590101a601010205035895a202828141008141010458898c0c0014a4
0150fa6b4a53d5ad5fdbe9de663e4d41ffe02501492af1425695e48bf42
9b2d51f2ab45035824822f582000112233445566778899aabbccddee01
23456789abcdeffedcba98765432100e1987d0010f020f0c0114a2035824
822f58200123456789abcdeffedcba987654321000112233445566778899
aabbccddee0e1a00012c220749880c00030f0c01030f0945840c001702
14584f900c0014a115781c687474703a2f2f6578616d706c652e636f6d2f
66696c65312e62696e1502030f0c0114a115781c687474703a2f2f657861
6d706c652e636f6d2f66696c65322e62696e1502030f
```

Appendix C. Design Rationale

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

The specialized byte code approach gives benefits equivalent to those provided by a scripting language or conventional byte code, with two substantial differences. First, the language is extremely high level, consisting of only the operations that a device may perform during update and trusted invocation of a firmware image. Second,

the language specifies linear behavior, without reverse branches. Conditional processing is supported, and parallel and out-of-order processing may be performed by sufficiently capable devices.

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

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

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

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

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

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

C.1. C.1 Design Rationale: Envelope

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

1. This enables the use of Severable Elements (Section 8.6)
2. This enables modular processing of manifests, particularly with large signatures.

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

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

Four authentication objects are supported by the Envelope:

- * COSE_Sign_Tagged
- * COSE_Sign1_Tagged
- * COSE_Mac_Tagged
- * COSE_Mac0_Tagged

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

C.2. C.2 Byte String Wrappers

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

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

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

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

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

Authors' Addresses

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

Hannes Tschofenig
University of Applied Sciences Bonn-Rhein-Sieg
Germany
Email: Hannes.Tschofenig@gmx.net

Henk Birkholz
Fraunhofer SIT
Rheinstrasse 75
64295 Darmstadt
Germany
Email: henk.birkholz@sit.fraunhofer.de

Koen Zandberg
Inria
Email: koen.zandberg@inria.fr

Øyvind Rønningstad
Nordic Semiconductor
Email: oyvind.ronningstad@gmail.com

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

B. Moran
Arm Limited
Å\230. RÅ, nningstad
Nordic Semiconductor
A. Tsukamoto
Openchip & Software Technologies, S.L.
22 July 2025

Cryptographic Algorithms for Internet of Things (IoT) Devices
draft-ietf-suit-mti-23

Abstract

The SUIT manifest, as defined in "A Manifest Information Model for Firmware Updates in Internet of Things (IoT) Devices" (RFC 9124), provides a flexible and extensible format for describing how firmware and software updates are to be fetched, verified, decrypted, and installed on resource-constrained devices. To ensure the security of these update processes, the manifest relies on cryptographic algorithms for functions such as digital signature verification, integrity checking, and confidentiality.

This document defines cryptographic algorithm profiles for use with the Software Updates for Internet of Things (SUIT) manifest. These profiles specify sets of algorithms to promote interoperability across implementations.

Given the diversity of IoT deployments and the evolving cryptographic landscape, algorithm agility is essential. This document groups algorithms into named profiles to accommodate varying levels of device capabilities and security requirements. These profiles support the use cases laid out in the SUIT architecture, published in "A Firmware Update Architecture for Internet of Things" (RFC 9019).

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

Copyright (c) 2025 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1.	Introduction	3
2.	Conventions and Definitions	4
3.	Profiles	4
3.1.	Profile suit-sha256-hmac-a128kw-a128ctr	5
3.2.	Profile suit-sha256-esp256-ecdh-a128ctr	5
3.3.	Profile suit-sha256-ed25519-ecdh-a128ctr	6
3.4.	Profile suit-sha256-esp256-ecdh-a128gcm	6
3.5.	Profile suit-sha256-ed25519-ecdh-chacha-poly	7
3.6.	Profile suit-sha256-hsslms-a256kw-a256ctr	7
4.	Security Considerations	8
4.1.	Payload Encryption as Part of a Defense-in-Depth Strategy	8
4.2.	Use of AES-CTR in Payload Encryption	9
5.	Operational Considerations	9
5.1.	Profile Support Discovery	9
5.2.	Profile Selection and Control	10
5.3.	Profile Provisioning and Constraints	10
5.4.	Logging and Reporting	11
6.	IANA Considerations	11
6.1.	Profile: suit-sha256-hmac-a128kw-a128ctr	12
6.2.	Profile: suit-sha256-esp256-ecdh-a128ctr	12
6.3.	Profile: suit-sha256-ed25519-ecdh-a128ctr	12
6.4.	Profile: suit-sha256-esp256-ecdh-a128gcm	13
6.5.	Profile: suit-sha256-ed25519-ecdh-chacha-poly	13
6.6.	Profile: suit-sha256-hsslms-a256kw-a256ctr	13

7. References	14
7.1. Normative References	14
7.2. Informative References	15
Appendix A. Full CDDL	16
Appendix B. Acknowledgments	19
Authors' Addresses	19

1. Introduction

This document defines algorithm profiles, in IANA registry (Section 6), intended for authors of Software Updates for Internet of Things (SUIT) manifests and their recipients, with the goal of promoting interoperability in software update scenarios for constrained nodes. These profiles specify sets of algorithms that are tailored to the evolving security landscape, recognizing that cryptographic requirements may change over time.

The following profiles are defined:

- * One profile designed for constrained devices that support only symmetric key cryptography
- * Two profiles for constrained devices capable of using asymmetric key cryptography
- * Two profiles that employ Authenticated Encryption with Associated Data (AEAD) ciphers
- * One constrained asymmetric profile that uses a hash-based signature scheme

Due to the asymmetric nature of SUIT deployments - where manifest authors typically operate in resource-rich environments while recipients are resource-constrained - the cryptographic requirements differ between these two roles.

This specification uses AES-CTR in combination with a digest algorithm, as defined in [RFC9459], to support use cases that require out-of-order block reception and decryption-capabilities not offered by AEAD algorithms. For further discussion of these constrained use cases, refer to Section 4.2. Other SUIT use cases (see [I-D.ietf-suit-manifest]) may define different profiles.

2. Conventions and Definitions

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.

This specification uses the following abbreviations:

- * Advanced Encryption Standard (AES)
- * AES Counter (AES-CTR) Mode
- * AES Key Wrap (AES-KW)
- * Authenticated Encryption with Associated Data (AEAD)
- * Concise Binary Object Representation (CBOR)
- * CBOR Object Signing and Encryption (COSE)
- * Concise Data Definition Language (CDDL)
- * Elliptic Curve Diffie-Hellman Ephemeral-Static (ECDH-ES)
- * Hash-based Message Authentication Code (HMAC)
- * Hierarchical Signature System / Leighton-Micali Signature (HSS/LMS)
- * Software Updates for Internet of Things (SUIT)

SUIT specifically addresses the requirements of constrained devices and networks, as described in [RFC9019].

The terms "Author", "Recipient", and "Manifest" are defined in [I-D.ietf-suit-manifest].

3. Profiles

Each profile, in IANA registry (Section 6), consists of algorithms from the following categories:

- * Digest Algorithms
- * Authentication Algorithms

- * Key Exchange Algorithms (optional)
- * Encryption Algorithms (optional)

Each profile references specific algorithm identifiers, as defined in [IANA-COSE]. Since these algorithm identifiers are used in the context of the IETF SUIF manifest [I-D.ietf-suit-manifest], they are represented using CBOR Object Signing and Encryption (COSE) structures as defined in [RFC9052] and [RFC9053].

The use of the profiles by authors and recipients is based on the following assumptions:

- * Recipients MAY choose which profile they wish to implement. It is RECOMMENDED that they implement the `suit-sha256-hsslms-a256kw-a256ctr` profile (Section 3.6). Recipients MAY implement any number of other profiles not defined in this document. Recipients MAY choose not to implement encryption and the corresponding key exchange algorithms if they do not intend to support encrypted payloads.
- * Authors MUST implement all profiles with a status set to 'MANDATORY' in Section 6. Authors MAY implement any number of additional profiles.

3.1. Profile `suit-sha256-hmac-a128kw-a128ctr`

This profile only offers support for symmetric cryptographic algorithms.

Algorithm Type	Algorithm	COSE Key
Digest	SHA-256	-16
Authentication	HMAC-256	5
Key Exchange	A128KW Key Wrap	-3
Encryption	A128CTR	-65534

Table 1

3.2. Profile `suit-sha256-esp256-ecdh-a128ctr`

This profile supports asymmetric algorithms for use with constrained devices.

Algorithm Type	Algorithm	COSE Key
Digest	SHA-256	-16
Authentication	ESP256	-9
Key Exchange	ECDH-ES + A128KW	-29
Encryption	A128CTR	-65534

Table 2

3.3. Profile suit-sha256-ed25519-ecdh-a128ctr

This profile supports an alternative choice of asymmetric algorithms for use with constrained devices.

Algorithm Type	Algorithm	COSE Key
Digest	SHA-256	-16
Authentication	Ed25519	-19
Key Exchange	ECDH-ES + A128KW	-29
Encryption	A128CTR	-65534

Table 3

3.4. Profile suit-sha256-esp256-ecdh-a128gcm

This profile supports asymmetric algorithms in combination with AEAD algorithms.

Algorithm Type	Algorithm	COSE Key
Digest	SHA-256	-16
Authentication	ESP256	-9
Key Exchange	ECDH-ES + A128KW	-29
Encryption	A128GCM	1

Table 4

3.5. Profile suit-sha256-ed25519-ecdh-chacha-poly

This profile also supports asymmetric algorithms with AEAD algorithms but offers an alternative to suit-sha256-esp256-ecdh-a128gcm.

Algorithm Type	Algorithm	COSE Key
Digest	SHA-256	-16
Authentication	Ed25519	-19
Key Exchange	ECDH-ES + A128KW	-29
Encryption	ChaCha20/Poly1305	24

Table 5

3.6. Profile suit-sha256-hsslms-a256kw-a256ctr

This profile utilizes a stateful hash-based signature algorithm, namely the Hierarchical Signature System / Leighton-Micali Signature (HSS/LMS), as a unique alternative to the profiles listed above.

A note regarding the use of the HSS/LMS: The decision as to how deep the tree is, is a decision that affects authoring tools only (see [RFC8778]). Verification is not affected by the choice of the "W" parameter, but the size of the signature is affected. To support the long lifetimes required by IoT devices, it is RECOMMENDED to use trees with greater height (see Section 2.2 of [RFC8778]).

Algorithm Type	Algorithm	COSE Key
Digest	SHA-256	-16
Authentication	HSS/LMS	-46
Key Exchange	A256KW	-5
Encryption	A256CTR	-65532

Table 6

4. Security Considerations

Payload encryption is used to protect sensitive content such as machine learning models, proprietary algorithms, and personal data [RFC6973]. In the context of SUIT, the primary purpose of payload encryption is to defend against unauthorized observation during distribution. By encrypting the payload, confidential information can be safeguarded from eavesdropping.

However, encrypting firmware or software update payloads on commodity devices do not constitute an effective cybersecurity defense against targeted attacks. Once an attacker gains access to a device, they may still be able to extract the plaintext payload.

4.1. Payload Encryption as Part of a Defense-in-Depth Strategy

To define the purpose of payload encryption as a defensive cybersecurity tool, it is important to define the capabilities of modern threat actors. A variety of capabilities are possible:

- * find bugs by binary code inspection
- * send unexpected data to communication interfaces, looking for unexpected behavior
- * use fault injection to bypass or manipulate code
- * use communication attacks or fault injection along with gadgets found in the code

Given this range of capabilities, it is important to understand which capabilities are impacted by firmware encryption. Threat actors who find bugs by manual inspection or use gadgets found in the code will need to first extract the code from the target. In the IoT context, it is expected that most threat actors will start with sample devices and physical access to test attacks.

Due to these factors, payload encryption serves to limit the pool of attackers to those who have the technical capability to extract code from physical devices and those who perform code-free attacks.

4.2. Use of AES-CTR in Payload Encryption

AES-CTR mode with a digest is specified, see [RFC9459]. All of the AES-CTR security considerations in [RFC9459] apply. See [I-D.ietf-suit-firmware-encryption] for additional background information.

5. Operational Considerations

While this document focuses on the cryptographic aspects of manifest processing, several operational and manageability considerations are relevant when deploying these profiles in practice.

5.1. Profile Support Discovery

To enable interoperability of the described profiles, it is important for a manifest author to determine which profiles are supported by a device. Furthermore, it is also important for the author and the distribution system (see Section 3 of [I-D.ietf-suit-firmware-encryption]) to know whether firmware for a particular device or family of devices needs to be encrypted, and which key distribution mechanism shall be used. This information can be obtained through:

- * Manual configuration.
- * Device management systems, as described in [RFC9019], which typically maintain metadata about device capabilities and their lifecycle status. These systems may use proprietary or standardized management protocols to expose supported features. LwM2M [LwM2M] is one such standardized protocol. The Trusted Execution Environment Provisioning (TEEP) protocol [I-D.ietf-teep-protocol] is another option.

- * Capability reporting mechanisms, such as those described in [I-D.ietf-suit-report], which define structures that allow a device to communicate supported SUIT features and cryptographic capabilities to a management or attestation entity.

5.2. Profile Selection and Control

When a device supports multiple algorithm profiles, it is expected that the SUIT manifest author indicates the appropriate profile based on the intended recipient(s) and other policies. The manifest itself indicates which algorithms are used; devices are expected to validate manifests using supported algorithms.

Devices do not autonomously choose which profile to apply; rather, they either accept or reject a manifest based on the algorithm profile it uses. There is no protocol-level negotiation of profiles at SUIT manifest processing time. Any dynamic profile selection or configuration is expected to occur as part of other protocols, for example, through device management.

5.3. Profile Provisioning and Constraints

Provisioning for a given profile may include:

- * Installation of trust anchors for acceptable signers.
- * Distribution of keys used by the content key distribution mechanism (see Section 4 of [I-D.ietf-suit-firmware-encryption]).
- * Availability of specific cryptographic libraries or hardware support (e.g., for post-quantum algorithms or AEAD ciphers).
- * Evaluation of the required storage and processing resources for the selected profile.
- * Support for manifest processing capabilities.

There may be conditions under which switching to a different algorithm profile is not feasible, such as:

- * Lack of hardware support (e.g., no crypto acceleration).
- * Resource limitations on memory-constrained devices (e.g., insufficient flash or RAM).
- * Deployment policy constraints or regulatory compliance requirements.

In such cases, a device management or update orchestration system should take these constraints into account when constructing and distributing manifests.

5.4. Logging and Reporting

Implementations MAY log failures to process a manifest due to unsupported algorithm profiles or unavailable cryptographic functionality. When supported, such events SHOULD be reported using secure mechanisms, such as those described in [I-D.ietf-suit-report], to assist operators in diagnosing update failures or misconfigurations.

6. IANA Considerations

IANA is requested to create a new "COSE SUIT Algorithm Profiles" registry, to be located within its own self-titled registry group. The registry will be listed in the "Software Update for the Internet of Things (SUIT)" category at <https://www.iana.org/protocols>.

While most profile attributes are self-explanatory, the status field warrants a brief explanation. This field can take one of three values: MANDATORY, NOT RECOMMENDED, or OPTIONAL.

- * MANDATORY indicates that the profile is mandatory to implement for manifest authors.
- * NOT RECOMMENDED means that the profile should generally be avoided in new implementations.
- * OPTIONAL suggests that support for the profile is permitted but not required.

IANA is requested to add a note that mirrors these status values to the registry.

Adding new profiles or updating the status of existing profiles requires Standards Action (Section 4.9 of [RFC8126]).

As time progresses, algorithm profiles may lose their MANDATORY status. When this occurs, their status may be changed to either OPTIONAL or NOT RECOMMENDED for new implementations. Similarly, a profile may be transitioned from OPTIONAL to NOT RECOMMENDED. However, profiles once marked as OPTIONAL or NOT RECOMMENDED MUST NOT be transitioned to MANDATORY status in future revisions. Since it may be impossible to update certain parts of IoT device firmware in the field, such as first-stage bootloaders, support for all relevant algorithms will almost always be required by authoring tools.

The initial content of the "COSE SUIE Algorithm Profiles" registry is:

6.1. Profile: suit-sha256-hmac-a128kw-a128ctr

* Profile: suit-sha256-hmac-a128kw-a128ctr

* Status: MANDATORY

* Digest: -16

* Auth: 5

* Key Exchange: -3

* Encryption: -65534

* Descriptor Array: [-16, 5, -3, -65534]

* Reference: Section 3.1 of THIS_DOCUMENT

6.2. Profile: suit-sha256-esp256-ecdh-a128ctr

* Profile: suit-sha256-esp256-ecdh-a128ctr

* Status: MANDATORY

* Digest: -16

* Auth: -9

* Key Exchange: -29

* Encryption: -65534

* Descriptor Array: [-16, -9, -29, -65534]

* Reference: Section 3.2 of THIS_DOCUMENT

6.3. Profile: suit-sha256-ed25519-ecdh-a128ctr

* Profile: suit-sha256-ed25519-ecdh-a128ctr

* Status: MANDATORY

* Digest: -16

* Auth: -19

- * Key Exchange: -29
 - * Encryption: -65534
 - * Descriptor Array: [-16, -19, -29, -65534]
 - * Reference: Section 3.3 of THIS_DOCUMENT
- 6.4. Profile: suit-sha256-esp256-ecdh-a128gcm
- * Profile: suit-sha256-esp256-ecdh-a128gcm
 - * Status: MANDATORY
 - * Digest: -16
 - * Auth: -9
 - * Key Exchange: -29
 - * Encryption: 1
 - * Descriptor Array: [-16, -9, -29, 1]
 - * Reference: Section 3.4 of THIS_DOCUMENT
- 6.5. Profile: suit-sha256-ed25519-ecdh-chacha-poly
- * Profile: suit-sha256-ed25519-ecdh-chacha-poly
 - * Status: MANDATORY
 - * Digest: -16
 - * Auth: -19
 - * Key Exchange: -29
 - * Encryption: 24
 - * Descriptor Array: [-16, -19, -29, 24]
 - * Reference: Section 3.5 of THIS_DOCUMENT
- 6.6. Profile: suit-sha256-hsslms-a256kw-a256ctr
- * Profile: suit-sha256-hsslms-a256kw-a256ctr

- * Status: MANDATORY
- * Digest: -16
- * Auth: -46
- * Key Exchange: -5
- * Encryption: -65532
- * Descriptor Array: [-16, -46, -5, -65532]
- * Reference: Section 3.6 of THIS_DOCUMENT

7. References

7.1. Normative References

[I-D.ietf-suit-manifest]

Moran, B., Tschofenig, H., Birkholz, H., Zandberg, K., and O. R 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>>.

[IANA-COSE]

"CBOR Object Signing and Encryption (COSE)", 2022, <<https://www.iana.org/assignments/cose/cose.xhtml>>.

[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>>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/rfc/rfc8126>>.

[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>>.
- [RFC8778] Housley, R., "Use of the HSS/LMS Hash-Based Signature Algorithm with CBOR Object Signing and Encryption (COSE)", RFC 8778, DOI 10.17487/RFC8778, April 2020, <<https://www.rfc-editor.org/rfc/rfc8778>>.
- [RFC9052] Schaad, J., "CBOR Object Signing and Encryption (COSE): Structures and Process", STD 96, RFC 9052, DOI 10.17487/RFC9052, August 2022, <<https://www.rfc-editor.org/rfc/rfc9052>>.
- [RFC9053] Schaad, J., "CBOR Object Signing and Encryption (COSE): Initial Algorithms", RFC 9053, DOI 10.17487/RFC9053, August 2022, <<https://www.rfc-editor.org/rfc/rfc9053>>.
- [RFC9459] Housley, R. and H. Tschofenig, "CBOR Object Signing and Encryption (COSE): AES-CTR and AES-CBC", RFC 9459, DOI 10.17487/RFC9459, September 2023, <<https://www.rfc-editor.org/rfc/rfc9459>>.

7.2. Informative 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-report]
Moran, B. and H. Birkholz, "Secure Reporting of Update Status", Work in Progress, Internet-Draft, draft-ietf-suit-report-14, 22 July 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-suit-report-14>>.

- [I-D.ietf-teep-protocol] Tschofenig, H., Pei, M., Wheeler, D. M., Thaler, D., and A. Tsukamoto, "Trusted Execution Environment Provisioning (TEEP) Protocol", Work in Progress, Internet-Draft, draft-ietf-teep-protocol-21, 3 March 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-teep-protocol-21>>.
- [Lwm2M] Open Mobile Alliance, "OMA Lightweight M2M", 20 April 2025, <<https://www.openmobilealliance.org/specifications/lwm2m>>.
- [RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <<https://www.rfc-editor.org/rfc/rfc6973>>.
- [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>>.

Appendix A. Full CDDL

The following CDDL snippet [RFC8610] creates a subset of COSE for use with SUIT. Both tagged and untagged messages are defined. SUIT only uses tagged COSE messages, but untagged messages are also defined for use in protocols that share a ciphersuite with SUIT.

To be valid, the following CDDL MUST have the COSE CDDL appended to it. The COSE CDDL can be obtained by following the directions in [RFC9053], Section 1.4.

===== NOTE: '\ ' line wrapping per RFC 8792 =====

```
SUIT_COSE_tool_tweak /= suit-sha256-hmac-a128kw-a128ctr
SUIT_COSE_tool_tweak /= suit-sha256-esp256-ecdh-a128ctr
SUIT_COSE_tool_tweak /= suit-sha256-ed25519-ecdh-a128ctr
SUIT_COSE_tool_tweak /= suit-sha256-esp256-ecdh-a128gcm
SUIT_COSE_tool_tweak /= suit-sha256-ed25519-ecdh-chacha-poly
SUIT_COSE_tool_tweak /= suit-sha256-hsslms-a256kw-a256ctr
SUIT_COSE_tool_tweak /= SUIT_COSE_Profiles

SUIT_COSE_Profiles /= SUIT_COSE_Profile_HMAC_A128KW_A128CTR
SUIT_COSE_Profiles /= SUIT_COSE_Profile_ESP256_ECDH_A128CTR
SUIT_COSE_Profiles /= SUIT_COSE_Profile_ED25519_ECDH_A128CTR
SUIT_COSE_Profiles /= SUIT_COSE_Profile_ESP256_ECDH_A128GCM
```

```

SUIT_COSE_Profiles /= \
    SUIT_COSE_Profile_ED25519_ECDH_CHACHA20_POLY1304
SUIT_COSE_Profiles /= SUIT_COSE_Profile_HSSLMS_A256KW_A256CTR

suit-sha256-hmac-a128kw-a128ctr    = [-16, 5, -3, -65534]
suit-sha256-esp256-ecdh-a128ctr   = [-16, -9, -29, -65534]
suit-sha256-ed25519-ecdh-a128ctr  = [-16, -19, -29, -65534]
suit-sha256-esp256-ecdh-a128gcm   = [-16, -9, -29, 1]
suit-sha256-ed25519-ecdh-chacha-poly = [-16, -19, -29, 24]
suit-sha256-hsslms-a256kw-a256ctr = [-16, -46, -5, -65532]

SUIT_COSE_Profile_HMAC_A128KW_A128CTR =
    SUIT_COSE_Profile<5,-65534> .and COSE_Messages
SUIT_COSE_Profile_ESP256_ECDH_A128CTR =
    SUIT_COSE_Profile<-9,-65534> .and COSE_Messages
SUIT_COSE_Profile_ED25519_ECDH_A128CTR =
    SUIT_COSE_Profile<-19,-65534> .and COSE_Messages
SUIT_COSE_Profile_ESP256_ECDH_A128GCM =
    SUIT_COSE_Profile<-9,1> .and COSE_Messages
SUIT_COSE_Profile_ED25519_ECDH_CHACHA20_POLY1304 =
    SUIT_COSE_Profile<-19,24> .and COSE_Messages
SUIT_COSE_Profile_HSSLMS_A256KW_A256CTR =
    SUIT_COSE_Profile<-46,-65532> .and COSE_Messages

SUIT_COSE_Profile<authid, encid> = SUIT_COSE_Messages<authid,encid>

SUIT_COSE_Messages<authid, encid> =
    SUIT_COSE_Untagged_Message<authid, encid> /
    SUIT_COSE_Tagged_Message<authid, encid>

SUIT_COSE_Untagged_Message<authid, encid> = SUIT_COSE_Sign<authid> /
    SUIT_COSE_Sign1<authid> / SUIT_COSE_Encrypt<encid> /
    SUIT_COSE_Encrypt0<encid> / SUIT_COSE_Mac<authid> /
    SUIT_COSE_Mac0<authid>

SUIT_COSE_Tagged_Message<authid, encid> =
    SUIT_COSE_Sign_Tagged<authid> / SUIT_COSE_Sign1_Tagged<authid> /
    SUIT_COSE_Encrypt_Tagged<encid> / SUIT_COSE_Encrypt0_Tagged<\
        encid> /
    SUIT_COSE_Mac_Tagged<authid> / SUIT_COSE_Mac0_Tagged<authid>

; Note: This is not the same definition as is used in COSE.
; It restricts a COSE header definition further without
; repeating the COSE definition. It should be merged
; with COSE by using the CDDL .and operator.
SUIT_COSE_Profile-Headers<algid> = (
    protected : bstr .cbor SUIT_COSE_alg_map<algid>,
    unprotected : SUIT_COSE_header_map

```

```
)
SUIT_COSE_alg_map<algid> = {
    1 => algid,
    * int => any
}

SUIT_COSE_header_map = {
    * int => any
}

SUIT_COSE_Sign_Tagged<authid> = #6.98(SUIT_COSE_Sign<authid>)

SUIT_COSE_Sign<authid> = [
    SUIT_COSE_Profile_Headers<authid>,
    payload : bstr / nil,
    signatures : [+ SUIT_COSE_Signature<authid>]
]

SUIT_COSE_Signature<authid> = [
    SUIT_COSE_Profile_Headers<authid>,
    signature : bstr
]

SUIT_COSE_Sign1_Tagged<authid> = #6.18(SUIT_COSE_Sign1<authid>)

SUIT_COSE_Sign1<authid> = [
    SUIT_COSE_Profile_Headers<authid>,
    payload : bstr / nil,
    signature : bstr
]

SUIT_COSE_Encrypt_Tagged<encid> = #6.96(SUIT_COSE_Encrypt<encid>)

SUIT_COSE_Encrypt<encid> = [
    SUIT_COSE_Profile_Headers<encid>,
    ciphertext : bstr / nil,
    recipients : [+SUIT_COSE_recipient<encid>]
]

SUIT_COSE_recipient<encid> = [
    SUIT_COSE_Profile_Headers<encid>,

```

```
    ciphertext : bstr / nil,  
    ? recipients : [+SUIT_COSE_recipient<encid>]  
  ]
```

SUIT_COSE_Encrypt0_Tagged<encid> = #6.16(SUIT_COSE_Encrypt0<encid>)

```
SUIT_COSE_Encrypt0<encid> = [  
  SUIT_COSE_Profile_Headers<encid>,  
  ciphertext : bstr / nil,  
]
```

SUIT_COSE_Mac_Tagged<authid> = #6.97(SUIT_COSE_Mac<authid>)

```
SUIT_COSE_Mac<authid> = [  
  SUIT_COSE_Profile_Headers<authid>,  
  payload : bstr / nil,  
  tag : bstr,  
  recipients : [+SUIT_COSE_recipient<authid>]  
]
```

SUIT_COSE_Mac0_Tagged<authid> = #6.17(SUIT_COSE_Mac0<authid>)

```
SUIT_COSE_Mac0<authid> = [  
  SUIT_COSE_Profile_Headers<authid>,  
  payload : bstr / nil,  
  tag : bstr,  
]
```

Appendix B. Acknowledgments

We would like to specifically thank Henk Birkholz, Mohamed Boucadair, Deb Cooley, Lorenzo Corneo, Linda Dunbar, Russ Housley, Michael B. Jones, Jouni Korhonen, Magnus Nyström, Michael Richardson, and Hannes Tschofenig for their review comments.

Authors' Addresses

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

Øyvind Rønningstad
Nordic Semiconductor
Email: oyvind.ronningstad@gmail.com

Akira Tsukamoto
Openchip & Software Technologies, S.L.
Email: akira.tsukamoto@gmail.com

SUIT
Internet-Draft
Updates: draft-ietf-suit-manifest (if approved)
Intended status: Standards Track
Expires: 4 September 2025

B. Moran
Arm Limited
H. Tschofenig
3 March 2025

Strong Assertions of IoT Network Access Requirements
draft-ietf-suit-mud-10

Abstract

The Manufacturer Usage Description (MUD) specification describes the access and network functionality required for a device to properly function. This description has to reflect the software running on the device and its configuration. Because of this, the most appropriate entity for describing device network access requirements is the same as the entity developing the software and its configuration.

A network presented with a MUD file by a device allows detection of misbehavior by the device software and configuration of access control.

This document defines a way to link the Software Updates for Internet of Things (SUIT) manifest to a MUD file offering a stronger binding between the two.

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 4 September 2025.

Copyright Notice

Copyright (c) 2025 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

- 1. Introduction 2
- 2. Terminology 4
- 3. Workflow 4
- 4. Operational Considerations 6
 - 4.1. Pros 6
 - 4.2. Cons 7
- 5. Extensions to SUIT 8
- 6. Security Considerations 8
- 7. IANA Considerations 9
- 8. References 9
 - 8.1. Normative References 9
 - 8.2. Informative References 10
- Acknowledgements 11
- Authors' Addresses 11

1. Introduction

A Manufacturer Usage Description (MUD) file describes what sort of network communication behavior a device is designed to have. For example, a manufacturer may use a MUD file to describe that a device uses HTTP, DNS and NTP communication but no other protocols. The communication patterns are described in a JSON-based format in the MUD file.

The MUD files do, however, need to be presented by the device to a MUD manager in the operational network where the device is deployed. Under [RFC8520], devices report a MUD URL to a MUD manager in the operational network. The MUD URL is a URL that can be used by the MUD manager to receive the MUD file from a MUD file server to ultimately obtain the MUD file.

Figure 1 shows the MUD architecture, as defined in RFC 8520.

This specification defines an extension to the Software Updates for Internet of Things (SUIT) manifest [I-D.ietf-suit-manifest] to include a MUD URL. A SUIT manifest is a bundle of metadata about code/data for an IoT device, where to find the code/data, the devices to which it applies, and cryptographic information protecting the manifest.

When combining a MUD URL with a manifest used for software/firmware updates then a network operator has more confidence in the description of the communication requirements for a device to properly function.

2. 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.

This document re-uses the terms defined in [RFC9334] related to remote attestation. Readers of this document are assumed to be familiar with the following terms: Evidence, Claim, Attester, Verifier, and Relying Party (RP).

This document also uses terms defined in [RFC8520], such as MUD, MUD file, MUD manager, MUD URL, etc.

3. Workflow

Figure 2 shows the architectural extensions introduced by combining SUIT and MUD. The key elements are that the developer, who produces the firmware is also generating a manifest and the MUD file. Information about the MUD file is embedded into the SUIT manifest and provided to the device via firmware update mechanism. Once this information is available on the device it can be presented during device onboarding, during network access authentication, or as part of other interactions that involve the conveyance of Evidence to the operational network. After retrieving the manifest, the MUD file can be obtained as well.

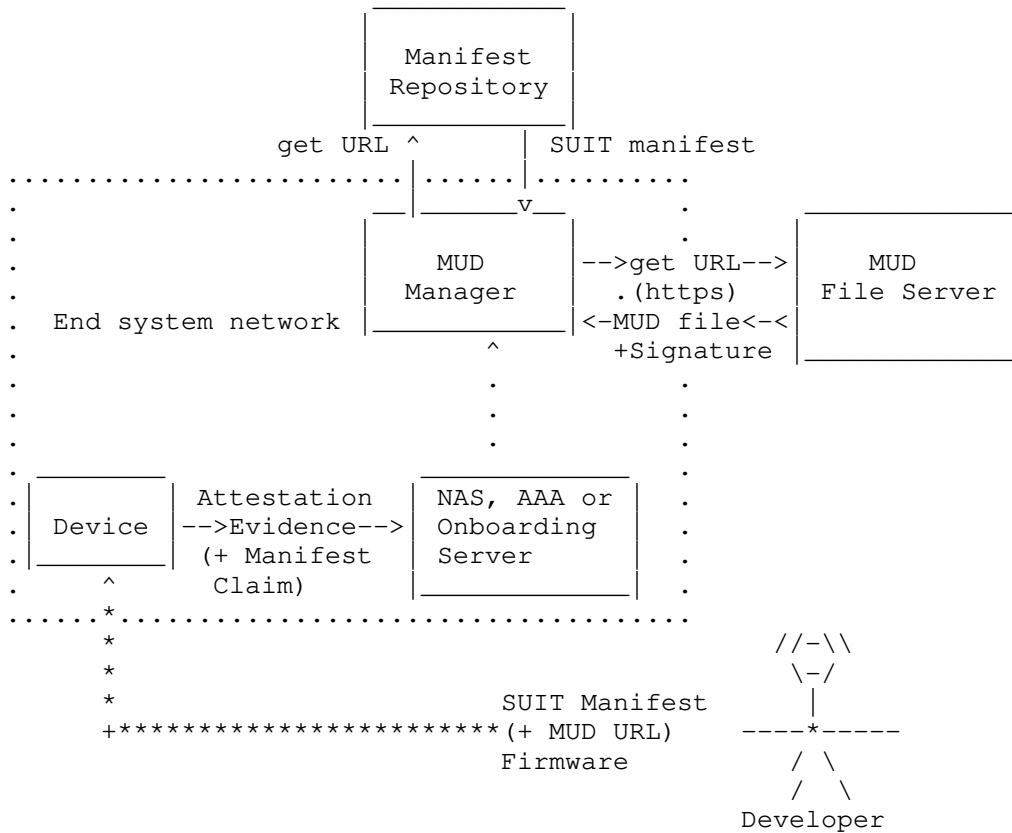


Figure 2: SUIT-MUD Architecture.

The intended workflow is as follows, and assumes an attestation mechanism between the device and the MUD Manager:

- * At the time of onboarding, devices report their manifest in use to the MUD Manager via some form of attestation Evidence and a conveyance protocol. The device thereby acts as an Attester. The normative specification of these mechanisms is out of scope for this document.
- * An example of an Evidence format is the Entity Attestation Token (EAT) [I-D.ietf-rats-eat], which offers a rich set of claims. This specification assumes that Evidence includes a link to the SUIT manifest via the "manifests" claim (see Section 4.2.15 of [I-D.ietf-rats-eat]) or that the manifest itself is embedded in the Evidence. This Evidence is conveyed to the operational network via some protocol, such as network access authentication

protocol (for example using the EAP-TLS 1.3 method [RFC9190] utilizing the attestation extensions [I-D.fossati-tls-attestation]) or an onboarding protocol like FIDO Device Onboard (FDO) [FDO] or Bootstrapping Remote Secure Key Infrastructure (BRSKI) [RFC8995].

- * The MUD Manager, acting as a Relying Party, relays the Evidence to the Verifier and receives an Attestation Result in response. This allows the MUD Manager to check that the device is operating with the expected version of software and configuration.
- * Since a URL to the manifest is contained in the Evidence, the MUD Manager can look up the corresponding manifest.
- * The MUD Manager acquires the MUD file from the MUD URL found in the SUIT manifest. The SUIT manifest contains the MUD URL and not the MUD file primarily to due the size of the MUD file. This also allows the MUD file to be updated rapidly in response to evolving threats.
- * The MUD Manager verifies the MUD file signature using the Subject Key Identifier (SKI) provided in the SUIT manifest.
- * Then, the MUD Manager can apply any appropriate policy as described by the MUD file.

Each time a device is updated, rebooted, or otherwise substantially changed, it will execute the remote attestation procedures again.

4. Operational Considerations

This specification assumes that the software/firmware author provides a MUD file that describes the behavior of the software running on a device.

4.1. Pros

The approach described in this document has several advantages over the RFC 8520 MUD URL reporting mechanisms:

- * The MUD URL is tightly coupled to device software/firmware version.
- * The device does not report the MUD URL, so the device cannot tamper with the MUD URL.

- * The author explicitly authorizes a key to sign MUD files, providing a tight coupling between the party that knows device behavior best (the author of the software/firmware) and the party that declares device behavior (MUD file signer).
- * Network operators do not need to know, a priori, which MUD URL to use for each device; this can be harvested from the device's manifest and only replaced if necessary.
- * A network operator can still replace a MUD URL in a SUIT manifest:
 - By providing a SUIT manifest that overrides the MUD URL.
 - By replacing the MUD URL in their network infrastructure.
- * Devices can be quarantined if the Attestation Result indicates that an out-dated or compromised software/firmware version has been used.
- * Devices cannot lie about which MUD URL to use.

4.2. Cons

This mechanism relies on the use of SUIT manifests to encode the MUD URL. Conceptually, the MUD file is similar to a Software Bill of Material (SBOM) but focuses on the external visible communication behavior, which is essential for network operators, rather than describing the software libraries contained within the device itself.

- * MUD Manager must be aware of the Status Tracker or vice versa so that the MUD Manager can obtain MUD URLs and MUD Signer SKIs from the Status Tracker. This implies a new API in the MUD manager or Status Tracker.
- * The MUD manager requires a failover mechanism to trigger the status tracker to obtain a copy of the SUIT Manifest in order to extract the MUD URL if it is not already aware of a device. This could be done, for example, as a part of an onboarding flow.
- * Attestation Evidence may convey the SUIT Manifest, in which case the Status Tracker becomes a Relying Party since it depends on Attestation Evidence. This workflow is expected, however.
- * This approach explicitly moves the decisions about device behaviour away from the Network Operator and towards the Manifest Author. While this is appropriate when the Manifest Author is trusted, not all IoT devices are fully trusted, and MUD files enable a Network Operator to restrict their capabilities. For a

Network Operator to override a device's manufacturer-provided MUD URL will require the MUD manager to have a mechanism to enable this override, which adds complexity

5. Extensions to SUIT

To enable strong assertions about the network access requirements that a device should have for a particular software/configuration pair a MUD URL is added to the SUIT manifest along with a subject key identifier (ski). Note that the subject key identifier refers to a more generic version of SubjectPublicKeyInfo defined in [RFC5280], which refers to an X.509-based ski. The subject key identifier MUST be generated according to the process defined in [I-D.ietf-cose-key-thumbprint] and the SUIT_Digest structure MUST be populated with the selected hash algorithm and obtained fingerprint. The subject key identifier corresponds to the key used in the MUD signature file described in Section 13.2 of [RFC8520].

Note: A key need not be in COSE Key format to create a COSE Key Thumbprint of it.

The following Concise Data Definition Language (CDDL) [RFC8610] describes the extension to the SUIT_Manifest structure:

The extension to the SUIT_Manifest is described here:

```
$$unseverable-manifest-member-extensions //= (
  suit-manifest-mud => bstr .cbor SUIT_MUD_container
)
```

The SUIT_MUD_container structure is defined as follows:

```
SUIT_MUD_container = {
  suit-mud-url => #6.32(tstr),
  suit-mud-ski => SUIT_Digest,
}
```

6. Security Considerations

This specification links MUD files to SUIT manifests for improving security protection and ease of use. By including MUD URLs in SUIT manifests an extra layer of protection has been created and synchronization risks can be minimized.

Used in this way, the MUD manager presents an additional layer of security on networks where they are enabled. The MUD manager configures the L2/L3 infrastructure of a Local Area Network to apply restrictive policies to certain devices. The MUD manager only has

the ability to elevate or restrict the network privileges of a device. Therefore, attacks on the MUD Manager cannot compromise devices, they can only enable a compromised device to access more of the network. Further security considerations related to the MUD Manager are covered in [RFC8520].

If the MUD file and the software/firmware loaded onto the device gets out-of-sync a device may be firewalled and, with firewalling by networks in place, the device may stop functioning. This is, however, not a concern specific to this specification but rather to the use of MUD in general. Below are two mitigations:

- * A manufacturer must update the MUD file in advance of network service or product changes so that the new services can be supported. Because the MUD file is accessed by a URL means that it can be subsequently updated. This requires a MUD file being retrieved again. This handles the case when the device is already deployed and in use.
- * There is a possibility that an IoT device has remained on-shelf inventory for an extended period, resulting in its MUD file being inaccessible at its previous location. This necessitates a decision on how to implement a fail-safe tailored to the particular environment.

7. IANA Considerations

IANA is requested to add a new value to the SUIT manifest elements registry created with [I-D.ietf-suit-manifest]:

- * Label: TBD1 [[Value allocated from the standards action address range]]
- * Name: Manufacturer Usage Description (MUD)
- * Reference: [[TBD: This document]]

8. References

8.1. Normative References

[I-D.ietf-cose-key-thumbprint]
Isobe, K., Tschofenig, H., and O. Steele, "CBOR Object Signing and Encryption (COSE) Key Thumbprint", Work in Progress, Internet-Draft, draft-ietf-cose-key-thumbprint-06, 6 September 2024, <<https://datatracker.ietf.org/doc/html/draft-ietf-cose-key-thumbprint-06>>.

- [I-D.ietf-rats-eat]
Lundblade, L., Mandyam, G., O'Donoghue, J., and C. Wallace, "The Entity Attestation Token (EAT)", Work in Progress, Internet-Draft, draft-ietf-rats-eat-31, 6 September 2024, <<https://datatracker.ietf.org/doc/html/draft-ietf-rats-eat-31>>.
- [I-D.ietf-suit-manifest]
Moran, B., Tschofenig, H., Birkholz, H., Zandberg, K., and O. R 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-33, 24 February 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-suit-manifest-33>>.
- [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>>.
- [RFC8520] Lear, E., Droms, R., and D. Romascanu, "Manufacturer Usage Description Specification", RFC 8520, DOI 10.17487/RFC8520, March 2019, <<https://www.rfc-editor.org/rfc/rfc8520>>.
- [RFC8610] Birkholz, H., Vigan , 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>>.
- [RFC9334] Birkholz, H., Thaler, D., Richardson, M., Smith, N., and W. Pan, "Remote ATtestation procedures (RATS) Architecture", RFC 9334, DOI 10.17487/RFC9334, January 2023, <<https://www.rfc-editor.org/rfc/rfc9334>>.

8.2. Informative References

- [FIDO] FIDO Alliance, "FIDO Device Onboard Specification 1.1", April 2022, <<https://fidoalliance.org/specifications/download-iot-specifications/>>.

[I-D.fossati-tls-attestation]

Tschofenig, H., Sheffer, Y., Howard, P., Mihalcea, I., Deshpande, Y., Niemi, A., and T. Fossati, "Using Attestation in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", Work in Progress, Internet-Draft, draft-fossati-tls-attestation-08, 21 October 2024, <<https://datatracker.ietf.org/doc/html/draft-fossati-tls-attestation-08>>.

[I-D.ietf-opsawg-mud-acceptable-urls]

Richardson, M., Pan, W., and E. Lear, "Authorized update to MUD URLs", Work in Progress, Internet-Draft, draft-ietf-opsawg-mud-acceptable-urls-12, 6 September 2024, <<https://datatracker.ietf.org/doc/html/draft-ietf-opsawg-mud-acceptable-urls-12>>.

[RFC5280]

Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/rfc/rfc5280>>.

[RFC8995]

Pritikin, M., Richardson, M., Eckert, T., Behringer, M., and K. Watsen, "Bootstrapping Remote Secure Key Infrastructure (BRSKI)", RFC 8995, DOI 10.17487/RFC8995, May 2021, <<https://www.rfc-editor.org/rfc/rfc8995>>.

[RFC9190]

PreuÅ\237 Mattsson, J. and M. Sethi, "EAP-TLS 1.3: Using the Extensible Authentication Protocol with TLS 1.3", RFC 9190, DOI 10.17487/RFC9190, February 2022, <<https://www.rfc-editor.org/rfc/rfc9190>>.

Acknowledgements

We would like to thank Roman Danyliw for his excellent review as the responsible security area director, Bahcet Sarikaya for his Genart review, Michael Richardson for his IoT directorate review and Susan Hares for her Opsdir review. During the IESG review Robert Wilton, Eliot Lear, Zaheduzzaman Sarker, Francesca Palombini, John Scudder, Paul Wouters, Å\211ric Vyncke, and Murray Kucherawy.

Authors' Addresses

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

Hannes Tschofenig
Email: hannes.tschofenig@gmx.net

SUIT
Internet-Draft
Intended status: Standards Track
Expires: 27 November 2026

B. Moran
Arm Limited
H. Birkholz
Fraunhofer SIT
26 May 2026

Secure Reporting of SUIT Update Status
draft-ietf-suit-report-20

Abstract

The Software Update for the Internet of Things (SUIT) manifest provides a way for many different update and boot workflows to be described by a common format. This document specifies a lightweight feedback mechanism that allows a developer in possession of a manifest to reconstruct the decisions made and actions performed by a manifest processor.

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 27 November 2026.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1.	Introduction	2
2.	Conventions and Terminology	3
3.	The SUIT_Record	4
4.	The SUIT_Report	7
4.1.	suit-report-records	9
4.2.	suit-report-result	10
5.	Attestation	11
6.	Capability Reporting	14
7.	EAT Claim	15
8.	SUIT_Report Container	15
9.	IANA Considerations	17
9.1.	Expert Review Instructions	18
9.2.	Media Type Registration	18
9.2.1.	application/suit-report+cose	18
9.3.	CoAP Content-Format Registration	19
9.4.	CBOR Tag Registration	19
9.5.	SUIT_Report Elements	20
9.6.	SUIT_Record Elements	21
9.7.	SUIT_Report Reasons	21
9.8.	SUIT Capability Report Elements	22
10.	Security Considerations	24
11.	Acknowledgements	25
12.	References	25
12.1.	Normative References	25
12.2.	Informative References	27
Appendix A.	Full CDDL	27
Authors' Addresses		31

1. Introduction

This document specifies a logging container, specific to Software Update for the Internet of Things (SUIT) Manifests ([I-D.ietf-suit-manifest]) that creates a lightweight feedback mechanism for developers in the event that an update or boot fails in the manifest processor. In this way, it provides the necessary link between the Status Tracker Client and the Status Tracker Server as defined in Section 2.3 of [RFC9019].

A SUIT Manifest Processor can fail to install or boot an update for many reasons. Frequently, the error codes generated by such systems fail to provide developers with enough information to find root causes and produce corrective actions, resulting in extra effort to reproduce failures. Logging the results of each SUIT command can simplify this process.

While it is possible to report the results of SUIT commands through existing logging or attestation mechanisms, this comes with several drawbacks:

- * data inflation, particularly when designed for text-based logging
- * missing information elements
- * missing support for multiple components

The CBOR objects defined in this document allow devices to:

- * report a trace of how an update was performed
- * report expected vs. actual values for critical checks
- * describe the installation of complex multi-component architectures
- * describe the measured properties of a system
- * report the exact reason for a parsing failure

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 terms "Author", "Recipient", and "Manifest" are defined in Section 2 of [I-D.ietf-suit-manifest].

Additionally, this document uses the term `Boot`: initialization of an executable image. Although this document refers to `boot`, any boot-specific operations described are equally applicable to starting an executable in an OS context.

The term "Reporting Engine" is used as a conceptual function that collects reporting information from a Manifest Processor and decides whether to emit report output. A Reporting Engine is not required to

be a separate implementation component. Implementations may expose this function through one or more internal APIs. This document specifies the report format and externally observable semantics; it does not specify a mandatory local API.

3. The SUIT_Record

The SUIT_Record is a record of a decision taken by the Manifest Processor. It contains the information that the Manifest Processor used to make the decision. The decision can be inferred from this information, so it is not included. If the developer has a copy of the manifest, then they need little information to reconstruct what the manifest processor has done. They need any data that influences the control flow of the manifest. The manifest only supports the following control flow primitives:

- * Set Component
- * Set/Override Parameters
- * Try-Each
- * Run Sequence
- * Conditions

Of these, only conditions change the behavior of the processor from the default, and then only when the condition fails.

To reconstruct the flow of a manifest, a developer needs a list of metadata about failed conditions:

- * the current manifest
- * the current Command Sequence (Section 5.3.3 of [I-D.ietf-suit-manifest])
- * the offset into the current Command Sequence
- * the current component index
- * the "reason" for failure

Most conditions compare a parameter to an actual value, so the "reason" is typically the actual value.

Since it is possible that a non-condition command (directive) may fail in an exceptional circumstance, a failure code for a non-condition command must be communicated to the developer. However, a failed directive will terminate processing of the manifest. To accommodate for a failed command and for explicit "completion," an additional "result" element is included as well; however, this is included in the SUIT_Report (Section 4). In the case of a command failure, the failure reason is typically a numeric error code. However, these error codes need to be standardised in order to be useful.

This approach effectively compacts the log of operations taken using the SUIT Manifest as a dictionary. This enables a full reconstruction of the log using a matching decompaction tool. The following CDDL ([RFC8610]) shows the structure of a SUIT_Record.

```
SUIT_Record = [
    suit-record-manifest-id      : [* uint ],
    suit-record-manifest-section : int,
    suit-record-section-offset   : uint,
    suit-record-component-index  : uint,
    suit-record-properties       : {*$SUIT_Parameters},
    * $$SUIT_Record_Extensions
]
```

suit-record-manifest-id identifies the manifest whose execution produced the record. The manifest id is a list of integers that form a walk of the manifest dependency tree, starting at the root manifest. An empty list indicates the root manifest. If the list is not empty, each integer is the Component Index, as defined for Set Component Index in Section 8.4.10.1 of [I-D.ietf-suit-manifest], that selected a dependency manifest from the manifest identified by the preceding path elements. The final integer identifies the dependency manifest whose execution produced the record.

suit-record-manifest-id and suit-record-component-index therefore refer to different index domains. Each element of suit-record-manifest-id is a Component Index that identifies a dependency envelope as a component of the manifest at the previous level of the dependency tree. suit-record-component-index is the current Component Index within the manifest identified by suit-record-manifest-id.

For example, suppose that the root manifest has 3 dependency components and each of those dependency manifests has 2 dependency components of its own:

```
* Root
```

- Dependency A (Component Index 0 in Root)
 - o Dependency AA (Component Index 0 in Dependency A)
 - o Dependency AB (Component Index 1 in Dependency A)
- Dependency B (Component Index 1 in Root)
 - o Dependency BA (Component Index 0 in Dependency B)
 - o Dependency BB (Component Index 1 in Dependency B)
- Dependency C (Component Index 2 in Root)
 - o Dependency CA (Component Index 0 in Dependency C)
 - o Dependency CB (Component Index 1 in Dependency C)

A `suit-record-manifest-id` of [1,0] would indicate that the record was produced while executing Dependency BA. Similarly, a `suit-record-manifest-id` of [2,1] would indicate Dependency CB.

`suit-record-manifest-section` indicates which Command Sequence of the manifest was active. Only the "top level" Command Sequences, with entries in the Manifest are identified by this element. These are:

- * `suit-validate` = 7
- * `suit-load` = 8
- * `suit-invoke` = 9
- * `suit-dependency-resolution` = 15
- * `suit-payload-fetch` = 16
- * `suit-candidate-verification` = 18
- * `suit-install` = 20

This list may be extended through extensions to the `SUIT_Manifest`.

`suit-record-manifest-section` is used in addition to an offset so that the developer can index into severable Command Sequences in a predictable way. The value of this element is the value of the key that identified the Command Sequence in the manifest.

suit-record-section-offset is the number of bytes into the current Command Sequence at which the current command is located.

suit-record-component-index is the index of the component that was specified at the time that the report was generated. This field is necessary due to the availability of set-current-component values of True and a list of components. Both of these values cause the manifest processor to loop over commands using a series of component-ids, so the developer needs to know which was selected when the command executed.

suit-record-properties contains any measured properties that led to the command failure. For example, this could be the actual value of a SUIT_Digest or class identifier. This is encoded as a map whose entries are drawn from the \$\$SUIT_Parameters group defined in Section 8.4.8 of [I-D.ietf-suit-manifest].

4. The SUIT_Report

The SUIT_Report is a SUIT-specific logging container. It contains the SUIT_Records needed to reconstruct the decisions made by a Manifest Processor as well as references to the Manifest being processed, the result of processing, and an optional capability report.

Some metadata is common to all records, such as the root manifest: the manifest that is the entry-point for the manifest processor. This metadata is aggregated with a list of SUIT_Records as defined in Section 3. The SUIT_Report may also contain a list of any System Properties that were measured and reported, and a reason for a failure if one occurred. The following CDDL describes the structure of a SUIT_Report and a SUIT_Reference:

```
SUIT_Report = {
  suit-reference           => SUIT_Reference,
  ? suit-report-nonce     => bstr,
  suit-report-records     => [
    * SUIT_Record / system-property-claims ],
  suit-report-result      => true / {
    suit-report-result-code => int,
    suit-report-result-record => SUIT_Record,
    suit-report-result-reason => SUIT_Report_Reasons,
  },
  ? suit-report-capability-report => SUIT_Capability_Report,
  * $$SUIT_Report_Extensions
}

system-property-claims = {
  system-component-id => SUIT_Component_Identifier,
  + $$SUIT_Parameters,
}

SUIT_Reference = [
  suit-report-manifest-uri : tstr,
  suit-report-manifest-digest : SUIT_Digest
]
```

Further details for each element appear in subsequent sections: the encoding of `suit-report-records` is defined in Section 4.1, the result semantics in Section 4.2, and the optional capability report in Section 6.

The `suit-reference` provides a reference URI and digest for a suit manifest. The URI **MUST** exactly match the `suit-reference-uri` (Section 8.4.3 of [I-D.ietf-suit-manifest]) that is provided in the manifest. The digest is the digest of the manifest, exactly as reported in `SUIT_Authentication`, element 0 (Section 8.3 of [I-D.ietf-suit-manifest]).

A recipient of a `SUIT_Report` cannot be assumed to already possess the `SUIT_Manifest` needed to interpret it. `SUIT_Records` identify locations in a manifest and contain Component Indices, so they are meaningful only when the matching manifest can be obtained and validated using `suit-report-manifest-uri` and `suit-report-manifest-digest`. In contrast, `system-property-claims` contain a `SUIT_Component_Identifier` rather than only a Component Index. A recipient **MAY** process `system-property-claims` without first obtaining the matching `SUIT_Manifest`, but **MUST NOT** use `SUIT_Records` for manifest reconstruction unless the matching `SUIT_Manifest` is available.

NOTE: The digest is used in preference to other identifiers in the manifest because it allows a manifest to be uniquely identified (collision resistance) whereas other identifiers, such as the sequence number, can collide, particularly in scenarios with multiple trusted signers.

suit-report-nonce provides a container for freshness or replay protection information. This field MAY be omitted where the suit-report is authenticated within a container that provides freshness already. For example, attestation evidence typically contains a proof of freshness.

suit-report-manifest-digest provides a SUIT_Digest (as defined in Section 10 of [I-D.ietf-suit-manifest]) that is the characteristic digest of the Root manifest. This digest MUST be the same digest as is held in the first element of SUIT_Authentication in the referenced Manifest_Envelope.

suit-report-manifest-uri provides the reference URI that was provided in the root manifest.

4.1. suit-report-records

suit-report-records is a list of 0 or more SUIT_Records or system-property-claims. Because SUIT_Records are only generated on failure, in simple cases this can be an empty list. SUIT_Records and suit-system-property-claims are merged into a single list because this reduces the overhead for a constrained node that generates this report. The use of a single log allows report generators to use simple memory management. Because the system-property-claims are encoded as maps and SUIT_Records are encoded as lists, a recipient need only filter the CBOR Type-5 entries from suit-report-records to obtain all system-property-claims.

System Properties can be extracted from suit-report-records by filtering suit-report-records for maps. System Properties are a list of measured or asserted properties of the system that creates the SUIT_Report. These properties are scoped by component identifier. Because this list is expected to be constructed on the fly by a constrained node, component identifiers may appear more than once. A recipient may convert the result to a more conventional structure:

```
SUIT_Record_System_Properties = {
  * component-id => {
    + $$SUIT_Parameters,
  }
}
```

4.2. suit-report-result

suit-report-result provides a mechanism to show that the SUIT procedure completed successfully (value is true) or why it failed (value is a map of an error code and a SUIT_Record).

suit-report-result-reason gives a high-level explanation of the failure. These reasons are intended for interoperable implementations. The reasons are divided into a small number of groups:

- * suit-report-reason-cbor-parse: a parsing error was encountered by the CBOR parser.
- * suit-report-reason-cose-unsupported: an unsupported COSE ([RFC9052]) structure or header was encountered.
- * suit-report-reason-alg-unsupported: an unsupported COSE algorithm was encountered.
- * suit-report-reason-unauthorised: Signature/MAC verification failed.
- * suit-report-reason-command-unsupported: an unsupported command was encountered.
- * suit-report-reason-component-unsupported: The manifest declared a component/prefix that does not exist.
- * suit-report-reason-component-unauthorised: The manifest declared a component that is not accessible by the signer.
- * suit-report-reason-parameter-unsupported: The manifest used a parameter that does not exist.
- * suit-report-reason-severing-unsupported: The manifest used severable fields but the Manifest Processor does not support them.
- * suit-report-reason-condition-failed: A condition failed with soft-failure off.
- * suit-report-reason-operation-failed: A command failed (e.g., download/copy/swap/write)
- * suit-report-reason-invoke-pending: Invocation is about to be attempted and the final outcome is not yet known.

The `suit-report-result-code` reports an internal error code that is provided for debugging reasons. This code is not intended for interoperability.

The `suit-report-result-record` indicates the exact point in the manifest or manifest dependency tree where the error occurred.

NOTE: Some deployments use `SUIT_Command_Invoke`, which can transfer control to invoked code that never returns to the Manifest Processor. When a `SUIT_Report` is produced for remote attestation, implementations often need to sign the report before attempting the invoke. Signing with an unconditional "success" result would be misleading if the invocation ultimately fails. Implementers can leave the invoke outcome implicit²⁴ allowing a verifier to infer that execution was handed off²⁴ or, when the result must be reported before invocation, use `suit-report-reason-invoke-pending` to signal that invocation is about to occur without asserting a final outcome.

`suit-report-capability-report` provides a mechanism to report the capabilities of the Manifest Processor. The `SUIT_Capability_Report` is described in Section 6. The capability report is optional to include in the `SUIT_Report`, according to an application-specific policy. While the `SUIT_Capability_Report` is not expected to be very large, applications should ensure that they only report capabilities when necessary in order to conserve bandwidth. A capability report is not necessary except when:

1. A client explicitly requests the capability report, or
2. A manifest attempts to use a capability that the Manifest Processor does not implement.

5. Attestation

Where Remote Attestation (see [RFC9334], the RATS Architecture) is in use, the RATS Verifier (Verifier hereafter) requires a set of Attestation Evidence. Attestation Evidence contains Evidence Claims about the Attester. These Evidence Claims contain measurements about the Attester. Many of these measurements are the same measurements that are generated in SUIT, which means that a `SUIT_Report` contains most of the Claims and some of the Endorsements that a Verifier requires.

Using a `SUIT_Manifest` and a `SUIT_Report` improves a Verifier's ability to appraise the trustworthiness of a remote device. Remote attestation is done by using the `SUIT_Envelope` along with the `SUIT_Report` in Evidence to reconstruct the state of the device at boot time. Additionally, by including `SUIT_Report` data as telemetry

(i.e., debug/failure information) next to measurements in Evidence, both types of Evidence data can be notarized via verifiable data structure, such as an append-only log (Section 3 of [I-D.ietf-scitt-architecture]) using the same conceptual message.

For the SUIT_Report to be usable as Attestation Evidence, the environment that generated the SUIT_Report also needs to be measured. Typically, this means that the software that executes the commands in the Manifest (the Manifest Processor) must be measured; similarly, the piece of software that assembles the measurements, taken by the Manifest Processor, into the SUIT_Report (the Report Generator) must also be measured. Any bootloaders or operating systems that facilitate the running of the Manifest Processor or Report Generator also need to be measured in order to demonstrate the integrity of the measuring environment.

Therefore, if a Remote Attestation format that conveys Attestation Evidence, such as an Entity Attestation Token (EAT, see [RFC9711]), contains a SUIT_Report, then it MUST also include an integrity measurement of the Manifest Processor, the Report Generator and any bootloader or OS environment that ran before or during the execution of both.

If Reference Values (Section 8.3 of [RFC9334]) required by the Verifier are delivered in a SUIT_Envelope, this codifies the delivery of appraisal information to the Verifier:

- * The Firmware Distributor:
 - sends the SUIT_Envelope to the Verifier without payload or text, but with Reference Values
 - sends the SUIT_Envelope to the Recipient without Reference Values, or text, but with payload
- * The Recipient:
 - Installs the firmware as described in the SUIT_Manifest and generates a SUIT_Report, which is encapsulated in an EAT by the installer and sent to the Firmware Distributor.
 - Boots the firmware as described in the SUIT_Manifest and creates a SUIT_Report, which is encapsulated in an EAT by the installer and sent to the Firmware Distributor.
- * The Firmware Distributor sends both reports to the Verifier (separately or together)

* The Verifier:

- Reconstructs the state of the device using the manifest
- Compares this state to the Reference Values
- Returns an Attestation Report to the Firmware Distributor

This approach simplifies the design of the bootloader since it is able to use an append-only log. It allows a Verifier to validate this report against signed Reference Values that is provided by the firmware author, which simplifies the delivery chain of verification information to the Verifier.

For a Verifier to consume the `SUIT_Report`, it requires a copy of the `SUIT_Manifest`. The Verifier then replays the `SUIT_Manifest`, using the `SUIT_Report` to resolve whether each condition is met. It identifies each measurement that is required by attestation policy and records this measurement as a Claim (Section 4 of [RFC9711]). It evaluates whether the `SUIT_Report` correctly matches the `SUIT_Manifest` as an element of evaluating trustworthiness. For example there are several indicators that would show that a `SUIT_Report` does not match a `SUIT_Manifest`. If any of the following (not an exhaustive list) occur, then the Manifest Processor that created the report is not trustworthy:

- * Hash of `SUIT_Manifest` at `suit-report-manifest-uri` does not match `suit-report-manifest-digest`
- * A `SUIT_Record` is issued for a `SUIT_Command_Sequence` that does not exist in the `SUIT_Manifest` at `suit-report-manifest-uri`.
- * A `SUIT_Record` is identified at an offset that is not a condition and does not have a reporting policy that would indicate a `SUIT_Record` is needed.

Many architectures require multiple Verifiers, for example where one Verifier handles hardware trust, and another handles software trust, especially the evaluation of software authenticity and freshness. Some Verifiers may not be capable of processing a `SUIT_Report` and, for separation of roles, it may be preferable to divide that responsibility. In this case, the Verifier of the `SUIT_Report` should perform an Evidence Transformation [I-D.ietf-rats-evidence-trans] and produce general purpose Measurement Results Claims that can be consumed by a downstream Verifier, for example a Verifying Relying Party, that does not understand `SUIT_Reports`.

6. Capability Reporting

Because SUIT is extensible, a manifest author must know what capabilities a device has available. To enable this, a capability report is a set of lists that define which commands, parameters, algorithms, and component IDs are supported by a manifest processor.

The CDDL for a `SUIT_Capability_Report` follows:

```
SUIT_Capability_Report = {
  suit-component-capabilities => [+ SUIT_Component_Capability]
  suit-command-capabilities   => [+ int],
  suit-parameters-capabilities => [+ int],
  suit-crypt-algo-capabilities => [+ int],
  ? suit-envelope-capabilities => [+ int],
  ? suit-manifest-capabilities => [+ int],
  ? suit-common-capabilities   => [+ int],
  ? suit-text-capabilities     => [+ int],
  ? suit-text-component-capabilities => [+ int],
  ? suit-dependency-capabilities => [+ int],
  * [+int]                    => [+ int],
  * $$SUIT_Capability_Report_Extensions
}
```

```
SUIT_Component_Capability = [*bstr,?true]
```

A `SUIT_Component_Capability` is similar to a `SUIT_Component_ID`, with one difference: it may optionally be terminated by a CBOR `'true'` which acts as a wild-card match for any component with a prefix matching the `SUIT_Component_Capability` leading up to the `'true.'` This feature is for use with filesystem storage, key value stores, or any other arbitrary-component-id storage systems.

When reporting capabilities, it is OPTIONAL to report capabilities that are declared mandatory by the SUIT Manifest [I-D.ietf-suit-manifest]. Capabilities defined by extensions MUST be reported.

Additional capability reporting can be added as follows: if a manifest element does not exist in this map, it can be added by specifying the CBOR path to the manifest element in an array and using this as the key. For example `SUIT_Dependencies`, as described in Section 5.2.2 of [I-D.ietf-suit-trust-domains], could have an extension added, which was key 3 in the `SUIT_Dependencies` map. This capability would be reported as: `[3, 3, 1] => [3]`, where the key consists of the key for `SUIT_Manifest` (3), the key for `SUIT_Common` (3), and the key for `SUIT_Dependencies` (1). Then the value indicates that this manifest processor supports the extension (3).

7. EAT Claim

The Entity Attestation Token (EAT, see [RFC9711]) is a secure container for conveying Attestation Evidence, such as measurements, and Attestation Results. The SUIT_Report is a form of measurement done by the SUIT Manifest Processor as it attempts to invoke a manifest or install a manifest. As a result, the SUIT_Report can be captured in an EAT measurements type.

The log-based structure of the SUIT_Report is not conducive to processing by a typical Relying Party: it contains only a list of waypoints through the SUIT Manifest--unless system parameter records are included--and requires additional information (the SUIT_Manifest) to reconstruct the values that must have been present at each test. A Verifier in possession of the SUIT_Manifest can reconstruct the measurements that would produce the waypoints in the SUIT_Report. The Verifier SHOULD convert a SUIT_Report into a more consumable version of the EAT claim by, for example, constructing a measurement results claim that contains the digest of a component, the Vendor ID and Class ID of a component, etc.

8. SUIT_Report Container

Transmission of the SUIT_Report MUST satisfy the requirements of Section 4.3.16 of [RFC9124]: REQ.SEC.REPORTING.

Status reports from the device to any remote system MUST be performed over an authenticated, confidential channel in order to prevent modification or spoofing of the reports.

As a result, the SUIT_Report MUST be transported using one of the following methods:

- * As part of a larger document that provides authenticity guarantees, such as within a measurements claim in an Entity Attestation Token (EAT, Section 4.2.16 of [RFC9711]).
- * As the payload of a message transmitted over a communication security protocol, such as DTLS [RFC9147].
- * Encapsulated within a secure container, such as a COSE structure. In the case of COSE, the container MUST provide authenticity and integrity using COSE_Sign1 or COSE_Mac0; confidentiality MAY be provided by carrying a COSE_Encrypt0 object as the authenticated payload. The SUIT_Report or encrypted SUIT_Report MUST be the sole payload, as illustrated by the CDDL fragment below.

```

SUIT_Report_Protected /= SUIT_Report_COSE_Sign1 \
    .and SUIT_COSE_Profiles
SUIT_Report_Protected /= SUIT_Report_COSE_Sign1_Tagged \
    .and SUIT_COSE_Profiles
SUIT_Report_Protected /= SUIT_Report_COSE_MAC0 \
    .and SUIT_COSE_Profiles
SUIT_Report_Protected /= SUIT_Report_COSE_MAC0_Tagged \
    .and SUIT_COSE_Profiles

SUIT_Report_COSE_Sign1_Tagged = #6.18(SUIT_Report_COSE_Sign1)
SUIT_Report_COSE_Sign1 = [
    protected : bstr,
    unprotected : { * int => any },
    payload : bstr .cbor SUIT_Report_Unprotected,
    signature : bstr
]
SUIT_Report_COSE_MAC0_Tagged = #6.17(SUIT_Report_COSE_MAC0)
SUIT_Report_COSE_MAC0 = [
    protected : bstr,
    unprotected : { * int => any },
    payload : bstr .cbor SUIT_Report_Unprotected,
    tag : bstr
]
SUIT_Report_Unprotected = SUIT_Report / SUIT_Report_COSE_Encrypt0
SUIT_Report_COSE_Encrypt0 = COSE_Encrypt0

```

Note that `SUIT_Report_COSE_Sign1` and `SUIT_Report_COSE_MAC0` MUST be combined with a `SUIT_COSE_Profiles` from [I-D.ietf-suit-mti] using the CDDL `.and` directive. The `SUIT_Report_COSE_Encrypt0` carries a ciphertext payload that MUST contain just the ciphertext obtained by encrypting the following CDDL:

```
SUIT_Report_plaintext = bstr .cbor SUIT_Report
```

`SUIT_COSE_Profiles`, which use AES-CTR encryption, are not integrity protected and authenticated. For this purpose, `SUIT_Report_Protected` defines authenticated containers with an encrypted payload.

If a report cannot be constructed or authenticated according to local policy, the recipient reports the failure through an implementation-specific local error or logging mechanism. A recipient MUST NOT emit an unauthenticated `SUIT_Report` when policy requires authenticated reports. If a partial `SUIT_Report` is emitted, it MUST satisfy the same authentication and integrity policy as any other `SUIT_Report` emitted by that recipient.

9. IANA Considerations

IANA is requested to rename the overall SUIT registry group (<https://www.iana.org/assignments/suit/suit.xhtml>) "Software Update for the Internet of Things (SUIT)".

IANA is requested to allocate a CBOR tag for each of the following items. Please see Section 9.4 for further details.

- * SUIT_Report_Protected
- * SUIT_Reference
- * SUIT_Capability_Report

IANA is requested to allocate a CoAP content-format [RFC7252] and a media-type for SUIT_Report Section 9.2. Please see Section 9.2 and Section 9.3 for further details.

IANA is also requested to add the following registries to the SUIT registry group (<https://www.iana.org/assignments/suit/suit.xhtml>).

- * SUIT_Report Elements Section 9.5
- * SUIT_Record Elements Section 9.6
- * SUIT_Report Reasons Section 9.7
- * SUIT_Capability_Report Elements Section 9.8

For each of these registries, registration policy is:

- * -256 to 255: Standards Action With Expert Review
- * -65536 to -257, 256 to 65535: Specification Required
- * -4294967296 to -65537, 65536 to 4294967295: First Come First Served

Requests in the Standards Action and Specification Required ranges MUST undergo designated expert review as described below; this guidance supplements the normal IANA processing for those policies.

9.1. Expert Review Instructions

The IANA registries established in this document allow values to be added based on expert review. This section gives some general guidelines for what the experts should be looking for, but they are being designated as experts for a reason, so they should be given substantial latitude.

Expert reviewers should take into consideration the following points:

- * Point squatting should be discouraged. Reviewers are encouraged to get sufficient information for registration requests to ensure that the usage is not going to duplicate one that is already registered, and that the point is likely to be used in deployments. The zones tagged as private use are intended for testing purposes and closed environments; code points in other ranges should not be assigned for testing.
- * Specifications are required for the standards track range of point assignment. Specifications should exist for all other ranges, but early assignment before a specification is available is considered to be permissible. When specifications are not provided, the description provided needs to have sufficient information to identify what the point is being used for.
- * Experts should take into account the expected usage of fields when approving point assignment. The fact that there is a range for standards track documents does not mean that a standards track document cannot have points assigned outside of that range. The length of the encoded value should be weighed against how many code points of that length are left, the size of device it will be used on, and the number of code points left that encode to that size.

9.2. Media Type Registration

9.2.1. application/suit-report+cose

IANA is requested to register application/suit-report+cose as a media type for the SUIT_Report.

Type name: application
Subtype name: suit-report+cose
Required parameters: n/a
Optional parameters: n/a
Encoding considerations: binary (CBOR)
Security considerations: Section 10 of RFCthis
Interoperability considerations: SUIT Reports are encoded as CBOR

and can be wrapped in COSE_Sign1 or COSE_Mac0, optionally carrying COSE_Encrypt0 as the authenticated payload. Receivers must be able to identify and process the COSE envelope used and support compatible COSE algorithms for validation or decryption. All versioning and structure are self-describing within the CBOR SUIT_Report format, and no media-type parameters are used for negotiation. Interoperability therefore depends on shared deployment profiles that specify the expected COSE protections and algorithms. Comprehension of SUIT_Records is dependent on obtaining a matching SUIT Manifest. System property claims can still be processed without the manifest because they contain SUIT_Component_Identifier values rather than only Component Indices.

Published specification: RFCthis
 Applications that use this media type: SUIT Manifest Processor, SUIT Manifest Distributor, SUIT Manifest Author, RATS Attesters, RATS Verifiers
 Fragment identifier considerations: The syntax and semantics of fragment identifiers are as specified for "application/cose".
 Person & email address to contact for further information: SUIT WG mailing list (suit@ietf.org)
 Intended usage: COMMON
 Restrictions on usage: none
 Author/Change controller: IETF
 Provisional registration: no

9.3. CoAP Content-Format Registration

IANA is requested to assign a CoAP Content-Format ID for the SUIT_Report media type in the "CoAP Content-Formats" registry, from the "IETF Review with Expert Review or IESG Approval with Expert Review" space (256..9999), within the "CoRE Parameters" registry group [RFC7252] [IANA.core-parameters]:

Content Type	Content Coding	ID	Reference
application/suit-report+cose		TBA	RFCthis

Table 1

9.4. CBOR Tag Registration

IANA is requested to allocate a tag in the "CBOR Tags" registry [IANA.cbor-tags], preferably in the Specification Required range:

Tag	Data Item	Semantics
TBA	array	SUIT_Report_Protected
TBA	array	SUIT_Reference
TBA	map	SUIT_Capability_Report

Table 2

9.5. SUIT_Report Elements

IANA is requested to create a new registry for SUIT_Report Elements.

Label	Name	CDDL Label	Reference
2	Nonce	suit-report-nonce	Section 4 of RFCthis
3	Records	suit-report-records	Section 4 of RFCthis
4	Result	suit-report-result	Section 4 of RFCthis
5	Result Code	suit-report-result-code	Section 4 of RFCthis
6	Result Record	suit-report-result-record	Section 4 of RFCthis
7	Result Reason	suit-report-result-reason	Section 4 of RFCthis
8	Capability Report	suit-report-capability-report	Section 4 of RFCthis
99	Reference	suit-reference	Section 4 of RFCthis

Table 3

9.6. SUIT_Record Elements

IANA is requested to create a new registry for SUIT_Record Elements.

Label	Name	CDDL Label	Reference
0	Manifest ID	suit-record-manifest-id	Section 3 of RFCthis
1	Manifest Section	suit-record-manifest-section	Section 3 of RFCthis
2	Section Offset	suit-record-section-offset	Section 3 of RFCthis
3	Component Index	suit-record-component-index	Section 3 of RFCthis
4	Record Properties	suit-record-properties	Section 3 of RFCthis

Table 4

9.7. SUIT_Report Reasons

IANA is requested to create a new registry for SUIT_Report Reasons.

Label	Name	CDDL Label	Reference
0	Result OK	suit-report-reason-ok	Section 4.2 of RFCthis
1	CBOR Parse Failure	suit-report-reason-cbor-parse	Section 4.2 of RFCthis
2	Unsupported COSE Structure or Header	suit-report-reason-cose-unsupported	Section 4.2 of RFCthis
3	Unsupported COSE Algorithm	suit-report-reason-alg-unsupported	Section 4.2 of RFCthis
4	Signature / MAC verification failed	suit-report-reason-unauthorised	Section 4.2 of RFCthis
5	Unsupported SUIT	suit-report-reason-	Section 4.2

	Command	command-unsupported	of RFCthis
6	Unsupported SUIT Component	suit-report-reason-component-unsupported	Section 4.2 of RFCthis
7	Unauthorized SUIT Component	suit-report-reason-component-unauthorised	Section 4.2 of RFCthis
8	Unsupported SUIT Parameter	suit-report-reason-parameter-unsupported	Section 4.2 of RFCthis
9	Severing Unsupported	suit-report-reason-severing-unsupported	Section 4.2 of RFCthis
10	Condition Failed	suit-report-reason-condition-failed	Section 4.2 of RFCthis
11	Operation Failed	suit-report-reason-operation-failed	Section 4.2 of RFCthis
12	Invocation Pending	suit-report-reason-invoke-pending	Section 4.2 of RFCthis

Table 5

9.8. SUIT Capability Report Elements

IANA is requested to create a new registry for SUIT Capability Report Elements.

Label	Name	CDDL Label	Reference
1	Components	suit-component-capabilities	Section 6 of RFCthis
2	Commands	suit-command-capabilities	Section 6 of RFCthis
3	Parameters	suit-parameters-capabilities	Section 6 of RFCthis
4	Cryptographic Algorithms	suit-crypt-algo-capabilities	Section 6 of RFCthis
5	Envelope Elements	suit-envelope-capabilities	Section 6 of RFCthis
6	Manifest Elements	suit-manifest-capabilities	Section 6 of RFCthis
7	Common Elements	suit-common-capabilities	Section 6 of RFCthis
8	Text Elements	suit-text-capabilities	Section 6 of RFCthis
9	Component Text Elements	suit-text-component-capabilities	Section 6 of RFCthis
10	Dependency Capabilities	suit-dependency-capabilities	Section 6 of RFCthis

Table 6

10. Security Considerations

The SUIT_Report serves four primary security objectives:

- * Validated Identity
- * Integrity
- * Replay protection
- * Confidentiality

The mechanisms for achieving these protections are outlined in Section 8.

Ideally, a SUIT_Report SHOULD be conveyed as part of a remote attestation procedure, such as embedding it in EAT tokens that represent RATS conceptual messages. This approach ensures that the SUIT_Report is cryptographically bound to the environment (hardware, software, or both) in which it was generated, thereby strengthening its authenticity.

A SUIT_Report may disclose sensitive information about the device on which it were produced. In such cases, the SUIT_Report MUST be encrypted, as specified in Section 8.

Furthermore, failure reports, particularly those involving cryptographic operations, can unintentionally reveal insights into system weaknesses or vulnerabilities. As such, SUIT_Reports SHOULD be encrypted whenever possible, to minimize the risk of information leakage.

In addition to these core security requirements, operational considerations must be taken into account. When a SUIT_Report is included within another protocol message (e.g., inside an encrypted EAT), care must be taken to avoid inadvertently leaking information and to uphold the principle of least privilege. For example, in many EAT-based remote attestation flows, the Verifier may not require the full SUIT_Report. Similarly, the Relying Party might not need access to it either.

To support least-privilege access, the SUIT_Report should be independently encrypted, even when the transport or enclosing token is also encrypted. This layered encryption ensures that only authorized entities can access the contents of the SUIT_Report.

In other scenarios, the EAT Verifier might require full access to a SUIT_Report. For example, the SUIT_Report must be accessible in its entirety for the EAT Verifier to extract or convert the SUIT_Report content into specific EAT claims, such as measres (Measurement Results). A typical case involves translating a successful suit-condition-image check into a digest-based claim within the EAT.

When applying cryptographic protection to the SUIT_Report, the same algorithm profile used for the corresponding SUIT manifest SHOULD be reused. The available algorithm profiles are detailed in [I-D.ietf-suit-mti]. If using the same profile is not feasible (e.g., due to constraints imposed by suit-sha256-hsslms-a256kw-a256ctr), then a profile offering comparable security strength SHOULD be selected—for instance, suit-sha256-esp256-ecdh-a128ctr.

In exceptional cases, if no suitable profile can be applied, the necessity of disabling a SUIT_Report functionality altogether might arise.

SUIT_Reports may expose information about the user to the Verifier, Firmware Distributor, or Manifest Author. Implementors MUST carefully consider user consent in the reporting system.

11. Acknowledgements

The authors would like to thank Dave Thaler for his feedback.

12. References

12.1. Normative References

[I-D.ietf-suit-manifest]

Moran, B., Tschofenig, H., Birkholz, H., Zandberg, K., and O. Rønningstad, "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>>.

[I-D.ietf-suit-mti]

Moran, B., Rønningstad, O., and A. Tsukamoto, "Cryptographic Algorithms for Internet of Things (IoT) Devices", Work in Progress, Internet-Draft, draft-ietf-suit-mti-23, 22 July 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-suit-mti-23>>.

- [IANA.cbor-tags]
IANA, "Concise Binary Object Representation (CBOR) Tags",
<<https://www.iana.org/assignments/cbor-tags>>.
- [IANA.core-parameters]
IANA, "Constrained RESTful Environments (CoRE)
Parameters",
<<https://www.iana.org/assignments/core-parameters>>.
- [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>>.
- [RFC8792] Watsen, K., Auerswald, E., Farrel, A., and Q. Wu,
"Handling Long Lines in Content of Internet-Drafts and
RFCs", RFC 8792, DOI 10.17487/RFC8792, June 2020,
<<https://www.rfc-editor.org/rfc/rfc8792>>.
- [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>>.
- [RFC9052] Schaad, J., "CBOR Object Signing and Encryption (COSE):
Structures and Process", STD 96, RFC 9052,
DOI 10.17487/RFC9052, August 2022,
<<https://www.rfc-editor.org/rfc/rfc9052>>.
- [RFC9334] Birkholz, H., Thaler, D., Richardson, M., Smith, N., and
W. Pan, "Remote Attestation procedureS (RATS)
Architecture", RFC 9334, DOI 10.17487/RFC9334, January
2023, <<https://www.rfc-editor.org/rfc/rfc9334>>.
- [RFC9711] Lundblade, L., Mandyam, G., O'Donoghue, J., and C.
Wallace, "The Entity Attestation Token (EAT)", RFC 9711,
DOI 10.17487/RFC9711, April 2025,
<<https://www.rfc-editor.org/rfc/rfc9711>>.

12.2. Informative References

- [I-D.ietf-rats-evidence-trans]
Damato, F., Draper, A., and N. Smith, "Evidence Transformations", Work in Progress, Internet-Draft, draft-ietf-rats-evidence-trans-02, 17 October 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-rats-evidence-trans-02>>.
- [I-D.ietf-scitt-architecture]
Birkholz, H., Delignat-Lavaud, A., Fournet, C., Deshpande, Y., and S. Lasker, "An Architecture for Trustworthy and Transparent Digital Supply Chains", Work in Progress, Internet-Draft, draft-ietf-scitt-architecture-22, 10 October 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-scitt-architecture-22>>.
- [I-D.ietf-suit-trust-domains]
Moran, B. and K. Takayama, "Software Update for the Internet of Things (SUIT) Manifest Extensions for Multiple Trust Domain", Work in Progress, Internet-Draft, draft-ietf-suit-trust-domains-12, 22 July 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-suit-trust-domains-12>>.
- [RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <<https://www.rfc-editor.org/rfc/rfc7252>>.
- [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>>.
- [RFC9147] Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", RFC 9147, DOI 10.17487/RFC9147, April 2022, <<https://www.rfc-editor.org/rfc/rfc9147>>.

Appendix A. Full CDDL

In order to create a valid SUIT_Report document the structure of the corresponding CBOR message MUST adhere to the following CDDL ([RFC8610]) data definition.

To be valid, the following CDDL MUST have the COSE CDDL appended to it. The COSE CDDL can be obtained by following the directions in Section 1.4 of [RFC9052]. It must also have the CDDL from [I-D.ietf-suit-mti] appended to it. This CDDL is line-wrapped per [RFC8792].

```
; NOTE: '\ ' line wrapping per RFC 8792
===== NOTE: '\ ' line wrapping per RFC 8792 =====

SUIT_Report_Tool_Tweak /= SUIT_start
SUIT_Report_Tool_Tweak /= SUIT_Report_Protected
SUIT_Report_Protected /= SUIT_COSE_tool_tweak

SUIT_Report_Protected /= SUIT_Report_COSE_Sign1 .and \
                          SUIT_COSE_Profiles
SUIT_Report_Protected /= SUIT_Report_COSE_Sign1_Tagged .and \
                          SUIT_COSE_Profiles
SUIT_Report_Protected /= SUIT_Report_COSE_MAC0 .and \
                          SUIT_COSE_Profiles
SUIT_Report_Protected /= SUIT_Report_COSE_MAC0_Tagged .and \
                          SUIT_COSE_Profiles

SUIT_Report_COSE_Sign1_Tagged = #6.18(SUIT_Report_COSE_Sign1)
SUIT_Report_COSE_Sign1 = [
  protected : bstr,
  unprotected : { * int => any },
  payload : bstr .cbor SUIT_Report_Unprotected,
  signature : bstr
]
SUIT_Report_COSE_MAC0_Tagged = #6.17(SUIT_Report_COSE_MAC0)
SUIT_Report_COSE_MAC0 = [
  protected : bstr,
  unprotected : { * int => any },
  payload : bstr .cbor SUIT_Report_Unprotected,
  tag : bstr
]
SUIT_Report_Unprotected = SUIT_Report / SUIT_Report_COSE_Encrypt0
SUIT_Report_COSE_Encrypt0 = COSE_Encrypt0

SUIT_Report = {
  suit-reference           => SUIT_Reference,
  ? suit-report-nonce     => bstr,
  suit-report-records     => [
    * SUIT_Record / system-property-claims ],
  suit-report-result      => true / {
    suit-report-result-code => int,
    suit-report-result-record => SUIT_Record,
    suit-report-result-reason => SUIT_Report_Reasons,
  }
```

```
    },
    ? suit-report-capability-report => SUIT_Capability_Report,
    * $$SUIT_Report_Extensions
}

SUIT_Reference = [
    suit-report-manifest-uri : tstr,
    suit-report-manifest-digest : SUIT_Digest
]

SUIT_Record = [
    suit-record-manifest-id          : [* uint ],
    suit-record-manifest-section     : int,
    suit-record-section-offset       : uint,
    suit-record-component-index      : uint,
    suit-record-properties           : {*$SUIT_Parameters},
    * $$SUIT_Record_Extensions
]

system-property-claims = {
    system-component-id => SUIT_Component_Identifier,
    + $$SUIT_Parameters,
}

SUIT_Capability_Report = {
    suit-component-capabilities => [+ SUIT_Component_Capability]
    suit-command-capabilities   => [+ int],
    suit-parameters-capabilities => [+ int],
    suit-crypt-algo-capabilities => [+ int],
    ? suit-envelope-capabilities => [+ int],
    ? suit-manifest-capabilities => [+ int],
    ? suit-common-capabilities   => [+ int],
    ? suit-text-capabilities     => [+ int],
    ? suit-text-component-capabilities => [+ int],
    ? suit-dependency-capabilities => [+ int],
    * [+int]                     => [+ int],
    * $$SUIT_Capability_Report_Extensions
}

SUIT_Component_Capability = [*bstr,?true]

suit-report-nonce = 2
suit-report-records = 3
suit-report-result = 4
suit-report-result-code = 5
suit-report-result-record = 6
suit-report-result-reason = 7
suit-report-capability-report = 8
```

```
suit-reference = 99

system-component-id = 0

suit-record-manifest-id = 0
suit-record-manifest-section = 1
suit-record-section-offset = 2
suit-record-component-index = 3
suit-record-properties = 4

SUIT_Report_Reasons /= suit-report-reason-ok
SUIT_Report_Reasons /= suit-report-reason-cbor-parse
SUIT_Report_Reasons /= suit-report-reason-cose-unsupported
SUIT_Report_Reasons /= suit-report-reason-alg-unsupported
SUIT_Report_Reasons /= suit-report-reason-unauthorised
SUIT_Report_Reasons /= suit-report-reason-command-unsupported
SUIT_Report_Reasons /= suit-report-reason-component-unsupported
SUIT_Report_Reasons /= suit-report-reason-component-unauthorised
SUIT_Report_Reasons /= suit-report-reason-parameter-unsupported
SUIT_Report_Reasons /= suit-report-reason-severing-unsupported
SUIT_Report_Reasons /= suit-report-reason-condition-failed
SUIT_Report_Reasons /= suit-report-reason-operation-failed
SUIT_Report_Reasons /= suit-report-reason-invoke-pending

suit-report-reason-ok = 0
suit-report-reason-cbor-parse = 1
suit-report-reason-cose-unsupported = 2
suit-report-reason-alg-unsupported = 3
suit-report-reason-unauthorised = 4
suit-report-reason-command-unsupported = 5
suit-report-reason-component-unsupported = 6
suit-report-reason-component-unauthorised = 7
suit-report-reason-parameter-unsupported = 8
suit-report-reason-severing-unsupported = 9
suit-report-reason-condition-failed = 10
suit-report-reason-operation-failed = 11
suit-report-reason-invoke-pending = 12

suit-component-capabilities = 1
suit-command-capabilities = 2
suit-parameters-capabilities = 3
suit-crypt-algo-capabilities = 4
suit-envelope-capabilities = 5
suit-manifest-capabilities = 6
suit-common-capabilities = 7
suit-text-capabilities = 8
suit-text-component-capabilities = 9
suit-dependency-capabilities = 10
```

Authors' Addresses

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

Henk Birkholz
Fraunhofer SIT
Rheinstrasse 75
64295 Darmstadt
Germany
Email: henk.birkholz@ietf.contact

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

Copyright (c) 2025 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1.	Introduction	3
2.	Conventions and Terminology	5
3.	Changes to SUIT Workflow Model	6
4.	Changes to Manifest Metadata Structure	6
5.	Dependencies	8
5.1.	Changes to Required Checks	8
5.2.	Changes to Manifest Structure	10
5.2.1.	Manifest Component ID	10
5.2.2.	SUIT_Dependencies Manifest Element	10
5.3.	Changes to Abstract Machine Description	12
5.4.	Processing Dependencies	13
5.4.1.	Multiple Manifest Processors	13
5.5.	Dependency Resolution	14
5.6.	Added and Modified Commands	14
5.6.1.	suit-directive-set-parameters	15
5.6.2.	suit-directive-process-dependency	16
5.6.3.	suit-condition-is-dependency	17
5.6.4.	suit-condition-dependency-integrity	17
5.6.5.	suit-directive-unlink	17
6.	Uninstall	18
7.	Staging and Installation	19
7.1.	suit-candidate-verification	19
8.	Creating Manifests	20
8.1.	Dependency Template	20
8.1.1.	Integrated Dependencies	21
8.2.	Encrypted Manifest Template	21
8.3.	Overriding Encryption Info Template	22
8.4.	Operating on Multiple Components	24
9.	IANA Considerations	25
9.1.	SUIT Envelope Elements	26
9.2.	SUIT Manifest Elements	26
9.3.	SUIT Common Elements	26
9.4.	SUIT Commands	26
10.	Security Considerations	27
11.	References	28
11.1.	Normative References	28
11.2.	Informative References	29

Appendix A. A. Full CDDL	30
Appendix B. B. Examples	31
B.1. Example 0: Process Dependency	32
B.2. Example 1: Integrated Dependency	36
Authors' Addresses	38

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 Manifest.
- * **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.

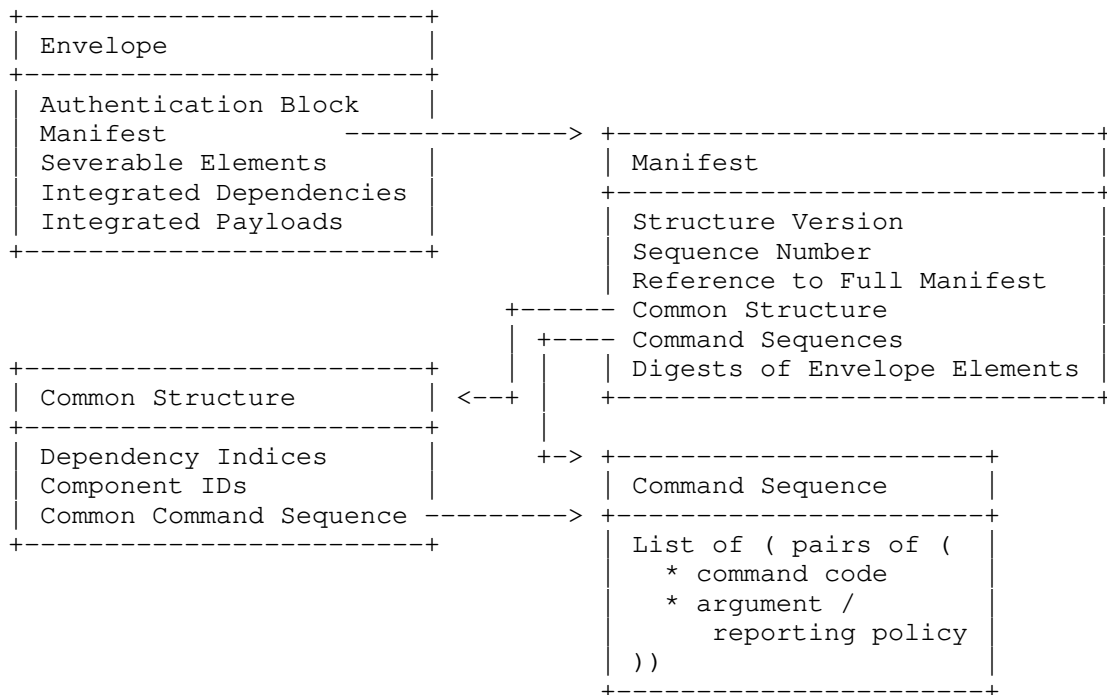


Figure 1: SUIT Metadata Structure

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 availability 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 overridden 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 Dependency'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 Sequence to the stored Command Sequence Offset

When identifying a Root Manifest's correct storage location, the Component Identifier MUST be evaluated vs. the access privileges 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. Rønningstad, "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., Keränen, 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'3F3E9A2CA98208FEAEAEAEADF7E1A0323
                      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

D86BA2025873825824822F5820AEB316A9A1E38253B29E6C99B60538368
B8AC8B5E6B9ACE1D239970830BBE62584AD28443A10128A0F658403F3E9A
2CA98208FEAEAEAEADF7E1A0323C97896ABFB79F91E8D0C1509B0A533CD0B
96BFC876A8F3B8ACE712FFF8EF7EA945E62A61E0BA5BD9929E4A1B47EC64
750358F9A70101020003581CA201A101A101814E646570656E64656E742E
7375697402818142313005814E646570656E64696E672E73756974095286
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
4F035842A6010102000347A102818142303005814E646570656E64656E74
2E73756974094D8414A11746636174203030170F14528414A1124B68656C
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
2E73756974094D8414A11746636174203030170F14528414A1124B68656C
6C6F20776F726C64120F'
})

```

Total size of Envelope with COSE authentication object: 519

Envelope with COSE authentication object:

```

D86BA3025873825824822F582088E1199580864EB1D1AD35EB5925BE68CA
565EE3BB39C27CDB31CEDA4DD667DF584AD28443A10128A0F65840074A36
1F7BBFA2ACF4EC3CFDAF4FDD8738414BAD672CAEA4F43607BE6031EA90CB
0C283A03C728608B0509C6FD2AFED40CFB0C3D341340830A00905E6A7298
900358BBA70101020003581CA201A101A101814E646570656E64656E742E
7375697402818142313005814E646570656E64696E672E73756974095286
0C0014A11749636174203030203130170F0F5844880C0114A3035824822F
58200F02CAF6D3E61920D36BF3CEA7F862A13BB8FB1F09C3F4C29B121FEA
B78EF3D80E18BE156F23646570656E64656E742E737569741502030F1458
288A0C010B000C0014A112581A20696E206D756C7469706C652074727573
7420646F6D61696E73120F6F23646570656E64656E742E7375697458BED8
6BA2025873825824822F58200F02CAF6D3E61920D36BF3CEA7F862A13BB8
FB1F09C3F4C29B121FEAB78EF3D8584AD28443A10128A0F65840A25F3371
26369D2E0B451C01DBD8CDB84A77E7F6C39E789DB3D227753494000C9D25
0001FDDCA39B4B4E3755A7278C11998171905F56C394CFBB907105DA804F
035842A6010102000347A102818142303005814E646570656E64656E742E
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

SUIT
Internet-Draft
Intended status: Standards Track
Expires: 27 November 2026

B. Moran
Arm Limited
K. Takayama
SECOM CO., LTD.
26 May 2026

Update Management Extensions for Software Updates for Internet of Things
(SUIT) Manifests
draft-ietf-suit-update-management-11

Abstract

This specification describes extensions to the SUIT manifest format. These extensions allow an update author, update distributor or device operator to more precisely control the distribution and installation of updates to devices. These extensions also provide a mechanism to inform a management system of Software Identifier and Software Bill Of Materials information about an updated device.

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 27 November 2026.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1.	Introduction	3
2.	Conventions and Terminology	3
3.	Extension Metadata	4
3.1.	suit-set-version	4
3.2.	suit-coswid	4
3.3.	suit-text-version-required	5
3.4.	text-current-version	6
4.	Extension Parameters	7
4.1.	suit-parameter-use-before	7
4.2.	suit-parameter-minimum-battery	7
4.3.	suit-parameter-update-priority	8
4.4.	suit-parameter-version	8
4.4.1.	suit-parameter-version Semantic Versioning encoding guidelines	9
4.5.	suit-parameter-wait-info	10
4.6.	suit-parameter-component-metadata	11
4.6.1.	Creator	12
4.6.2.	Creation & Modification Time	12
4.6.3.	Component Default Permissions	13
4.6.4.	User, Role, Group permissions	13
4.6.5.	File Type	14
5.	Extension Commands	15
5.1.	suit-condition-use-before	16
5.2.	suit-condition-image-not-match	17
5.3.	suit-condition-minimum-battery	17
5.4.	suit-condition-update-authorized	17
5.5.	suit-condition-version	17
5.6.	suit-directive-wait	17
5.7.	suit-directive-override-multiple	18
5.8.	suit-directive-copy-params	19
6.	IANA Considerations	19
6.1.	SUIT Envelope Elements	19
6.2.	SUIT Manifest Elements	19
6.3.	SUIT Commands	20
6.4.	SUIT Parameters	20
6.5.	SUIT Component Text Values	21
7.	Security Considerations	21

8. References	21
8.1. Normative References	21
8.2. Informative References	22
Appendix A. Full CDDL	22
Authors' Addresses	26

1. Introduction

Full management of software updates for unattended, connected devices requires a cooperation between the update author(s) and management, distribution, policy enforcement, and auditing systems. This specification provides the extensions to the SUIT manifest [I-D.ietf-suit-manifest] that enable an author to coordinate with these other systems. These extensions enable authors to instruct devices to examine update priority, local update authorisation, update lifetime, and system properties. They also enable devices to report and distributors to collect Software Bill of Materials (SBOM) information.

Extensions in this specification are OPTIONAL to implement and OPTIONAL to include in manifests. A Recipient that encounters a command or parameter it does not implement MUST reject the manifest as defined in [I-D.ietf-suit-manifest] Section 8.4.2, ensuring that update behaviour is never ambiguous. Conversely, when a deployment relies on update-management behaviour defined here, the manifest author MUST ensure that targeted recipients advertise support for the required extensions (for example via enablement policy or capability negotiation) before shipping such manifests so that required commands will be honoured rather than rejected.

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.

This draft makes use of terminology defined in [RFC9019] and [I-D.ietf-suit-manifest].

In addition, this document uses the following term:

- * _Primary cell_: a single-use battery chemistry that can be discharged but not recharged, making energy budgeting a one-way operation.

3. Extension Metadata

Some additional metadata makes management of SUIT updates easier:

- * A semantic version number for the update represented by the manifest
- * Concise Software Identifiers (CoSWID) [RFC9393]
- * Text descriptions of requirements
- * Text description of the current versions of components

3.1. `suit-set-version`

This metadata encodes a semantic version for the component set that the manifest updates, including any dependencies. This enables version comparisons to be performed on manifests. Non-manifest images encode their versions independently of the manifest.

Manifest Authors SHOULD encode `suit-set-version` whenever the release can be represented as a semantic version so that Recipients can compare manifests deterministically. The version MUST be encoded as a semantic version, according to [semver], to preserve that deterministic ordering. Deployments that cannot supply a semantic version without loss of fidelity MUST omit `suit-set-version` and convey any human-facing numbering via `text-current-version` (Section 3.4). Because `suit-set-version` is a machine-readable parameter for determining compatibility and because [semver] mandates that the build-number is ignored, build numbers MUST NOT be included.

The composition of `suit-set-version` is the same as `suit-parameter-version` (Section 4.4).

If a build number is desired, the manifest author MAY include it via `text-current-version` (Section 3.4).

3.2. `suit-coswid`

A CoSWID can enable Software Bill of Materials (SBOM) use-cases. Tightly coupling update and attestation ensures that verification infrastructure always knows what software to expect on each device.

suit-coswid is a member of the suit-manifest. It contains a Concise Software Identifier (CoSWID) as defined in [RFC9393]. This element SHOULD be made severable so that it can be discarded by the Recipient or an intermediary if it is not used by the Recipient while preserving the manifest signature. Implementations that cannot support severable elements MAY include suit-coswid non-severably, but MUST ensure that Recipients can still process the manifest.

suit-coswid is optional extension metadata and typically requires no processing by the Recipient. Recipients that do not understand or do not use optional extension metadata are not required to interpret the CoSWID content. A Recipient MUST NOT fail solely because a well-formed, policy-permitted suit-coswid field is present. A Recipient MAY still fail or reject the manifest when the suit-coswid field or its digest is malformed, when local policy rejects the metadata, when processing would exhaust available resources, when validation of processed CoSWID metadata fails, or when a manifest relies on unsupported critical behaviour. This requirement does not imply that every Recipient implements CoSWID processing.

suit-coswid is RECOMMENDED to implement and RECOMMENDED to include in manifests because management systems commonly need a durable software identity after update installation. CoSWID and related Software Bill of Materials metadata can support inventory, vulnerability management, compliance checks, and reconciliation between the installed update state and management-system records. This recommendation is scoped to the operational and security value of identifying installed software; it does not imply that the presence of SBOM metadata proves that the software is free of vulnerabilities or policy issues. Other extension metadata is not generally RECOMMENDED unless required by deployment policy or by a SUIT profile.

3.3. suit-text-version-required

suit-text-version-required is used to represent a version-based dependency on suit-parameter-version as described in Section 4.4 and Section 5.5. When a Manifest Author needs to communicate such a dependency to operators, the author SHOULD populate the suit-text map with a SUIT_Component_Identifier key for the dependency component, and place in the corresponding map a suit-text-version-required key with a free text expression that is representative of the version constraints placed on the dependency so that field personnel can validate compliance. Deployments that provide operator guidance exclusively through other channels MAY omit this field. This text SHOULD be expressive enough that a device operator can be expected to understand the dependency; predefined tokens MAY be used when supporting documentation ensures equivalent clarity. Expressions in

this field MUST be encoded as UTF-8 text limited to printable characters (Unicode general categories L, N, P, or Zs) and SHOULD use simple relational operators (for example >, >=, <, <=, =) so that automated tooling can perform lint checks. Implementations that render this text SHOULD escape or filter it to prevent markup or control-code injection. This is a free text field and there are no additional specific formatting rules beyond the requirements above.

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

```
[ 'x', 'y' ] : {  
  7 : ">=1.2.5,<2"  
}
```

3.4. text-current-version

suit-text-current-version is used to provide human-readable version information equivalent to suit-set-version (Section 3.1). This metadata MAY have a version listed for each or any component. The Manifest Processor MUST NOT consume this version; it is for human readability only.

To describe a version, a Manifest Author SHOULD populate the suit-text map with a SUIT_Component_Identifier key for the dependency component, and place in the corresponding map a suit-text-current-version key with a free text version that is representative of the version of the component so that operators can reconcile machine and human-readable records. Deployments that provide human-facing version information through other configuration channels MAY omit this text. This text SHOULD be expressive enough that a device operator can be expected to understand the version; environments that rely on catalog identifiers MAY use those identifiers when supporting documentation provides the necessary context. Values in this field MUST be encoded as UTF-8 text limited to printable characters, and implementations MUST treat suit-set-version and suit-parameter-version as authoritative when a discrepancy exists. Recipients MUST NOT interpret this text as executable code or markup and MUST treat it as display-only information. Implementations that render this text SHOULD sanitize, escape, or otherwise filter it before presentation. This is a free text field and there are no additional specific formatting rules beyond the requirements above.

It is RECOMMENDED that the Manifest Author use a Semantic Version ([semver]) in the free-text field to keep human-readable and machine-readable versions aligned. Unlike `suit-set-version` (Section 3.1), the full semantic version specification can be used.

4. Extension Parameters

Several parameters are needed to define the behaviour of the commands specified in Extension Commands (Section 5). These parameters follow the same considerations as defined in Section 8.4.8 of [I-D.ietf-suit-manifest].

Name	CDDL Structure	Reference
Use Before	<code>suit-parameter-use-before</code>	Section 4.1
Minimum Battery	<code>suit-parameter-minimum-battery</code>	Section 4.2
Update Priority	<code>suit-parameter-update-priority</code>	Section 4.3
Version	<code>suit-parameter-version</code>	Section 4.4
Wait Info	<code>suit-parameter-wait-info</code>	Section 4.5
Component Metadata	<code>suit-parameter-component-metadata</code>	Section 4.6

Table 1

4.1. `suit-parameter-use-before`

An expiry date for the use of the manifest encoded as the non-negative integer number of seconds since 1970-01-01. Implementations that use this parameter MUST use a 64-bit internal representation of the integer. Used with Section 5.1.

4.2. `suit-parameter-minimum-battery`

This parameter sets the minimum battery level in mWh. This parameter is encoded as a non-negative integer. Used with `suit-condition-minimum-battery` (Section 5.3).

4.3. `suit-parameter-update-priority`

This parameter sets the priority of the update. This parameter is encoded as an integer. It is used along with `suit-condition-update-authorized` (Section 5.4) to ask an application for permission to initiate an update. This does not constitute a privilege inversion because an explicit request for authorization has been provided by the Update Authority in the form of the `suit-condition-update-authorized` command.

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

4.4. `suit-parameter-version`

Indicates allowable versions for the specified component. One version comparison can be made with each `suit-parameter-version`. This parameter is compared with the version asserted by the current component when `suit-condition-version` (Section 5.5) is invoked. The current component can assert the current version in many ways, including storage in a parameter storage database, in a metadata object, or in a known location within the component itself.

Each `suit-parameter-version` contains a comparison operator and a version, according to the following CDDL:

```
SUIT_Parameter_Version_Match = [  
  suit-condition-version-comparison-type:  
    SUIT_Condition_Version_Comparison_Types,  
  suit-condition-version-comparison-value:  
    SUIT_Condition_Version_Comparison_Value  
]
```

The comparison type can be:

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

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

4.4.1. `suit-parameter-version` Semantic Versioning encoding guidelines

The encoded versions follow semantic versioning (see [semver]). Manifest Authors SHOULD keep their encoding aligned with Semantic Versioning so that Recipients can compare versions deterministically; if another numbering scheme is required, the sequence of integers encoded here MUST still preserve release ordering (for example, [2025,12,6] for a calendar-based release).

Versions are composed of:

1. A release version encoded as a sequence of 1 to 3 non-negative integers (allowing zero values as defined by [semver])
2. An optional pre-release indicator encoded as a negative integer, followed by zero or more non-negative integers

While [semver] allows a build number, it mandates that the build number is ignored. Because `suit-parameter-version` exists solely to enable the Manifest Processor to make a decision about version compatibility, build numbers MUST NOT be included.

In [semver],

1. The first integer represents the major number. This indicates breaking changes to the component.
2. The second integer represents the minor number. This is typically reserved for new features or large, non-breaking changes.
3. The third integer is the patch version. This is typically reserved for bug fixes.

The pre-release indicator MUST NOT appear as element 0. The pre-release indicator is encoded as:

- * -1: Release Candidate (RC)

- * -2: Beta
- * -3: Alpha

This allows these releases to compare correctly with final releases. For example, Version 2.0, RC1 is lower than Version 2.0.0 and higher than any Version 1.x. By encoding RC as -1, this works correctly: [2,0,-1,1] compares as lower than [2,0,0]. Similarly, beta (-2) is lower than RC and alpha (-3) is lower than RC.

For example:

- * 1.2.3 = [1,2,3].
- * 1.2-rc.3 = [1,2,-1,3].
- * 1.2-beta = [1,2,-2].
- * 1.2-alpha = [1,2,-3].
- * 1.2.3-alpha.4 = [1,2,3,-3,4].

4.5. suit-parameter-wait-info

suit-directive-wait (Section 5.6) directs the manifest processor to pause until a specified event occurs. The suit-parameter-wait-info encodes the parameters needed for the directive.

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

suit-parameter-wait-info is encoded as a map of wait events. All wait events MUST be satisfied before the Manifest Processor continues. The wait events currently defined are described in the following table.

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

Table 2

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

4.6. suit-parameter-component-metadata

In some instances, a system needs to know the file metadata for a component. This metadata can include:

- * creator
- * creation time
- * modification time

- * default permissions (rwx)
- * a map of user/permission pairs
- * a map of role/permission pairs
- * a map of group/permission pairs
- * file type

Unless otherwise stated, all string values in this structure MUST be encoded as UTF-8 without control characters (Unicode general categories Cc or Cf) and SHOULD be limited to human-readable identifiers such as names or POSIX-style paths. Binary values conveyed via bstr MUST be well-formed for the consuming platform (for example, a UUID or permissions bitmap) and MUST NOT exceed the minimum length required to represent the value canonically.

Component metadata is applied at time of fetch, copy, or write; see [I-D.ietf-suit-manifest], Sections 8.4.10.4, 8.4.10.5, and 8.4.10.6. Therefore, the component metadata parameter MUST be set in advance of the component being fetched, copied into, or written.

4.6.1. Creator

Sometimes, management of file systems requires that the creator of each file is correctly recorded. Because the default creator of files will be the update agent, this can obscure the actual creator of each file. The Creator metadata element allows overriding the default behaviour and setting the correct creator.

The creator is defined as follows:

```
SUIT_meta_actor_id = UUID_Tagged / bstr / tstr / int
UUID_Tagged = #6.37(bstr)
```

The actor ID can be whatever is most appropriate for any given system. For example, the actor ID might be a string (e.g., username), integer (e.g., POSIX userid), or UUID (e.g., TEEP TA UUID).

4.6.2. Creation & Modification Time

The creation and modification times are defined by CBOR time types. These are defined in [RFC8949], Section 3.4.2. The CBOR tag is REQUIRED when either creation or modification time are provided.

```
suit-meta-modification-time => #6.1(uint)
suit-meta-creation-time => #6.1(uint)
```

4.6.3. Component Default Permissions

Typical permissions management systems require read, write, and execute permissions that are applied to all users who do not have their own explicit permissions. These are the default permissions for the current component. Default permissions are described by the following CDDL:

```
SUIT_meta_permissions = uint .bits SUIT_meta_permission_bits
SUIT_meta_permission_bits = &(amp;
    write_attr_ex: 13,
    read_attr_ex: 12,
    sync: 11,
    delete: 10,
    recurse_delete: 9,
    write_attr: 8,
    change_owner: 7,
    change_perm: 6,
    read_perm: 5,
    read_attr: 4,
    creatdir_append: 3,
    list_read: 2,
    create_write: 1,
    traverse_exec: 0,
    * $$SUIT_meta_permission_bits_extensions
)
```

4.6.4. User, Role, Group permissions

Many filesystems have users and groups. Additionally some have roles. Actors that have these associations can have specific permissions associated with them for each component. Each of these sets of permissions is defined the same way: with a map of actor identifiers to permissions.

```
SUIT_meta_permission_map = {
    + SUIT_meta_actor_id => SUIT_meta_permissions
}
```

The `SUIT_meta_actor_id` is the same as defined for Creator, Section 4.6.1.

4.6.5. File Type

File Type typically identifies whether a file is a directory, regular file, or symbolic link. If not specified, File Type defaults to regular file.

This enables specific management operations for SUIT command sequences:

- * To create a directory
 - Set the Component Index to the Component Identifier of the directory to be created
 - Set the Component metadata, including the file type for directory
 - Set `suit-parameter-content` to an empty bstr
 - Invoke `suit-directive-write`
- * To create a symbolic link
 - Set the Component Index to the Component Identifier of the link to be created
 - Set the Component metadata, including the file type for symbolic link
 - Set `suit-parameter-content` to the link target
 - Invoke `suit-directive-write`

For example, the following Payload Fetch & Install sequences will create a new `/usr/local/bin` directory, download `https://cdn.example/example3.bin` into a new file: `/usr/local/bin/example3`, then create a symlink at `/usr/bin/example` that points to `/usr/local/bin/example3`.

- * Common has components for:
 - `/usr/bin/example`
 - `/usr/local/bin`
 - `/usr/local/bin/example3`
- * Payload fetch:

- set component index = 1
 - set parameters:
 - o content = h''
 - o metadata = {file-type: directory}
 - write
 - set component index = 2
 - set URI = "https://cdn.example/example3.bin"
 - fetch
 - condition image digest
- * Install:
- set component index = 0
 - set parameters:
 - o content = "/usr/local/bin/example3"
 - o metadata = {file-type: symlink}
 - write

5. Extension Commands

The following table defines the semantics of the commands defined in this specification in the same way as in the Abstract Machine Description, Section 6.4, of [I-D.ietf-suit-manifest].

All commands defined in this specification are OPTIONAL to implement. A Recipient that encounters a command it does not implement MUST reject the manifest as defined in [I-D.ietf-suit-manifest] Section 8.4.2, ensuring that update behaviour is never ambiguous.

Command Name	CDDL Identifier	Semantic of the Operation
Use Before	suit-condition-use-before	assert(now() < current.params[use-before])
Check Image Not Match	suit-condition-image-not-match	assert(not binary-match(digest(current), current.params[digest]))
Check Minimum Battery	suit-condition-minimum-battery	assert(battery >= current.params[minimum-battery])
Check Update Authorized	suit-condition-update-authorized	assert(isAuthorized(current.params[priority]))
Check Version	suit-condition-version	assert(version_check(current, current.params[version]))
Wait For Event	suit-directive-wait	until event(arg), wait
Override Multiple	suit-directive-override-multiple	components[i].params[k] := v for-each k,v in d for-each i,d in arg
Copy Params	suit-directive-copy-params	current.params[k] = components[i].params[k] for k in l for i,l in arg

Table 3

5.1. suit-condition-use-before

Verify that the current time is BEFORE the specified time. `suit-condition-use-before` is used to specify the last time at which an update is to be installed. The recipient evaluates the current time against the `suit-parameter-use-before` parameter (Section 4.1), which MUST have already been set as a parameter, encoded as seconds after 1970-01-01 00:00:00 UTC. Timestamp conditions MUST be evaluated in 64 bits, regardless of encoded CBOR size. `suit-condition-use-before` is OPTIONAL to implement.

5.2. `suit-condition-image-not-match`

Verify that the current component does not match the `suit-parameter-image-digest` (Section 8.4.8.6 of [I-D.ietf-suit-manifest]). If no digest is specified, the condition fails. `suit-condition-image-not-match` is OPTIONAL to implement.

5.3. `suit-condition-minimum-battery`

`suit-condition-minimum-battery` provides a mechanism to test a Recipient's battery level before installing an update. This condition is primarily for use in primary-cell applications, where the battery is only ever discharged. For batteries that are charged, `suit-directive-wait` is more appropriate, since it defines a "wait" until the battery level is sufficient to install the update. `suit-condition-minimum-battery` is specified in mWh. `suit-condition-minimum-battery` is OPTIONAL to implement. `suit-condition-minimum-battery` consumes `suit-parameter-minimum-battery` (Section 4.2).

5.4. `suit-condition-update-authorized`

Request authorization from the application and fail if not authorized. This can allow a user to decline an update. `suit-parameter-update-priority` (Section 4.3) provides an integer priority level that the application can use to determine whether or not to authorize the update. Priorities are application defined. `suit-condition-update-authorized` is OPTIONAL to implement.

5.5. `suit-condition-version`

`suit-condition-version` allows comparing versions of firmware. Verifying image digests is preferred to version checks because digests are more precise. `suit-condition-version` examines a component's version against the version info specified in `suit-parameter-version` (Section 4.4).

5.6. `suit-directive-wait`

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

1. Authorization
2. External power
3. Network availability
4. Other device firmware version

5. Time
6. Time of day
7. Day of week

5.7. `suit-directive-override-multiple`

This directive enables setting parameters for multiple components at the same time. This allows a small reduction in encoding overhead:

- * without `override-multiple`, the encoding for each component consists of:
 - `set-component-index` (2 bytes)
 - `override-parameters` (1 byte + parameter map)
- * with `override-multiple`, the encoding for each component consists of:
 - the component index key (1 byte)
 - the parameter map

`Override-multiple` requires the command (1-2 bytes) and one additional map to hold the parameter sets (1 byte). For one component, there is no savings. For multiple components, there is an encoding savings of 2 bytes per component.

Implementations can structure code so that `override-multiple` follows a code-path nearly identical to `set-component-index` + `override-parameters`.

This command is purely an encoding alias for `set-component-index` and `override-parameters`. The component index is set to the last component listed in the `override-multiple` argument when `override-multiple` completes.

The following CDDL defines the argument for `suit-directive-override-multiple`:

```
CDDL SUIT_Override_Mult_Arg = { + uint => {+ $$SUIT_Parameters} }
```

5.8. suit-directive-copy-params

suit-directive-copy-params enables a manifest author to specify one or more components to copy parameters from, and a list of parameters to copy from each specified source component.

The behaviour is exactly the same as override parameters, but with parameter values defined in existing components. Parameters are only copied between identical keys (no copying from URI to digest, for example).

For each entry in the map, the manifest processor sets the source component to be the component identified by the index contained in the map key. For each parameter identified in the copy list, the manifest processor copies the parameter from the source component to the current component.

The following CDDL defines the argument for suit-directive-copy-params:

```
CDDL SUIT_Directive_Copy_Params = { + uint => [+ int] }
```

6. IANA Considerations

IANA is requested to allocate the commands, parameters, and metadata values shown in the following tables.

6.1. SUIT Envelope Elements

Label	Name	Reference
14	CoSWID	Section 3.2

Table 4

6.2. SUIT Manifest Elements

Label	Name	Reference
6	Set Version	Section 3.1
14	CoSWID	Section 3.2

Table 5

6.3. SUIT Commands

Label	Name	Reference
4	Use Before	Section 5.1
25	Image Not Match	Section 5.2
26	Minimum Battery	Section 5.3
27	Update Authorized	Section 5.4
28	Version	Section 5.5
29	Wait For Event	Section 5.6
34	Override Multiple	Section 5.7
35	Copy Params	Section 5.8

Table 6

6.4. SUIT Parameters

Label	Name	Reference
4	Use Before	Section 4.1
26	Minimum Battery	Section 4.2
27	Update Priority	Section 4.3
28	Version	Section 4.4
29	Wait Info	Section 4.5
30	Component Metadata	Section 4.6

Table 7

6.5. SUIT Component Text Values

Label	Name	Reference
7	Component Version Required	Section 3.3
8	Current Version	Section 3.4

Table 8

7. Security Considerations

This document extends the SUIT manifest specification. A detailed security treatment can be found in the architecture [RFC9019] and in the information model [I-D.ietf-suit-information-model] documents.

The free-text fields introduced in Sections Section 3.3 and Section 3.4 are intended solely for human consumption. Recipients MUST treat those values as untrusted input: they MUST NOT evaluate the text, execute embedded markup, or override machine-readable decisions derived from `suit-set-version` or `suit-parameter-version`. Implementations SHOULD bound the length of displayed text to mitigate interface flooding and log injection.

Component metadata (Section 4.6) can expose operator identifiers, file paths, or other locally meaningful strings. Deployments SHOULD validate these values against local policy before applying them, and MUST handle missing or malformed metadata defensively so that the update agent does not escalate privileges or disclose sensitive information inadvertently.

8. References

8.1. Normative References

[I-D.ietf-suit-manifest]

Moran, B., Tschofenig, H., Birkholz, H., Zandberg, K., and O. R  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>>.
- [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>>.
- [RFC9393] Birkholz, H., Fitzgerald-McKay, J., Schmidt, C., and D. Waltermire, "Concise Software Identification Tags", RFC 9393, DOI 10.17487/RFC9393, June 2023, <<https://www.rfc-editor.org/rfc/rfc9393>>.
- [semver] "Semantic Versioning 2.0.0", 18 June 2013, <<https://semver.org>>.

8.2. Informative References

- [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://datatracker.ietf.org/doc/html/draft-ietf-suit-information-model-13>>.
- [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>>.

Appendix 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].

```

$$unseverable-manifest-member-extensions //= (
    suit-set-version =>
        bstr .cbor SUIT_Condition_Version_Comparison_Value
)
$$SUIT_severable-members-extensions //= (
    suit-coswid => bstr .cbor concise-swid-tag)

$$severable-manifest-members-choice-extensions //= (
    suit-coswid => bstr .cbor concise-swid-tag / SUIT_Digest
)

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

SUIT_Directive //= (
    suit-directive-wait,              SUIT_Rep_Policy)

SUIT_Directive //= (
    suit-directive-override-multiple, SUIT_Override_Mult_Arg)
SUIT_Directive //=(
    suit-directive-copy-params,      SUIT_Directive_Copy_Params)

SUIT_Override_Mult_Arg = {
    + uint => {+ $$SUIT_Parameters}
}
SUIT_Directive_Copy_Params = {
    + uint => [+ int]
}

SUIT_Wait_Event = { + SUIT_Wait_Events }

SUIT_Wait_Events //= (suit-wait-event-authorization => int)
SUIT_Wait_Events //= (suit-wait-event-power => int)
SUIT_Wait_Events //= (suit-wait-event-network => int)
SUIT_Wait_Events //= (suit-wait-event-other-device-version
=> SUIT_Wait_Event_Argument_Other_Device_Version)
SUIT_Wait_Events //= (suit-wait-event-time => uint); Timestamp
SUIT_Wait_Events //= (suit-wait-event-time-of-day
=> uint); Time of Day (seconds since 00:00:00)

```

```

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

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

$$SUIT_Parameters // = (suit-parameter-use-before => uint)
$$SUIT_Parameters // = (suit-parameter-minimum-battery => uint)
$$SUIT_Parameters // = (suit-parameter-update-priority => int)
$$SUIT_Parameters // = (suit-parameter-version =>
    bstr .cbor SUIT_Parameter_Version_Match)
$$SUIT_Parameters // = (suit-parameter-wait-info =>
    bstr .cbor SUIT_Wait_Event)
$$SUIT_Parameters // = (suit-parameter-component-metadata =>
    bstr .cbor SUIT_Component_Metadata)

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

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

SUIT_Condition_Version_Comparison_Value = [+int]

```

```
SUIT_Component_Metadata = {
    ? suit-meta-default-permissions => SUIT_meta_permissions,
    ? suit-meta-user-permissions => SUIT_meta_permission_map,
    ? suit-meta-group-permissions => SUIT_meta_permission_map,
    ? suit-meta-role-permissions => SUIT_meta_permission_map,
    ? suit-meta-file-type => SUIT_Filetype,
    ? suit-meta-modification-time => #6.1(uint),
    ? suit-meta-creation-time => #6.1(uint),
    ? suit-meta-creator => SUIT_meta_actor_id,
    * $$SUIT_Component_Metadata_Extensions
}

suit-meta-default-permissions = 1
suit-meta-user-permissions = 2
suit-meta-group-permissions = 3
suit-meta-role-permissions = 4
suit-meta-file-type = 5
suit-meta-modification-time = 6
suit-meta-creation-time = 7
suit-meta-creator = 8

SUIT_meta_permissions = uint .bits SUIT_meta_permission_bits
SUIT_meta_permission_bits = &(amp;
    write_attr_ex: 13,
    read_attr_ex: 12,
    sync: 11,
    delete: 10,
    recurse_delete: 9,
    write_attr: 8,
    change_owner: 7,
    change_perm: 6,
    read_perm: 5,
    read_attr: 4,
    creatdir_append: 3,
    list_read: 2,
    create_write: 1,
    traverse_exec: 0,
    * $$SUIT_meta_permission_bits_extensions
)

SUIT_meta_permission_map = {
    + SUIT_meta_actor_id => SUIT_meta_permissions
}

SUIT_meta_actor_id = UUID_Tagged / bstr / tstr / int
UUID_Tagged = #6.37(bstr)

SUIT_Filetype /= suit-filetype-regular
```

```

SUIT_Filetype /= suit-filetype-directory
SUIT_Filetype /= suit-filetype-symlink

suit-filetype-regular = 1
suit-filetype-directory = 2
suit-filetype-symlink = 3

$$suit-text-component-key-extensions // = (
    suit-text-version-required => tstr)
$$suit-text-component-key-extensions // = (
    suit-text-current-version => tstr)

suit-set-version = 6
suit-coswid = 14
suit-condition-use-before = 4
suit-condition-image-not-match = 25
suit-condition-minimum-battery = 26
suit-condition-update-authorized = 27
suit-condition-version = 28

suit-directive-wait = 29
suit-directive-override-multiple = 34
suit-directive-copy-params = 35

suit-wait-event-authorization = 1
suit-wait-event-power = 2
suit-wait-event-network = 3
suit-wait-event-other-device-version = 4
suit-wait-event-time = 5
suit-wait-event-time-of-day = 6
suit-wait-event-day-of-week = 7
suit-wait-event-time-of-day-utc = 8
suit-wait-event-day-of-week-utc = 9

suit-parameter-use-before = 4
suit-parameter-minimum-battery = 26
suit-parameter-update-priority = 27
suit-parameter-version = 28
suit-parameter-wait-info = 29
suit-parameter-component-metadata = 30

suit-text-version-required = 7
suit-text-current-version = 8

```

Authors' Addresses

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

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