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

Table of Contents

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.

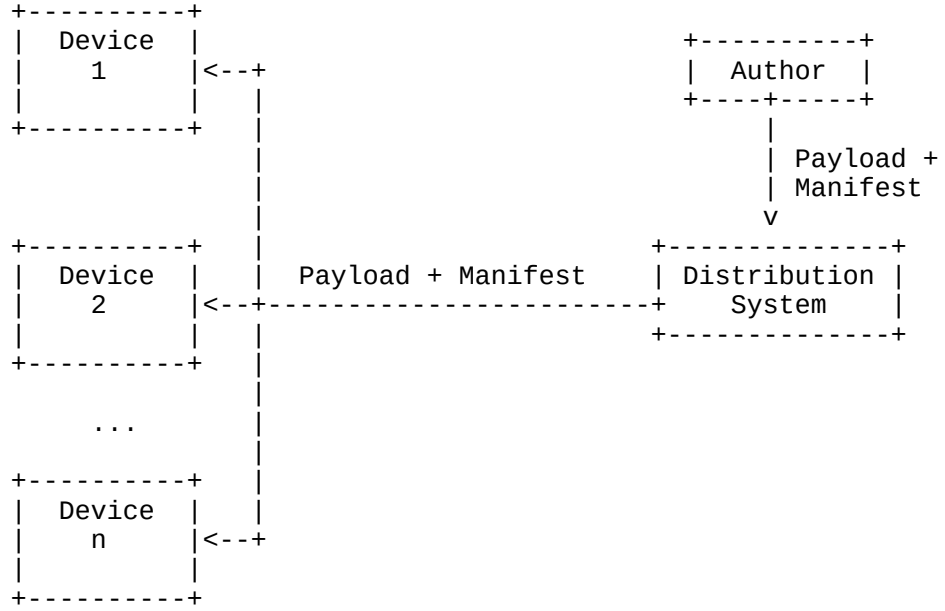

Figure 1: Architecture for the distribution of Encrypted Payloads.

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 SUIT 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 SUIT_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[Ri, S] using ES-DH
2b.      ENC(CEK, KEK[Ri, 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. his 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[Ri, S]) where each recipient Ri receives a unique CEK. Assume that KEK[R1, S],..., KEK[Rn, 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 SUIT_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].

```
SUIT_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 "SUIT 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’5886CD61DD875862E5AAA820E7A15274C968A9BC96048DDCACE32F50C3
    651BA3’
  - y: h’9EED8125E932CD60C0EAD3650D0A485CF726D378D1B016ED4298B2961E
    258F1B’
  - d: h’60FE6DD6D85D5740A5349B6F91267EEAC5BA81B8CB53EE249E4B4EB102
    C476B3’
  - kid: ’kid-2’
* KDF Context
  - Algorithm ID: -3 (A128KW)
  - SuppPubInfo
    - keyDataLength: 128
    - protected: << { / alg / 1: -29 / ECDH-ES+A128KW / } >>
    - other: ’SUIT Payload Encryption’

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

```
D8608443A10101A1054CF14AAB9D81D51F7AD943FE87F6818344A101381C
A120A401022001215820730 24F415AA51529A66CCEFD88F3F62A734492FF
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:

* Pi = Plaintext blocks
* Ci = Ciphertext blocks
* E = Encryption function
* k = Symmetric key
* = XOR operation

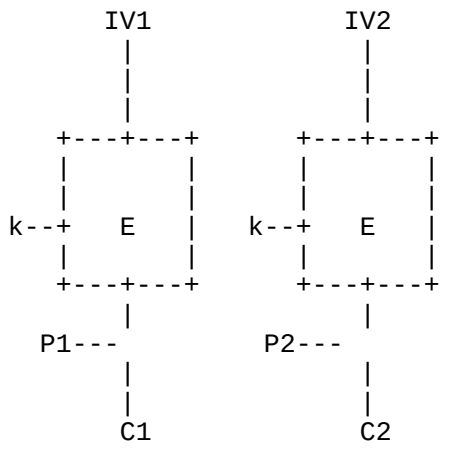

Figure 11: AES-CTR Operation

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'5886CD61DD875862E5AAA820E7A15274C968A9BC96048DDCACE32F50C3
    651BA3'

  - y: h'9EED8125E932CD60C0EAD3650D0A485CF726D378D1B016ED4298B2961E
    258F1B'

  - d: h'60FE6DD6D85D5740A5349B6F91267EEAC5BA81B8CB53EE249E4B4EB102
    C476B3'

- kid: 'kid-2'

* KDF Context

  - Algorithm ID: -3 (A128KW)

  - SuppPubInfo

    o keyDataLength: 128

    o protected: << { / alg / 1: -29 / ECDH-ES+A128KW / } >>

    o other: 'SUIT 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 concent.

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.

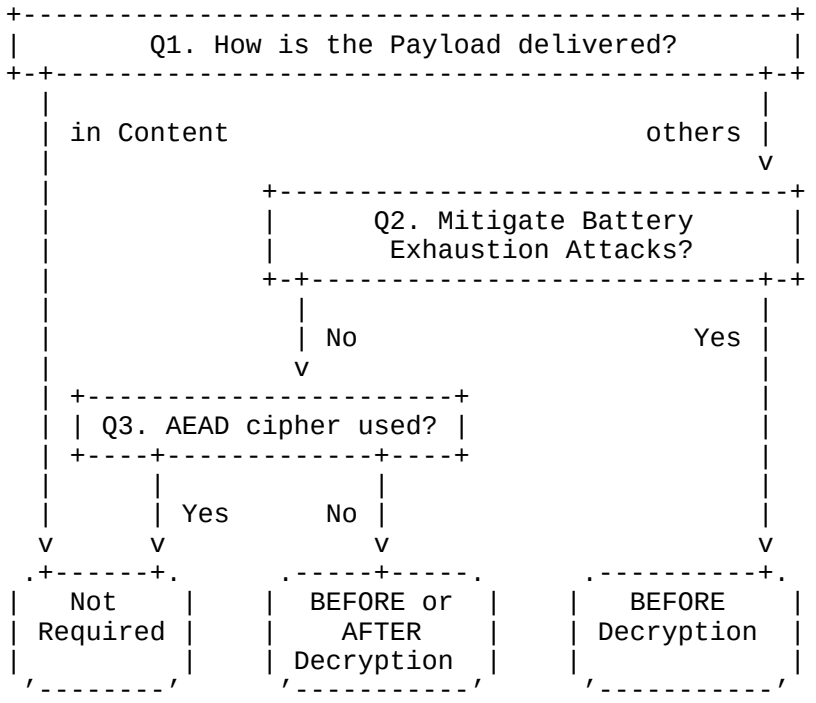

Figure 16: Decision Tree: Validating the Payload

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.

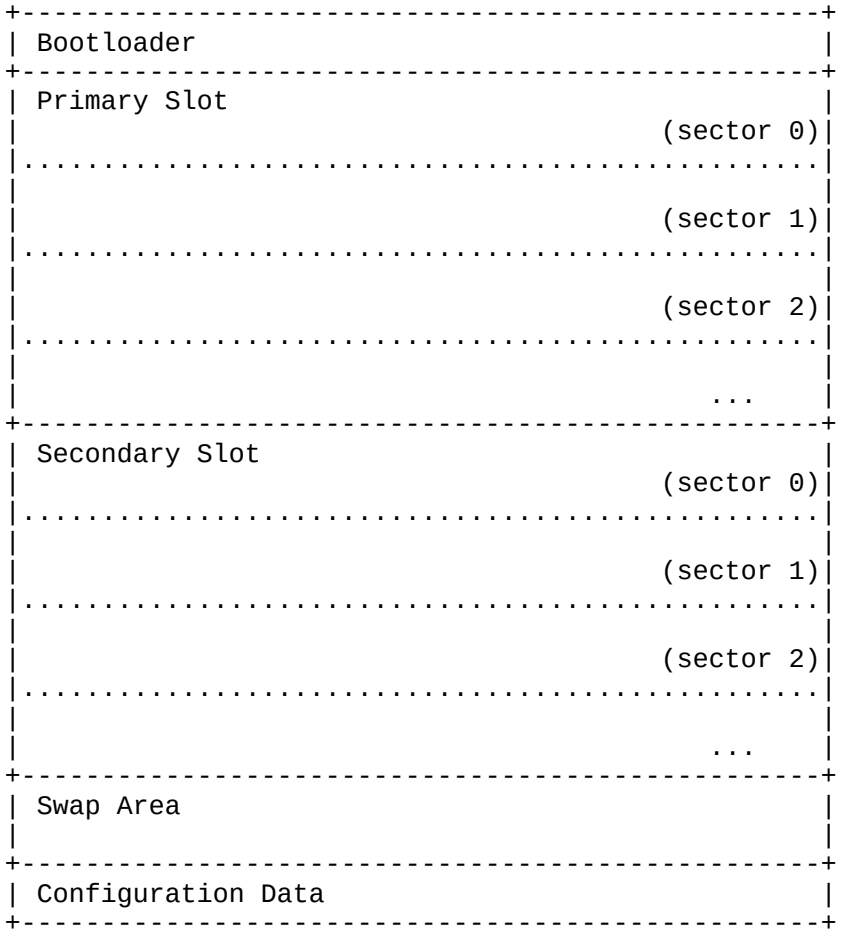

Figure 17: Example Flash Area Layout

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

```
/  1/ / SUIT_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
7B0358E8A4010102010357A10281815264656372797074656 42D6669726D
776172651458C7860C0014A212582E758C4B7BBAE2C4C1D462423E0F0DC3
164FFA7B85BB94D4BD6D7ED26AB32FEB063385D4D3465927EC82CB5E198A
5913588CD8608443A10101A1054CF14AAB9D81D51F7AD943FE87F6818344
A101381CA220A401022001215820730 24F415AA51529A66CCEFD88F3F62A
734492FF45F6AD37FD2888E73EAF19DA2258204005B48A6FD091AA6ABFE3
CFBEEDE88B347E521D43405FDBD7D2CFF0EBC21B2604456B69642D325818
A06B8E6550F308712B1DF044B21B7D11D9B22792F1DE0997120F
```

9.4. ES-DH Example with Dependency

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

The dependent manifest is signed with another key:

```
-----BEGIN EC PRIVATE KEY-----
MHcCAQEEIIQa67e56m8CYL5zVaJFiLl30j0qxb8ray2DeUMqH+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 SUIT manifest in diagnostic notation (with line breaks added for readability) is shown here:

```
/  1/ / SUIT_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 / ['dependenty-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
540359016CA501010201035837A201A101A10181581864657065 6E64656E
63792D6D616E69666573742E737569740281815264656372797074656 42D
6669726D7761726505815764657065 6E64656E742D6D616E69666573742E
737569741459010F8E0C0014A212582E758C4B7BBAE2C4C1D462423E0F0D
C3164FFA7B85BB94D4BD6D7ED26AB32FEB063385D4D3465927EC82CB5E19
8A5913588CD8608443A10101A1054CF14AAB9D81D51F7AD943FE87F68183
44A101381CA220A401022001215820 73024F415AA51529A66CCEFD88F3F6
2A734492FF45F6AD37FD2888E73EAF19DA2258204005B48A6FD091AA6ABF
E3CFBEEDE88B347E521D43405FDBD7D2CFF0EBC21B2604456B69642D3258
18A06B8E6550F308712B1DF044B21B7D11D9B22792F1DE09970C0114A303
5824822F58204B15C90FBD776A820E7E733DF040D90B356B5C75982ECAEC
E8673818179BDF160E18F7157423646570656E64656E63792D6D616E6966
657374150F070F0B0F7423646570656E64656E63792D6D616E6966657374
58F7D86BA2025873825824822F58204B15C90FBD776A820E7E733DF040D9
0B356B5C75982ECAECE8673818179BDF16584AD28443A10128A0F65840BF
95C29295B45470EF819E7F4E3C9084F4534E26469C0A0F2B8B9664881A53
59D500F81BD3A6436A025C3E92E51CD7148F83DF47D29FF253F8D41B4D35
0A2B7503587BA601010201035849A202818152646563727970746564 2D66
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 ofLong-TermKeys	Number of ContentEncryption Keys (CEKs)	Use Case	Recommended?
Same key for all devices	Single CEK per payload shared with all devies	Legacy Usage	No, bad practice
One key per device	Single CEK per payload shared with all devies	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. Rnningstad, "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., Rnningstad, 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>.

[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>.

[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 SUIT 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 SUIT Manifest CDDL. The SUIT 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, yvind Rnningstad, Dave Thaler, Laurence Lundblade, Christian Amsss, 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, ric 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