Encrypted Payloads in SUIT Manifests

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 25 April 2024.

Copyright Notice

Copyright (c) 2023 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. Conventions and Terminology
3. Architecture
4. Encryption Extensions
5. Extended Directives
6. Content Key Distribution
   6.1. Content Key Distribution with AES Key Wrap
       6.1.1. Introduction
       6.1.2. Deployment Options
       6.1.3. CDDL
       6.1.4. Example
   6.2. Content Key Distribution with Ephemeral-Static Diffie-Hellman
       6.2.1. Introduction
       6.2.2. Deployment Options
       6.2.3. CDDL
       6.2.4. Context Information Structure
       6.2.5. Example
   6.3. Content Encryption
7. Firmware Updates on IoT Devices with Flash Memory
   7.1. AES-CBC
       7.1.1. AES-KW + AES-CBC Example
       7.1.2. ES-DH + AES-CBC Example
   7.2. AES-CTR
       7.2.1. AES-KW + AES-CTR Example
       7.2.2. ES-DH + AES-CTR Example
   7.3. Battery Exhaustion Attacks
8. Complete Examples
   8.1. AES Key Wrap Example with Write Directive
   8.2. AES Key Wrap Example with Fetch + Copy Directives
   8.3. ES-DH Example with Write + Copy Directives
   8.4. ES-DH Example with Dependency
9. Operational Considerations
10. Security Considerations
11. IANA Considerations
12. References
   12.1. Normative References
   12.2. Informative References
Appendix A. Full CDDL
Acknowledgements
Authors' Addresses
1. Introduction

Vulnerabilities with Internet of Things (IoT) devices have raised the need for a reliable and secure firmware update mechanism that is also suitable for constrained devices. To protect firmware images, the SUIT manifest format was developed [I-D.ietf-suit-manifest]. It provides a bundle of metadata, including where to find the payload, the devices to which it applies and a security wrapper.

[RFC9124] details the information that has to be provided by the SUIT manifest format. In addition to offering protection against modification, via a digital signature or a message authentication code, confidentiality may also be afforded.

Encryption prevents third parties, including attackers, from gaining access to the payload. Attackers typically need intimate knowledge of a binary, such as a firmware image, to mount their attacks. For example, return-oriented programming (ROP) [ROP] requires access to the binary and encryption makes it much more difficult to write exploits.

While the original motivating use case of this document was firmware encryption, the use of SUIT manifests has been extended to other use cases requiring integrity and confidentiality protection, such as:

* software packages,
* personalization data,
* configuration data, and
* machine learning models.

Hence, we use the term payload to generically refer to all those objects.

The payload is encrypted using a symmetric content encryption key, which can be established using a variety of mechanisms; this document defines two content key distribution methods for use with the IETF SUIT manifest, namely:

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

The former method relies on asymmetric key cryptography while the latter uses symmetric key cryptography.

Our goal was to reduce the number of content key distribution methods for use with payload encryption and thereby increase
interoperability between different SUIT manifest parser implementations.

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 IETF 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]

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 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] describes the architecture for distributing payloads and manifests from an author to devices. It does, however, not detail
the use of payload encryption. This document enhances the architecture to support encryption.

Figure 1 shows the distribution system, which represents a file server and the device management infrastructure.

The sender (author) needs to know the recipient (device) to use encryption. For AES-KW, the KEK needs to be known and, in case of ES-DH, the sender needs to be in possession of the public key of the recipient. The public key and parameters may be in the recipient's X.509 certificate [RFC5280]. For authentication of the sender and for integrity protection the recipients must be provisioned with a trust anchor when a manifest is protected using a digital signature. When a MAC is used to protect the manifest then a symmetric key must be shared by the recipient and the sender.

With encryption, the author cannot just create a manifest for the payload and sign it, since the subsequent encryption step by the distribution system would invalidate the signature over the manifest. (The content key distribution information is embedded inside the COSE_Encrypt structure, which is included in the SUIT manifest.) Hence, the author has to collaborate with the distribution system. The varying degree of collaboration is discussed below.

![Figure 1: Architecture for the distribution of Encrypted Payloads.](image)

---

Figure 1: Architecture for the distribution of Encrypted Payloads.
The author has several deployment options, namely:

1. The author, as the sender, obtains information about the recipients and their keys from the distribution system. Then, it performs the necessary steps to encrypt the payload. As a last step it creates one or more manifests. The device(s) perform decryption and act as recipients.

2. The author treats the distribution system as the initial recipient. Then, the distribution system decrypts and re-encrypts the payload for consumption by the device (or the devices). Delegating the task of re-encrypting the payload to the distribution system offers flexibility when the number of devices that need to receive encrypted payloads changes dynamically or when updates to KEKs or recipient public keys are necessary. As a downside, the author needs to trust the distribution system with performing the re-encryption of the payload.

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

1. The distributor replaces the COSE_Encrypt in the manifest and then signs the manifest again. However, the COSE_Encrypt structure is contained within a signed container, which presents a problem: replacing the COSE_Encrypt with a new one will cause the digest of the manifest to change, thereby changing the signature. This means that the distributor must be able to sign the new manifest. If this is the case, then the distributor gains the ability to construct and sign manifests, which allows the distributor the authority to sign code, effectively presenting the distributor with full control over the recipient. Because distributors typically perform their re-encryption online in order to handle a large number of devices in a timely fashion, it is not possible to air-gap the distributor's signing operations. This impacts the recommendations in Section 4.3.17 of [RFC9124].

2. The distributor uses a two-manifest system. More precisely, the distributor constructs a new manifest that overrides the COSE_Encrypt using the dependency system defined in [I-D.ietf-suit-trust-domains]. This incurs additional overhead: one additional signature verification and one additional manifest, as well as the additional machinery in the recipient needed for dependency processing.

These two models also present different threat profiles for the distributor. If the distributor only has encryption rights, then an attacker who breaches the distributor can only mount a limited attack: they can encrypt a modified binary, but the recipients will
identify the attack as soon as they perform the required image digest check and revert back to a correct image immediately.

It is RECOMMENDED that distributors are implemented using a two-manifest system in order to distribute content encryption keys without requiring re-signing of the manifest, despite the increase in complexity and greater number of signature verifications that this imposes on the recipient.

4. Encryption Extensions

This specification introduces a new extension to the SUIT_Parameters structure.

The SUIT_Encryption_Info structure (called suit-parameter-encryption-info in Figure 2) contains the content key distribution information. The content of the SUIT_Encryption_Info structure is explained in Section 6.1 (for AES-KW) and in Section 6.2 (for ES-DH).

Once a CEK is available, the steps described in Section 6.3 are applicable. These steps apply to both content key distribution methods described in this section.

The SUIT_Encryption_Info structure is either carried inside the suit-directive-override-parameters or the suit-directive-set-parameters parameters used in the "Directive Write" and "Directive Copy" directives. An implementation claiming conformance with this specification must implement support for these two parameters. Since a device will typically only support one of the content key distribution methods, the distribution system needs to know which of two specified methods is supported. Mandating only a single content key distribution method for a constrained device also reduces the code size.

SUIT_Parameters //= (suit-parameter-encryption-info => bstr .cbor SUIT_Encryption_Info)
suit-parameter-encryption-info = 19

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.]
5. Extended Directives

This specification extends these directives:

*Directive Write (suit-directive-write) to decrypt the content specified by suit-parameter-content with suit-parameter-encryption-info.

*Directive Copy (suit-directive-copy) to decrypt the content of the component specified by suit-parameter-source-component with suit-parameter-encryption-info.

Examples of the two directives are shown below.

Figure 3 illustrates the Directive Write. The encrypted payload specified with parameter-content, namely h'EA1...CED' in the example, is decrypted using the SUIT_Encryption_Info structure referred to by parameter-encryption-info, i.e., h'D86...1F0'. The resulting plaintext payload is stored into component #0.

```
/directive-override-parameters / 20, {
  / parameter-content / 18: h'EA1...CED',
  / parameter-encryption-info / 19: h'D86...1F0'
},
/directive-write / 18, 15
```

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

Figure 4 illustrates the Directive Copy. In this example the encrypted payload is found at the URI indicated by the parameter-uri, i.e. "http://example.com/encrypted.bin". The encrypted payload will be downloaded and stored in component #1. Then, the information in the SUIT_Encryption_Info structure of the parameter-encryption-info, i.e. h'D86...1F0', will be used to decrypt the content in component #1 and the resulting plaintext payload will be stored into component #0.

```
/directive-set-component-index / 12, 1,
/directive-fetch / 21, 15,
/directive-set-component-index / 12, 0,
/directive-override-parameters / 20, {
},
/directive-copy / 22, 15
```
Figure 4: Example showing the extended suit-directive-copy.

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 payload 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 6.3 are applicable. These steps apply to both content key distribution methods.

6. Content Key Distribution

The sub-sections below describe two content key distribution methods, namely AES Key Wrap (AES-KW) and Ephemeral-Static Diffie-Hellman (ES-DH). Many other methods are specified in the literature, and even supported by COSE. New methods can be added via enhancements to this specification. The two specified methods were selected to their maturity, different security properties, and to ensure interoperability in deployments.

The two content key distribution methods require the CEKs to be randomly generated. It must be ensured that the guidelines for random number generation in [RFC8937] are followed.

When an encrypted payload is sent to multiple recipients, there are different deployment options. To explain these options we use the following notation:

- KEK(R1, S) refers to a KEK shared between recipient R1 and the sender S. The KEK, as a concept, is used by AES Key Wrap but not by ES-DH.
- 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.
6.1. Content Key Distribution with AES Key Wrap

6.1.1. Introduction

The AES Key Wrap (AES-KW) algorithm is described in [RFC3394], and can be 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 in the COSE_recipient structure alongside the information needed for AES-KW. The COSE_recipient structure, which is a substructure of the COSE_Encrypt structure, contains the CEK encrypted by the KEK.

To provide high security for AES Key Wrap, it is important that the KEK is of high entropy, and that implementations protect the KEK from disclosure. Compromise of the KEK may result in the disclosure of all data protected with that KEK, including binaries, and configuration data.

The COSE_Encrypt structure conveys information for encrypting the payload, which includes information like the algorithm and the IV, even though the payload may not be embedded in the COSE_Encrypt.ciphertext if it is conveyed as detached content.

6.1.2. 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). 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. }

6.1.3. CDDL

The CDDL for the COSE_Encrypt_Tagged structure is shown in Figure 5. empty_or_serialized_map and header_map are structures defined in [RFC9052].
outer_header_map_protected = empty_or_serialized_map
outer_header_map_unprotected = header_map

SUIT_Encryption_Info_AESKW = [
    protected : bstr .cbor outer_header_map_protected,
    unprotected : outer_header_map_unprotected,
    ciphertext : bstr / nil,
    recipients : [ + COSE_recipient_AESKW .within COSE_recipient ]
]

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, ; algorithm identifier
    ? 4 => bstr, ; identifier of the KEK pre-shared with the recipi
    * 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.

6.1.4. Example

This example uses the following parameters:

*Algorithm for payload encryption: AES-GCM-128

-IV: h'93702C81590F845D9EC866CCAC767BD1'

*Algorithm id for key wrap: A128KW

*KEK COSE_Key (Secret Key):

 */ kty / 1: 4 / Symmetric /

 */ k / -1: 'aaaaaaaaaaaaaaaaaa'

*KID: 'kid-1'
"Plaintext: "This is a real firmware image."

-in hex:
546869732069732061207265616C206669726D7761726520696D6167652E

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

D8608443A10101A10550F14AA89D81D51F7AD943FE87AF4F70CDF6818341
A0A2012204456B69642D31581875603FFC9518D794713C8CA8A115A7FB32
565A6D59534D62

The resulting COSE_Encrypt structure in a diagnostic format is shown in Figure 6.

96([
  / protected: / << {
    / alg / 1: 1 / AES-GCM-128 /
  } >>,
  / unprotected: / {
    / IV / 5: h'F14AAB9D81D51F7AD943FE87AF4F70CD' },
  / payload: / null / detached ciphertext /,
  / recipients: / [
    [ / protected: / << {
      } >>,
    / unprotected: / {
      / alg / 1: -3 / A128KW /,
      / kid / 4: 'kid-1'
    },
    / payload: / h'75603FFC9518D794713C8CA8A115A7FB32565A6D59534D62'
      / CEK encrypted with KEK /
  ]
]
])

Figure 6: COSE_Encrypt Example for AES Key Wrap

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

2F59C3A34D9570FB99A5382E66466A3221A8AD85CE508BA306FB431A60EF
A5AAAA078355070205A4B196832DF17F
6.2. Content Key Distribution with Ephemeral-Static Diffie-Hellman

6.2.1. Introduction

Ephemeral-Static Diffie-Hellman (ES-DH) is a scheme that provides public key encryption given a recipient's public key. There are multiple variants of this scheme; this document re-uses the variant specified in Section 8.5.5 of [RFC9052].

The following two layer structure is used:

*Layer 0: Has a content encrypted with the CEK. The content may be detached.

*Layer 1: Uses the AES Key Wrap algorithm to encrypt the randomly generated CEK with the KEK derived with ES-DH, whereby the resulting symmetric key is fed into the HKDF-based key derivation function.

As a result, the two layers combine ES-DH with AES-KW and HKDF. An example is given in Figure 9.

6.2.2. Deployment Options

There are two deployment options with this approach. We assume that recipients are always configured with a device-unique public / private key pair.

*A sender wants to transmit a payload to multiple recipients. All recipients shall receive the same encrypted payload, i.e. the same CEK is used. One COSE_recipient structure per recipient is used and it contains the CEK encrypted with the KEK. To generate the KEK, each COSE_recipient structure contains a COSE_recipient_inner structure to carry the sender's ephemeral key and an identifier for the recipients 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 a payload with a different CEK for each recipient. This results in n-manifests. This approach is useful when payloads contain information unique to a device. The encryption operation then effectively becomes ENC(payload_i,
CEK(Ri, S)). Assume that KEK(R1, S),..., KEK(Rn, S) have been generated for the different recipients using ES-DH. The following steps need to be made by the sender:

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))
   }

6.2.3. CDDL

The CDDL for the COSE_Encrypt_Tagged structure is shown in Figure 7. Only the minimum number of parameters is shown. empty_or_serialized_map and header_map are structures defined in [RFC9052].

outer_header_map_protected = empty_or_serialized_map
outer_header_map_unprotected = header_map

SUIT_Encryption_Info_ESDH = [
  protected : bstr .cbor outer_header_map_protected,
  unprotected : outer_header_map_unprotected,
  ciphertext : bstr / nil,
  recipients : [ + COSE_recipient_ESDH .within COSE_recipient ]
]

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,         ; algorithm identifier
  * label => values ; extension point
}

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

Figure 7: CDDL for ES-DH-based Content Key Distribution
See Section 6.3 for a description on how to encrypt the payload.

6.2.4. Context Information Structure

The context information structure is used to ensure that the derived keying material is "bound" to the context of the transaction. This specification re-uses the structure defined in Section 5.2 of [RFC9053] and tailors it accordingly.

The following information 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 8 require an explanation:

* The COSE_KDF_Context.AlgorithmID field MUST contain the algorithm identifier for AES Key Wrap algorithm utilized. This specification uses the following values: A128KW (value -3), A192KW (value -4), or A256KW (value -5).

* The COSE_KDF_Context.SuppPubInfo.keyDataLength field MUST contain the key length of the algorithm in the COSE_KDF_Context.AlgorithmID field expressed as the number of bits. 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 in which the ES-DH content key distribution algorithm is used and 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 fields) the identifier of the content key distribution method.
The HKDF-based key derivation function MAY contain a salt value, as described in Section 5.1 of [RFC9053]. This optional value is used to influence the key generation process. This specification does not mandate the use of a salt value. If the salt is public and carried in the message, then the "salt" algorithm header parameter MUST be used. The purpose of the salt is to provide extra randomness in the KDF context. If the salt is sent in the 'salt' algorithm header parameter, then the receiver MUST be able to process the salt and MUST pass it into the key derivation function. For more information about the salt, see [RFC5869] and NIST SP800-56 [SP800-56].

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

6.2.5. Example

This example uses the following parameters:

*Algorithm for payload encryption: AES-GCM-128

-IV: h'3517CE3E78AC2BF3D1CDFDAF955E8600'

*Algorithm for content key distribution: ECDH-ES + A128KW
**KEK COSE_Key (Receiver's Private Key):**

- kty / 1: 2 / EC2/
- crv / -1: 1 / P-256/
- x / -2:
  h'5886CD61DD875862E5AAA820E7A15274C968A9BC96048DDCACE32F50C3651BA3'
- y / -3:
  h'9EED8125E932CD60C0EAD3650D0A485CF726D378D1B016ED4298B2961E258F1B'
- d / -4:
  h'60FE6DD6D85D5740A5349B6F91267EEAC5BA81B8CB53EE249E4B4EB102C476B3'

*KID: 'kid-2'*

**KDF Context**

- Algorithm ID: -3 (A128KW)
- SuppPubInfo

  okeyDataLength: 128
  oprotected = << { / alg / 1: -3 / A128KW / } >>
  oother = 'SUIT Payload Encryption'

*Plaintext: "This is a real firmware image."

-in hex:
  546869732069732061207265616E97626777726520696672656469766520737472696365

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

D8608443A10101A10550DAE6138E0DC55F4322BE38BDAB9DC68F6818344
A101381CA220401022001215820FF6E266DABAF51B7207569E31CF72646
183E94CE64FCD8695AD9A505AEDFA2258205FBC4A2984450B3AC22AB
30C7F7004B59DB88D60D79977349FA0124B65989504456B69642D325818
B0E21628283F3E409F8158D8FFCA567F340E379AC39E49C9

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

This section summarizes the steps taken for content encryption, which applies to both content key distribution methods.

For use with AEAD ciphers, the COSE specification requires a consistent byte stream for the authenticated data structure to be created. This structure is shown in Figure 10 and is defined in Section 5.3 of [RFC9052].
7. Firmware Updates on IoT Devices with Flash Memory

Note: This section is specific to firmware images and does not apply to generic software, configuration data, and machine learning models.

Flash memory on microcontrollers is a type of non-volatile memory that erases data in units called blocks, pages, or sectors and re-writes data at the byte level (often 4-bytes) or larger units. Flash memory is furthermore segmented into different memory regions, which store the bootloader, different versions of firmware images (in so-called slots), and configuration data. Figure 11 shows an example layout of a microcontroller flash area. The primary slot typically contains the firmware image to be executed by the bootloader, which is a common deployment on devices that do not offer the concept of position independent code. Position independent code is not a feature frequently found in real-time operating systems used on microcontrollers. There are many flavors of embedded devices, the market is large and fragmented. Hence, it is likely that some implementations and deployments implement their firmware update procedure different than described below. On a positive note, the SUIT manifest allows different deployment scenarios to be supported easily thanks to the "scripting" functionality offered by the commands.
When the encrypted firmware image has been transferred to the device, it will typically be stored in a staging area, in the secondary slot in our example.

At the next boot, the bootloader will recognize a new firmware image in the secondary slot and will start decrypting the downloaded image sector-by-sector and will swap it with the image found in the primary slot.

The swap will only take place after the signature on the plaintext is verified. Note that the plaintext firmware image is available in the primary slot only after the swap has been completed, unless "dummy decrypt" is used to compute the hash over the plaintext prior to executing the decrypt operation during a swap. Dummy decryption here refers to the decryption of the firmware image found in the secondary slot sector-by-sector and computing a rolling hash over the resulting plaintext firmware image (also sector-by-sector) without performing the swap operation. While there are performance optimizations possible, such as conveying hashes for each sector in the manifest rather than a hash of the entire firmware image, such optimizations are not described in this specification.

This approach of swapping the newly downloaded image with the previously valid image requires two slots to allow the update to be reversed in case the newly obtained firmware image fails to boot. This approach adds robustness to the firmware update procedure.

Since the image in primary slot is available in cleartext, it may need to be re-encrypted before copying it to the secondary slot. This may be necessary when the secondary slot has different access permissions or when the staging area is located in off-chip flash memory and is therefore more vulnerable to physical attacks. Note that this description assumes that the processor does not execute encrypted memory by using on-the-fly decryption in hardware.
The ability to restart an interrupted firmware update is often a requirement for low-end IoT devices. To fulfill this requirement it is necessary to chunk a firmware image into sectors and to encrypt each sector individually using a cipher that does not increase the size of the resulting ciphertext (i.e., by not adding an authentication tag after each encrypted block).

When an update gets aborted while the bootloader is decrypting the newly obtained image and swapping the sectors, the bootloader can restart where it left off. This technique offers robustness and better performance.

For this purpose, ciphers without integrity protection are used to encrypt the firmware image. Integrity protection of the firmware
image MUST be provided and the suit-parameter-image-digest, defined in Section 8.4.8.6 of [I-D.ietf-suit-manifest], MUST be used.

[I-D.ietf-cose-aes-ctr-and-cbc] registers AES Counter (AES-CTR) mode and AES Cipher Block Chaining (AES-CBC) ciphers that do not offer integrity protection. These ciphers are useful for use cases that require firmware encryption on IoT devices. For many other use cases where software packages, configuration information or personalization data need to be encrypted, the use of Authenticated Encryption with Associated Data (AEAD) ciphers is RECOMMENDED.

The following sub-sections provide further information about the initialization vector (IV) selection for use with AES-CBC and AES-CTR in the firmware encryption context. An IV MUST NOT be re-used when the same key is used. For this application, the IVs are not random but rather based on the slot/sector-combination in flash memory. The text below assumes that the block-size of AES is (much) smaller than the sector size. The typical sector-size of flash memory is in the order of KiB. Hence, multiple AES blocks need to be decrypted until an entire sector is completed.

7.1. AES-CBC

In AES-CBC, a single IV is used for encryption of firmware belonging to a single sector, since individual AES blocks are chained together, as shown in Figure 12. The numbering of sectors in a slot MUST start with zero (0) and MUST increase by one with every sector till the end of the slot is reached. The IV follows this numbering.

For 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 AES-128-CBC uses an AES-block size of 128 bit (16 bytes). Hence, sector 0 needs 4096/16=256 AES-128-CBC operations using IV 0. If the firmware image fills the entire slot, then that slot contains 16 sectors, i.e. IVs ranging from 0 to 15.
7.1.1. AES-KW + AES-CBC Example

This example uses the following parameters:

*Algorithm for payload encryption: A128-CBC

-IV: h'93702C81590F845D9EC866CCAC767BD1'

*Algorithm id for key wrap: A128KW

*KEK COSE_Key (Secret Key):

  -/ kty / 1: 4 / Symmetric /

  -/ k / -1: 'aaaaaaaaaaaaaaaaa'

*KID: 'kid-1'

*Plaintext: "This is a real firmware image."

-in hex:

546869732069732061207265616420696D6167652E
The COSE_Encrypt structure, in hex format, is (with a line break inserted):

D8608445A10139FFFFA1055093702C81590F845D9EC866CCAC767BD1F681
8341A0A2012204456B69642D315818E198FF269626EC43299D33586FC7B2
646B13292261160422

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

96([
   / protected: / << {
      / alg / 1: -65531 / A128CBC /
   } >>,
   / unprotected: / {
      / IV / 5: h'93702C81590F845D9EC866CCAC767BD1'
   },
   / payload: / null / detached ciphertext /,
   / recipients: / [
      [
         / protected: / << {
            } >>,
         / unprotected: / {
            / alg / 1: -3 / A128KW /,
            / kid / 4: 'kid-1'
         },
         / payload: /
            h'E198FF269626EC43299D33586FC7B2646B13292261160422'
            / CEK encrypted with KEK /
         ]
      ]
   ])

Figure 13: COSE_Encrypt Example for AES Key Wrap

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

9C09156CF4ACE0401086D98586E0B09FA5B5CF78F2BCCBF6C914DDB42BF0
E21E

7.1.2. ES-DH + AES-CBC Example

This example uses the following parameters:

*Algorithm for payload encryption: AES-CBC-128
   -IV: h'DAE613B2E0DC55F4322BE38BDBA9DC68'
*Algorithm for content key distribution: ECDH-ES + A128KW
*KEK COSE Key (Receiver's Private Key):*

- kty / 1: 2 / EC2 / 
- crv / -1: 1 / P-256 / 
- x / -2: h'5886CD61DD875862E5AAA820E7A15274C968A9BC96048DDCACE32F50C3651BA3'  
- y / -3: h'9EED8125E932CD60C0EAD3650D0A485CF726D378D1B016ED4298B2961E258F1B'  
- d / -4: h'60FE6DD6D85D5740A5349B6F91267EEAC5BA81B8CB53EE249E4B4EB102C476B3'  

*KDF Context*

- Algorithm ID: -3 (A128KW)  
- SuppPubInfo  
  okeyDataLength: 128  
  oprotected = << { / alg / 1: -3 / A128KW / } >>  
  oother = 'SUIT Payload Encryption'  

*Plaintext: "This is a real firmware image."*

- in hex:  
  546869732069732061207265776172652E 

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

D8608445A10139FFFAA10550DAE613B2E0DC55F4322BE38BDBA9DC68F681 
8344A101381CA120A401022001215820BE4FB61E951F0378F453B616C91D 
DD29EF00CD36A0957F9686AC7D693E8880A32258200DF7D9253B34FC0247 
A11E15F20281680447896BE91B65E1D1E06D3E5FC57B0A581832970E4511 
E3709F118B838C5ED62A05A1421558CE56C20 

The resulting COSE_Encrypt structure in a diagnostic format is shown in Figure 14. Note that the COSE_Encrypt structure also needs to be protected and authenticated by the suit-authentication-wrapper, which is not shown below.
The encrypted payload (with a line feed added) was:

```
9C09156CF4ACE0401086D98586E0B09FA5B5CF78F2BCCBF6C914DB42BF0
E21E
```

7.2. AES-CTR

Unlike AES-CBC, AES-CTR uses an IV per AES operation, as shown in Figure 15. Hence, when an image is encrypted using AES-CTR-128 or AES-CTR-256, the IV MUST start with zero (0) and MUST be incremented by one for each 16-byte plaintext block within the entire slot.

Using the previous example with a slot size of 64 KiB, the sector size 4096 bytes and the AES plaintext block size of 16 byte requires IVs from 0 to 255 in the first sector and 16 * 256 IVs for the remaining sectors in the slot.
7.2.1. AES-KW + AES-CTR Example

This example uses the following parameters:

*Algorithm for payload encryption: AES-CTR-128

-IV: h'93702C81590F845D9EC866CCAC767BD1'

*Algorithm id for key wrap: A128KW

*KEK COSE.Key (Secret Key):

- / kty / 1: 4 / Symmetric /
- / k / -1: 'aaaaaaaaaaaaaaaaa'

*KID: 'kid-1'

*Plaintext: "This is a real firmware image."

-in hex:
546869732069732072656167652E

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

D8608445A10139FFDA1055093702C81590F845D9EC866CCAC767BD1F681
8341A0A2012204456B69642D315818CE34035CE5C2E2666E46D4C131FC56
1DD190A6D26CFA1990
The resulting COSE_Encrypt structure in a diagnostic format is shown in Figure 16.

```
96(
    / protected: / << {
        / alg / 1: -65534 / A128CTR /
    } >>,
    / unprotected: / {
        / IV / 5: h'93702C81590F845D9EC866CCAC767BD1'
    },
    / payload: / null / detached ciphertext /,
    / recipients: / [
        / protected: / << {
        / alg / 1: A128KW /,
        / kid / 4: 'kid-1'
        } ],
        / payload: / h'CE34035C5E5C2E2666E46D4C131FC561DD190A6D26CFA1990' / CEK encrypted with KEK /
    ]
)"

Figure 16: COSE_Encrypt Example for AES Key Wrap
```

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

B74188DE68CD904873C7D4AB265A7F5608AC63F68DBCE7773ADCB38DA07

### 7.2.2. ES-DH + AES-CTR Example

This example uses the following parameters:

*Algorithm for payload encryption: AES-CTR-128

- IV: h'DAE613B2E0DC55F4322BE38BDBA9DC68'

*Algorithm for content key distribution: ECDH-ES + A128KW

*KEK COSE_Key (Receiver's Private Key):

- / kty / 1: 2 / EC2 /

- / crv / -1: 1 / P-256 /
-/ x / -2:
  h'5886CD61DD875862E5AAA820E7A15274C968A9BC96048DDCACE32F50C3651BA3' 

-/ y / -3:
  h'9EED8125E932CD60C0EAD3650D0A485CF726D378D1B016ED4298B2961E258F1B'

-/ d / -4:
  h'60FE6DD6D85D5740A5349B6F91267EEAC5BA81B8CB53EE249E4B4EB102C476B3'

*KDF Context*

-Algorithm ID: -3 (A128KW)

-SuppPubInfo

  okeyDataLength: 128

  oprotected = << { / alg / 1: -3 / A128KW / } >>

  oother = 'SUIT Payload Encryption'

*Plaintext: "This is a real firmware image."

-in hex:
  54686973206972657374656167696E6976652E

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

D8608445A10139FFDA10550DAE613B2E0DC55F4322BE38BDBA9DC68F681
8344A101381CA120A401022001215820CBB2AA8EEAF7E260F02AD5C29E4608
939A6487E2172D239562819E5F52C6D34674225820FB6F870883864A9B14
828501FE47A4E9282A7065CD1C234E8A3BCE46F49A1A9F581818DC08054C
462C7977EE90FA2B9DE62FF6D42E398CEE8168

The resulting COSE_Encrypt structure in a diagnostic format is shown in Figure 17. Note that the COSE_Encrypt structure also needs to be protected and authenticated by the suit-authentication-wrapper, which is not shown below.
7.3. Battery Exhaustion Attacks

The use of flash memory opens up for another attack. An attacker may swap detached payloads and thereby force the device to process a wrong payload. While this attack will be detected, a device may have performed energy-expensive flash operations already. These operations may reduce the lifetime of devices when they are battery powered IoT devices. See Section 7 for further discussion about IoT devices using flash memory.

Including the digest of the encrypted payload allows the device to detect a battery exhaustion attack before energy consuming decryption and flash operations took place. Including the digest of the plaintext payload is adequate when battery exhaustion attacks are not a concern.
8. Complete Examples

The following manifests exemplify how to deliver encrypted payload and its encryption info to devices.

HMAC-256 MAC are added in AES-KW examples using the following secret key:

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

ES-DH examples are signed using the following ECDSA secp256r1 key:

-----BEGIN PRIVATE KEY-----
MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgApZYjZCUGLM50VBC
CjYStX+09jGmnyJPpDLTz/hiX0hRANCAASEEloEarguqq9JhVxie7NomvqqL8Rtv
P+bitWchdvArTsFKtscYExwKNtrNHXi9OB3N+wnAUtszmR23M4tKiW
-----END PRIVATE KEY-----

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

-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEhJaBGq4LqqvSYVcYnuzaJr6qi/Eb
bz/m4rVlnIXbwK07HypLbAmBMcCjbaZR14vTgdzfsJwFLbMSktz0LSolg==
-----END PUBLIC KEY-----

Each example uses SHA-256 as the digest function.

8.1. AES Key Wrap Example with Write Directive

The following SUIT manifest requests a parser to authenticate the manifest with COSE_Mac0 HMAC256, and to write and to decrypt the encrypted payload into a component with the suit-directive-write directive.

The SUIT manifest in diagnostic notation (with line breaks added for readability) is shown here:
/ CEK encrypted with KEK /

],

},

/ decrypt encrypted firmware /
/ directive-write / 18, 15
/ consumes the SUIT_Encryption_Info above /

]} >>
}
} >>
})
In hex format, the SUIT manifest is this:

D86BA2025853825824822F5820536EC695E423342FF57FA89B3E3C12C0F9
257992F7D96F017281782D2DF1C50F582AD18443A10105A0F658203B7057
1169B0FEE5E6220BF86E5E973F7F32875495908EDAA91EC9948CA44B2903
589DA4010102010357A102818152706C1696E746578742D66697266D7761
726511587C8414A212582E2F59C3A34D9570FB99A5382E66466A3221A8AD
85CE508BA306FB431A60EFA5AAA078355070205A4B196832DF1F135843
D8608443A10101A10550F14AAB9D81D51F7AD943FE87AF4F70CDF6818341
A0A2012204456B69642D31581875603FFC9518D794713C8CA8A115A7FB32
565A6D59534D62120F

8.2. AES Key Wrap Example with Fetch + Copy Directives

The following SUIT manifest requests a parser to fetch the encrypted payload and to stores it. Then, the payload is decrypted and stored into another component with the suit-directive-copy directive. This approach works well on constrained devices with execute-in-place flash memory.

The SUIT manifest in diagnostic notation (with line breaks added for readability) is shown here:
/ SUIT_Envelope_Tagged / 107({
  / authentication-wrapper / 2: << [
    << [
      / digest-algorithm-id: / -16 / SHA256 /,
      / digest-bytes: / h'AAB6A7868C4E43D5983BDE019EF22779
                   21F6F8EF1FCAF949C3CA97255BED2CD30'
    ] >>,
    << / COSE_Mac0_Tagged / 17([[
      / protected: / << {
        / algorithm-id / 1: 5 / HMAC256 /
      } >>,
      / unprotected: / {},
      / payload: / null,
      / tag: / h'93B48774A5D0421ED6FB5EBF890A284C
               DAC7816C8048BF47EE7FA7FF3BC02C3'
    ]) >>,
  ] >>,
  / manifest / 3: << {
    / manifest-version / 1: 1,
    / manifest-sequence-number / 2: 1,
    / common / 3: << {
      / components / 2: [
        'plaintext-firmware',
        'encrypted-firmware'
      ]
    } >>,
  } >>,
  / install / 17: << []
    / fetch encrypted firmware /
    / directive-set-component-index / 12, 1 / ['encrypted-firmware'] /
    / directive-override-parameters / 20, {
      / parameter-image-size / 14: 46,
      / parameter-uri / 21: "https://example.com/encrypted-firmware"
    },
    / directive-fetch / 21, 15,
    / decrypt encrypted firmware /
    / directive-set-component-index / 12, 0 / ['plaintext-firmware'] /
    / directive-override-parameters / 20, {
      / parameter-encryption-info / 19: << 96([{
        / protected: / << {
          / alg / 1: 1 / AES-GCM-128 /
        } >>,
        / unprotected: / {
          / IV / 5: h'F14AAB9D81D51F7AD943FE87AF4F70CD'
        },
        / payload: / null / detached ciphertext /,
        / recipients: / [
          / protected: / << {

```
} >>,
/ unprotected: / {
    / alg / 1: -3 / A128KW /,
    / kid / 4: 'kid-1'
},
/ payload: /
    h'75603FFC9518D794713C8CA8A115A7FB32565A6D59534D62'
    / CEK encrypted with KEK /
] ]
}) >>,
/ parameter-source-component / 22: 1 / ['encrypted-firmware'] / }
, / directive-copy / 22, 15 / consumes the SUIT_Encryption_Info above ] >>
} >>
})
In hex format, the SUIT manifest is this:

D86BA2025853825824822F5820AA6A7868C4E43D5983BDE019EF2277921
F6F8EF1FCAF9403CA97255BED2CD30582AD18443A10105A0F6582093B4B7
74A5D0421ED6FB5EBF890A284CDAC7816CBC048BF47EE7FA7FF3BC02C303
58B7A40101020103582BA1528152706C61696E746578742D669726D77
6172658152656E637279707465642D669726D77617265158818C0C0114
A20E182E157826687470733A2F2F6578616D706C652E63662F666E63
7279707465642D669726D77617265158818C0C0114A2135843D860843A101
01A10550F14AAB9D81D51F7AD943FE87AF4F70C94318341A0A201220445
6B69642D31581875603FC9518794713C8CA8A115A7FB32565A6D59534D
621601160F

8.3. ES-DH Example with Write + Copy Directives

The following SUIT manifest requests a parser to authenticate the manifest with COSE_Sign1 ES256, and to write and to decrypt the encrypted payload into a component with the suit-directive-write directive.

The SUIT manifest in diagnostic notation (with line breaks added for readability) is shown here:
/ SUITEnvelope_Tagged / 107{
   / authentication-wrapper / 2: << [ 
      << [ 
         / digest-algorithm-id: / -16 / SHA256 /,
         / digest-bytes: / h'CEF034223D7F2C39D676876995B4ED4E8221AC5BF184B6606EE62C41C149C266'
      ] >>,
      << / COSE_Sign1_Tagged / 18([ 
         / protected: / << { 
            / algorithm-id / 1: -7 / ES256 / 
         } >>,
         / unprotected: / {},
         / payload: / null,
         / signature: / h'65E59AAB8A35BDE9547458316D1C769FDB2CEA304C9FB6151E5C8A88A002A292C5B8C63C81B5AC0AE31948B610834E12CBDBB2753EA221544B6733076A92EE20'
      ]) >>,
   ] >>,
   / manifest / 3: << { 
      / manifest-version / 1: 1,
      / manifest-sequence-number / 2: 1,
      / common / 3: << { 
         / components / 2: [ 'decrypted-firmware' ]
      }
   }
}

/ install / 17: << [ 
   / directive-set-component-index / 12, 0 / ['plaintext-firmware'] /,
   / directive-override-parameters / 20, { 
      / parameter-content / 18: 
         h'344FA2D5AD2F43F6F363DA6FF2C337FE69E33E3D63714D23985BF024998E8B231D45C37B245DA3611C160CC511',
      / parameter-encryption-info / 19: << 96{ 
         / alg / 1: 1 / AES-GCM-128 / 
      } >>,
      / unprotected: / { 
         / IV / 5: h'DAE613B2E0DC55F4322BE38BDBA90C6B' 
      },
      / payload: / null / detached ciphertext /,
      / recipients: / [ 
         / protected: / << { 
            / alg / 1: -29 / ECDH-ES + A128KW / 
         } >>,
         / unprotected: / { 
            / alg / 1: -29 / ECDH-ES + A128KW / 
         } >>,
         / payload: / null / detached ciphertext /,
         / recipients: / [ 
            / alg / 1: 1 / AES-GCM-128 / 
         ] >>
      },
   ] >>
}

/ ephemeral key / -1: {
 / kty / 1: 2 / EC2 /,
 / crv / -1: 1 / P-256 /,
 / x / -2: h'FF6E266DABAF51B7207569E31CF72646
  183E94CEE64FCDC8695AD9A505AEFDEA',
 / y / -3: h'5FBC4A29844450B3AC22AB30C7F7004B
  B59D8BD60D7997734A9FA0124B650895'
},
 / kid / 4: 'kid-2'
},
 / payload: /
 h'B0E21628283F3E409F8158D8FFCA567F340E379AC39E49C9'
 / CEK encrypted with KEK /
 ]
]
})  >>
},
 / directive-write / 18, 15
 / consumes the SUIT_Encryption_Info above /
]  >>
}  >>
})
In hex format, the SUIT manifest is this:

D86BA205873825824822F5820CEF034223D7F2C39D676876995B4ED4E82
21AC5BF184B6606EE62C41C149C266584AD28443A10126A0F6584065E59A
AB8A35BDE9547458316D1C769FFB2CEA304C9FB6151E5C8A88A002A292C5
B8C63C81B5AC0AE319482610834E12CBDBB2753EA221544B6733076A92EE
200358ECA4010102010357A102818152646563727977465642D6669726D
776172651158CB860C0014A212582E344FA2D5AD2F43F6F363DA6FF2C337
FE69E33E3D63714D23985BF02499EB0E8B231D45C378245DA3611C160CC5
11135890D8608443A10101A10550DAE613B2E0DC55F4322BE38BDBA9DC68
F6818344A101381CA220A401022001215820FF6E266DABAF51B7207569E3
1CF72646183E94CEE64FCDC8695AD9A505AEFDEA2258285FBC4A29844450
B3AC22AB30C7F7004BB59D8BD60D7997734A9FA0124B65089504456B6964
2D32581880E21628283F3E409F8158D8FFCA567F340E379AC39E49C9120F

8.4. ES-DH Example with Dependency

The following SUIT manifest requests a parser to resolve the delegation chain and dependency respectively. The parser validates the COSE_Key in the suit-delegation section using the key above, and then dynamically trusts it. The dependency manifest is embedded as an integrated-dependency and referred by uri "#dependency-manifest".

The SUIT manifest in diagnostic notation (with line breaks added for readability) is shown here:
SUIT_Envelope_Tagged / 107{
    delegation / 1: << [
        NOTE: signed by trust anchor /
        18([
            protected: / << {
                alg / 1: -7 / ES256 /
            } >>,
            unprotected / {
                payload: / << {
                    cnf / 8: {
                        NOTE: public key of delegated authority /
                        COSE_Key / 1: {
                            kty / 1: 2 / EC2 /,
                            crv / -1: 1 / P-256 /,
                            x / -2: h'0E908AA8F066DB1F084E0C3652C63952BD99F2A5BDB22F9E01367AAD03ABA68B',
                            y / -3: h'77DA1BD8AC4F0CB490BA210648BF79AB164D49AD3551D71D314B2749EE42D29A'
                        }
                    }
                }
            } >>,
            signature: / h'FB2D5ACF66B9C8573CE92E13BF88D113F798715CC10B5A0010B11925C155E7245A64E131073B87AC50CAC71650A21315B82D06CA2298CD1A95519AAE4C4B5315'
        ] >>,
        ] >>,
        authentication-wrapper / 2: << [
        ] >>,
        / COSE_Sign1_Tagged / 18([
            protected: / << {
                algorithm-id / 1: -7 / ES256 /
            } >>,
            unprotected: / {},
            payload: / null,
            signature: / h'DF493BDBF167EFFB40593C5910D33B66429721467DF05800EA66A88B91729CD51007981F151FC324745FF43E6F75AA5197DD5EC4AA6BCEFCE43E4B1E35C948E'
        ])) >>
    ]}
manifest / 3: << {
  manifest-version / 1: 1,
  manifest-sequence-number / 2: 1,
  common / 3: << {
    dependencies / 1: {
      component-index / 1: {
        dependency-prefix / 1: ['dependency-manifest.suit']
      }
    }
  },
  components / 2: [
    ['decrypted-firmware']
  ]
}

install / 17: << [
  NOTE: set SUIT_Encryption_Info /
  directive-set-component-index / 12, 0
  ['decrypted-firmware'] /
  directive-override-parameters / 20, {
    parameter-content / 18: h'344FA2D5AD2F43F6F363DA6FF2C337FE69E33E3D63714D
    23985BF02499EB0E8B231D45C378245DA3611C160CC511',
    parameter-encryption-info / 19: << 96([{
      protected: / << {
        alg / 1: 1 / AES-GCM-128 /
      } >>,
      unprotected: / {
        IV / 5: h'DAE613B2E0DC55F4322BE38BDBA9DC68'
      },
      payload: / 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'FF6E266DABA5F51B7207569E31CF72646
            183E94CEE64FCDC8695AD9A505AEFDEA',
            y / -3: h'5FBC4A29B44450B3AC22AB30C7F7004B
            B59D8BD60D7997734A9FA0124B650895'
          }
        }
      }
    }
  ]}
/ kid / 4: 'kid-2'
},
/ payload: /
  h'B0E21628283F3E409F8158D8FFCA567F340E379AC39E49C9'
  / CEK encrypted with KEK /
]
],
]) >>
}
,
/ NOTE: call dependency-manifest /
/ directive-set-component-index / 12, 1
/ ['dependenty-manifest.suit'] /,
/ directive-override-parameters / 20, {
  / parameter-image-digest / 3: << [
    / algorithm-id / -16 / SHA256 /,
    / digest-bytes / h'1051324059C5193317CAC9A099BBC0B6
      AFB56184C04277F566A3A4131F4A1C25'
  ] >>,
  / parameter-image-size / 14: 247,
  / parameter-uri / 21: "#dependency-manifest"
},
/ directive-fetch / 21, 15,
/ condition-dependency-integrity / 7, 15,
/ directive-process-dependency / 11, 15
] >>
} >>,
"#dependency-manifest": <<
/ SUIT_Envelope_Tagged / 107({
  / authentication-wrapper / 2: << [
    << [
      / digest-algorithm-id / -16 / SHA256 /,
      / digest-bytes / h'1051324059C5193317CAC9A099BBC0B6
        AFB56184C04277F566A3A4131F4A1C25'
    ] >>,
    << / COSE_Sign1_Tagged / 18([
      / protected: / << {
        / algorithm-id / 1: -7 / ES256 /
      } >>,
      / unprotected: / {},
      / payload: / null,
      / signature: / h'55990F3745DC4F200FF946643A6DE30D
        DCE57B080B7D68DE9896D8190B9A63E2
        D60E7C3D9693B67221AA6D07BBF0AB45
        314C236827A242C22B5E688DDC467269'
    ])) >>
  ] >>,
  / manifest / 3: << {

/ manifest-version / 1: 1,
/ manifest-sequence-number / 2: 1,
/ common / 3: << {
/ components / 2: [
[ 'decrypted-firmware' ]
],
/ shared-sequence / 4: << [
/ directive-set-component-index / 12, 0
/ [ 'decrypted-firmware' ] /,
/ directive-override-parameters / 20, {
/ parameter-image-digest / 3: << [
/ algorithm-id / -16 / SHA256 /,
/ digest-bytes / h'36921488FE6680712F734E11F58D87EEB66D4B21A8A1AD3441060814DA16D50F'
] >>,
/ parameter-image-size / 14: 30
} ] >>
} >>,
/ manifest-component-id / 5: [ 'dependency-manifest.suit' ],
/ validate / 7: << [
/ condition-image-match / 3, 15
] >>,
/ install / 17: << [
/ directive-set-component-index / 12, 0
/ [ 'decrypted-firmware' ] /,
/ directive-write / 18, 15
/ consumes the SUIT_Encryption_Info set by dependent /,
/ condition-image-match / 3, 15
/ check the integrity of the decrypted payload /
] >>
} >>
})

}}
In hex format, the SUIT manifest is this:

```
D06BA401589E8181589AD28443A10126A0584FA108A101A40102020012158
200E908A08F066DB1F084E0C3652C63952BD99F2A5BB22F9E01367A0D3
ABA68B22582077DA1BD8AC4B90BA2106488F79AB164D49AD3551D71D
314B2749EE42D29A5840FB2D5ACF66BC8573CE92E13BF6B1D13F798715C
C10B5A0108111925C155E7245A64E131073B87AC50CAC71650A21315B8B2D
06CA2298CD15919A1AE4C4B5315025873825824822F58206A19DF42E7B4
047D2F54046019AE3ED43A8ACC467AC16576B176D8E633042D2584AD284
43A10126A0F65840DF493B8F167EF8F40593C5910D33B66429721467DF0
5800E66A88B91729DC51007981F516C32747FF4F36F75AFA5197DD5EC
4AA6BCEFE43E4B1E35C948E03590170A501010201035837A201A101
81581864657065E64656E63792D6D161E6966657374E7375697402181
526465637279707465642D66697276771762505817564675065E64656E
742D6616E6966657374E73756974115901138E0C0014A212582E344FA2
D5AD2F43F6F363DA6FF2C337FE69E33E3D63714D23985BF02499EB08B23
1D45C378254DA3611C160CC511135908D6808443A10101A0550DAE13B2
E0D5DF4322BE38DBA9DC68F6818344A101381CA22A40102001215820
FF6E266DABAF51B7207569E31CF72646183E94CEE64FCD86959A505AE
FDEA2258205FB4A29844450B3AC22AB30C7F7004B59DB6D07997734A
9FA0124B650895045666462D325818BE021622823F3E499F818DBBFCA
567F346ECEF439CE49C90C0114A305824822F58201051324059C5193317
CAC9A099BBCB06AFB56184C04277F566A3A4131F4A1C250E18F715742364
657065E64656EE63792D6D161E6966657374150F070F0B742364657065
6E64656E63792D6D161E696665737458F7D86BA202587382582482F5820
1051324059C5193317CAC9A099BBCB06AFB56184C04277F566A3A4131F4A
1C25584AD28443A10126A0F6584055990F3745DC4F200FF946643A6DE30D
DCE57B800B7D6DE896E190B9A63E260DE7C3D9693867211A6D07BBF0
AB45314C236827A242CC2B56E56HD46726903587A60101020102035849A2
028B181526465637279707465642D66697276771762504582F840C0014A2
035824822F582036921488F6680712F734E11F58D87EEB6D4B1A8A1AD
3441060814DA16D50FE181E058158B1864657065E64656E63792D6D16E
6966657374E73756974074382030F1147860C00120F030F
```

9. Operational Considerations

The algorithms described in this document assume that the party performing payload encryption

*shares a key-encryption key (KEK) with the recipient (for use with the AES Key Wrap scheme), or*

*is in possession of the public key of the recipient (for use with ES-DH).*

Both cases require some upfront communication interaction to distribute these keys to the involved communication parties. This interaction may be provided by a device management protocol, as described in [RFC9019], or may be executed earlier in the lifecycle.
of the device, for example during manufacturing or during commissioning. In addition to the keying material key identifiers and algorithm information need to be provisioned. This specification places no requirements on the structure of the key identifier.

In some cases third party companies analyse binaries for known security vulnerabilities. With encrypted payloads, this type of analysis is prevented. Consequently, these third party companies either need to be given access to the plaintext binary before encryption or they need to become authorized recipients of the encrypted payloads. In either case, it is necessary to explicitly consider those third parties in the software supply chain when such a binary analysis is desired.

10. Security Considerations

This entire document is about security.

Note that it is good security practise to use different long-term keys for different purpose. For example, the KEK used with an AES-KW-based content key distribution method for encryption should be different from the long-term symmetric key used for authentication and integrity protection when uses with COSE_Mac0.

The design of this specification allows to use different long-term keys for encrypting payloads. For example, KEK_1 may be used with an AES-KW content key distribution method to encrypt a firmware image while KEK_2 would be used to encrypt configuration data. This approach reduces the attack surface since permissions of authors to these long-term keys may vary based on their privileges.

11. IANA Considerations

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

<table>
<thead>
<tr>
<th>Label</th>
<th>Name</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBD19</td>
<td>Encryption Info</td>
<td>Section 4</td>
</tr>
</tbody>
</table>

[Editor's Note: TBD19: Proposed 19]

12. References

12.1. Normative References

12.2. Informative References

[iana-suit] Internet Assigned Numbers Authority, "IANA SUIT Manifest Registry", 2023, <TBD>.


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]
; Define SUIT_Encryption_Info_\* as a subset of COSE_Encrypt

SUIT_Encryption_Info = #6.96(
    SUIT_Encryption_Info_AESKW .within COSE_Encrypt /
    SUIT_Encryption_Info_ESDH .within COSE_Encrypt)

SUIT_Encryption_Info_AESKW = [
    protected : bstr .cbor outer_header_map_protected,
    unprotected : outer_header_map_unprotected,
    ciphertext : bstr / nil,
    recipients : [ + COSE_recipient_AESKW .within COSE_recipient ]
]

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, ; algorithm identifier
    ? 4 => bstr, ; identifier of the recipient public key
    * label => values ; extension point
}

SUIT_Encryption_Info_ESDH = [
    protected : bstr .cbor outer_header_map_protected,
    unprotected : outer_header_map_unprotected,
    ciphertext : bstr / nil,
    recipients : [ + COSE_recipient_ESDH .within COSE_recipient ]
]

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, ; algorithm identifier
    * label => values ; extension point
}

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

; common definitions
outer_header_map_protected =
{
    1 => int, ; algorithm identifier
    * label => values ; extension point
}

outer_header_map_unprotected =
{
    5 => bstr, ; IV
    * label => values ; extension point
}

; Extends SUIT Manifest

$$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, Øyvind Rønningstad, Dave Thaler, Laurence Lundblade, Christian Amsüss, and Carsten Bormann for their review feedback. Finally, we would like to thank Dick Brooks for making us aware of the challenges encryption imposes on binary analysis.

Authors’ Addresses

Hannes Tschofenig
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