SUIT
Internet-Draft
Intended status: Standards Track
Expires: 27 February 2024

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

26 August 2023

Encrypted Payloads in SUIT Manifests
draft-ietf-suit-firmware-encryption-14

Abstract

This document specifies techniques for encrypting software, firmware 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 27 February 2024.

Copyright Notice

Table of Contents

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 ~~about the firmware for an IoT device~~, including where to find the firmware image, 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, the ~~firmware image~~payload may also be afforded confidentiality using encryption.

Encryption prevents third parties, including attackers, from gaining access to the firmware binary. Hackers typically need intimate knowledge of the target firmware 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.

The ~~firmware image~~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 algorithms for use with the IETF SUIT manifest, namely:

* Ephemeral-Static (ES) Diffie-Hellman (DH), and

* AES Key Wrap (AES-KW).

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

Our goal was to reduce the number of content key distribution algorithms and thereby increase interoperability between different SUIT manifest parser implementations.

While the original motivating use case of this document was firmware encryption, SUIT manifests may require payloads other than firmware images to experience confidentiality protection, such as

* software packages,

* personalization data,

* configuration data, and

Commented [DT1]: But it's not limited to firmware, nor to IoT devices. So I'd recommend deleting this phrase.

Commented [DT2]: s/firmware image/payload/ (since can be used just for personalization data for example).

Yes I now the text at the bottom of this section already talks about not just firmware, but I still think this reads better just using payload here, especially given the last sentence in this section.

Commented [DT3]: Same here and possibly other places

* machine learning models.

Hence, the term payload is used to generically refer to those objects that may be subject to encryption.

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 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 the firmware images. 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 ~~firmware image~~payload, does not know the recipients. It is important to note that the distribution system is far more than a file server.

Commented [DT4]: payload

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.

Finally, 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]

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 the firmware server and the device management infrastructure.

Commented [DT5]: Payload?

To apply encryption the sender (author) needs to know the recipient (device). 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 ES-DH the recipients must be provisioned with a public key (or a certificate) for verifying the digital signature covering the manifest.

With encryption the author cannot just create a manifest for the ~~firmware image~~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.

Commented [DT6]: payload

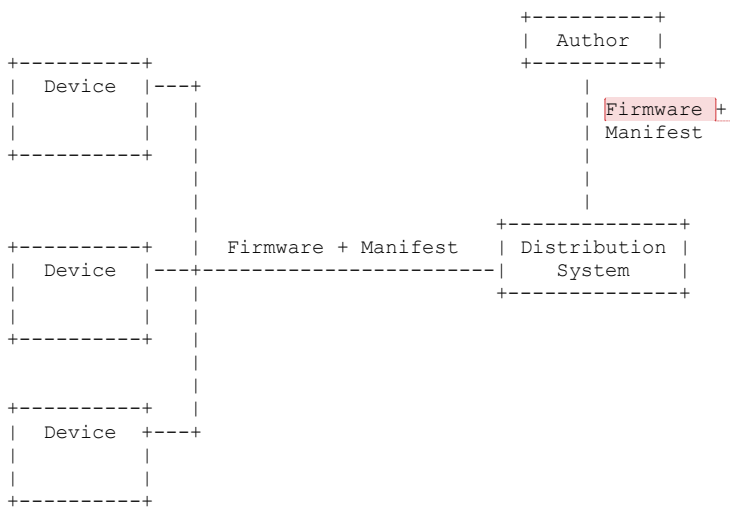


Figure 1: Firmware Encryption Architecture.

The author has several deployment options, namely

* 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.

* 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 flexiblity 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 and distributor are separate entities, then the author must delegate encryption rights to the distributor. By the principle of least privilege, this delegation should only grant the distributor decryption and re-encryption rights. There are two models:

Commented [DT7]: Payload?

Commented [DT8]: Payload

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 alternative is to use a two-manifest system, where 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.

Commented [DT9]: typo

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).

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 algorithms, the distribution system needs to know about the properties of the deployed devices. Mandating only a single content key distribution algorithm for a constrained device also reduces the code size.

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

suit-parameter-encryption-info   = [TBD1: Proposed 19]
```

Figure 2: CDDL of the SUIT_Parameters Extension.

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.

Commented [DT10]: Nit: always put a comma after "i.e." per Chicago Manual of Style. Multiple occurrences throughout

```
/ 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-override-parameters / 20, {
  / parameter-uri / 21: "http://example.com/encrypted.bin",
},
/ directive-fetch / 21, 15,
/ directive-set-component-index / 12, 0,
/ directive-override-parameters / 20, {
  / parameter-source-component / 22: 1,
  / parameter-encryption-info / 19: h'D86...1F0'
},
/ 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 a digital signature or a MAC of 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 authenticate the payload in the detached payload case a SUIT Digest Container with the digest of the encrypted and/or plaintext payload MUST be included in the manifest.

Another attack concerns battery exhaustion. An attacker may swap detached payloads and thereby force the device to process a wrong payload. While this attack will be detected the device has performed energy-expensive flash operations already. These operations may reduce the lifetime of devices when they are battery powered.

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.

Commented [DT11]: "may have". Don't assume all devices have "flash". IoT devices maybe, but this spec is not just for such devices.

Commented [DT12R11]: Also maybe provide a forward reference to section 7.

6. Content Key Distribution

The sub-sections below describe two content key distribution algorithms, namely AES Key Wrap (AES-KW) and Ephemeral-Static Diffie-Hellman (ES-DH). Other algorithms are supported by COSE and may be supported via enhancements to this specification.

When an encrypted firmware image 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.

* 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.

* KEK_i or CEK_i refers to the i-th instance of the KEK or CEK, respectively.

6.1. Content Key Distribution with AES Key Wrap

6.1.1. Introduction

The AES Key Wrap (AES-KW) algorithm is described in RFC 3394 [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.

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 embedded in the COSE_Encrypt.ciphertext if it is conveyed as detached content.

Commented [DT13]: payload

Commented [DT14]: Can't parse grammar. Maybe a missing "be"?

6.1.2. Deployment Options

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

* If all authorized recipients have access to the KEK, a single COSE_recipient structure contains the encrypted CEK. The sender executes the following steps:

```
Fetch KEK(*,S)
Generate CEK
ENC(CEK,KEK)
ENC(payload,CEK)
```

* 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:

```
Generate CEK
for i=1 to n {
   Fetch KEK_i(Ri, S)
   ENC(CEK, KEK_i)
}
ENC(payload,CEK)
```

* The third option is to use different CEKs encrypted with KEKs of authorized recipients. Assume there are n recipients with their unique KEKs - KEK_1(R1, S),..., KEK_n(Rn, S). The sender needs to make the following steps:

```
for i=1 to n {
   Fetch KEK_i(Ri, S)
   Generate CEK_i
   ENC(CEK_i, KEK_i)
   ENC(payload,CEK_i)
}
```

This approach is appropriate when no benefits can be gained from encrypting and transmitting payloads only once.

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 recipient public key
  * 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.

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 6 and defined in Section 5.3 of [RFC9052].

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

Figure 6: CDDL for Enc_structure Data Structure

This Enc_structure needs to be populated as follows:

The protected field in the Enc_structure from Figure 6 refers to the content of the protected field from the COSE_Encrypt structure. It is important to note that there are two protected fields shown in Figure 5:

* one in the COSE_Encrypt structure, and

* a second one in the COSE_recipient structure.

The value of the external_aad MUST be set to a null value (major type 7, value 22).

For use with ciphers that do not provide integrity protection, such as AES-CTR and AES-CBC (see [I-D.ietf-cose-aes-ctr-and-cbc]), the Enc_structure shown in Figure 6 MUST NOT be used because the Enc_structure represents the Additional Authenticated Data (AAD) byte string consumable only by AEAD ciphers. The protected header in the SUIT_Encryption_Info_AESKW structure MUST be a zero-length byte string.

Commented [DT15]: the

6.1.4. Example

This example uses the following parameters:

* Algorithm for payload encryption: AES-GCM-128

* Algorithm id for key wrap: A128KW

* IV: h'11D40BB56C3836AD44B39835B3ABC7FC'

* KEK: "aaaaaaaaaaaaaaaa"

* KID: "kid-1"

* Plaintext firmware (txt): "This is a real firmware image." (in hex): 546869732069732061207265616C206669726D7761726520696D6167652E

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

D8608443A10101A1054C26682306D4FB28CA01B43B80F68340A2012204456B69642D
315818AF09622B4F40F17930129D18D0CEA46F159C49E7F68B644D

The resulting COSE_Encrypt structure in a diagnostic format is shown in Figure 7.

```
96([
  / protected: / << {
    / alg / 1: 1 / AES-GCM-128 /
  } >>,
  / unprotected: / {
    / IV / 5: h'11D40BB56C3836AD44B39835B3ABC7FC'
  },
  / payload: / null / detached ciphertext /,
  / recipients: / [
    [
      / protected: / << {
      } >>,
      / unprotected: / {
        / alg / 1: -3 / A128KW /,
        / kid / 4: 'kid-1'
      },
      / payload: / h'E01F4443C88CA89DF93A9C7E6D79D1C9BC330757C7D2D75A'
        / CEK encrypted with KEK /
    ]
  ]
])
```

Figure 7: COSE_Encrypt Example for AES Key Wrap

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

```
CE9AB65E7591EE38669C4CCA7A58FA324C1A0DBFDBC2C7C057376AFB805D
660048310E8DAB045A2BE0A93F014FC9
```

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 10.

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 emphemeral key and an identifier for the recipients public key.

The steps taken by the sender are:

```
Generate CEK
for i=1 to n {
   Generate KEK_i(Ri, S) using ES-DH
   ENC(CEK, KEK_i)
}
ENC(payload,CEK)
```

* The alternative is to encrypt a payload with a different CEK for each recipient. Assume there are KEK_1(R1, S),..., KEK_n(Rn, S) have been generated for the different recipients using ES-DH. The following steps needs to be made by the sender:

```
for i=1 to n {
   Generate KEK_i(Ri, S) using ES-DH
   Generate CEK_i
   ENC(CEK_i, KEK_i)
   ENC(payload,CEK_i)
}
```

This results in n-manifests. This approach is useful when payloads contain information unique to a device. The encryption operation effectively becomes ENC(payload_i,CEK_i).

6.2.3. CDDL

The CDDL for the COSE_Encrypt_Tagged structure is shown in Figure 8. Only the minimum number of parameters are 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 8: CDDL for ES-DH-based Content Key Distribution

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 RFC 9053 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 9 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 -4), 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_pr_map field, which contains (among other fields) the content key distribution algorithm identifier.

Commented [DT16]: Nit: missing space

```
PartyInfoSender = (
    identity : nil,
    nonce : nil,
    other : nil
)

PartyInfoRecipient = (
    identity : nil,
    nonce : nil,
    other : nil
)

COSE_KDF_Context = [
    AlgorithmID : int,
    PartyUInfo : [ PartyInfoSender ],
    PartyVInfo : [ PartyInfoRecipient ],
    SuppPubInfo : [
        keyDataLength : uint,
        protected : bstr .cbor recipient_header_pr_map,
        other: bstr "SUIT Payload Encryption"
    ],
    SuppPrivInfo : bstr .size 0
]
```

Figure 9: 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 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 value is to provide extra randomness in the KDF context. If the salt sent in the "salt" algorithm header parameter, then the receiver MUST be able to process a salt and MUST pass it into the key derivation function. For more information about the salt value, see [RFC5869] and NIST SP800-56 [SP800-56].

Commented [DT17]: Grammar: "is"?

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

For use with ciphers that do not provide integrity protection, such as AES-CTR and AES-CBC (see [I-D.ietf-cose-aes-ctr-and-cbc]), the Enc_structure MUST NOT be used and the protected header in the SUIT_Encryption_Info_ESDH structure MUST be a zero-length byte string.

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

* KID: "kid-2"

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

* Firmware (hex):
  546869732069732061207265616C206669726D7761726520696D6167652E

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

```
D8608443A10101A1055035 17CE3E78AC2BF3D1CDFDAF955E8600F6818344
A101381CA220A401022001215820AAE9A733DEF11E9160A66BD81CC8215F
045ACAC3F8490C7749D58A627323624A22582008A7B88B7F00762BA0919C
A065ABF45C2A303B483E86D674E50B015122F8E51504456B69642D325818
0A44E77C3DBBB0780F2DB42C64FD325D18FBE13A25A9369D
```

The resulting COSE_Encrypt structure in a diagnostic format is shown in Figure 10. Note that the COSE_Encrypt structure also needs to protected by a COSE_Sign1, which is not shown below.

```
/ SUIT_Envelope_Tagged / 107({
  / authentication-wrapper / 2: << [
    << [
      / digest-algorithm-id: / -16 / SHA256 /,
      / digest-bytes: / h'4C56CA660A5D1414BC04C835025D52CC
                          A9AE6101202E127329AD2465B38A1C89'
    ] >>,
    << / COSE_Sign1_Tagged / 18([
      / protected: / << {
        / algorithm-id / 1: -7 / ES256 /
      } >>,
      / unprotected: / {},
      / payload: / null,
      / signature: /
        h'ACC8962628B78BF30DD74BDEEA9305D7
          3BFA302D82B280A7E2FCE8331C363F27
          9ECCABE920DA97F9074DF5B3B2AAD170
          9D844B8DE1D33F80FA99AC806B9778D0'
```

```
      ]) >>
    ] >>,
    / 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'B94272BD7C7E9A144D12CF46D9CEE6318753574A6F7808
              29B87911BE1CF2B24477BA4E7D1337541F308010088920',
          / parameter-encryption-info / 19: << 96([
            / protected: / << {
              / alg / 1: 1 / AES-GCM-128 /
            } >>,
            / unprotected: / {
              / IV / 5: h'3517CE3E78AC2BF3D1CDFDAF955E8600'
            },
            / 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'AAE9A733DEF11E9160A66BD81CC8215F
                                045ACAC3F8490C7749D58A627323624A',
                    / y / -3: h'08A7B88B7F00762BA0919CA065ABF45C
                                2A303B483E86D674E50B015122F8E515'
                  },
                  / kid / 4: 'kid-2'
                },
                / payload: /
                  h'0A44E77C3DBBB0780F2DB42C64FD325D18FBE13A25A9369D'
                  / CEK encrypted with KEK /
              ]
            ]
          ]) >>
        },
```

```
        / directive-write / 18, 15
          / consumes the SUIT_Encryption_Info above /
      ] >>
    } >>
})
```

Figure 10: COSE_Encrypt Example for ES-DH

The encrypted firmware (with a line feed added) was: ~~~
B94272BD7C7E9A144D12CF46D9CEE6318753574A6F780829B87911BE1CF2
B24477BA4E7D1337541F308010088920 ~~~

7. Firmware Updates on IoT Devices with Flash Memory

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). 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 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.

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 should 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.

Commented [DT18]: Maybe prefix this with "Example Use Case: "?

Commented [DT19]: grammar

Commented [DT20]: "will"? (for consistency with previous paragraph)

This approach of swapping the newly downloaded image with the previously valid image is often referred as A/B approach. A/B refers to the two slots involved. Two slots are used 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 re-encrypted it 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.

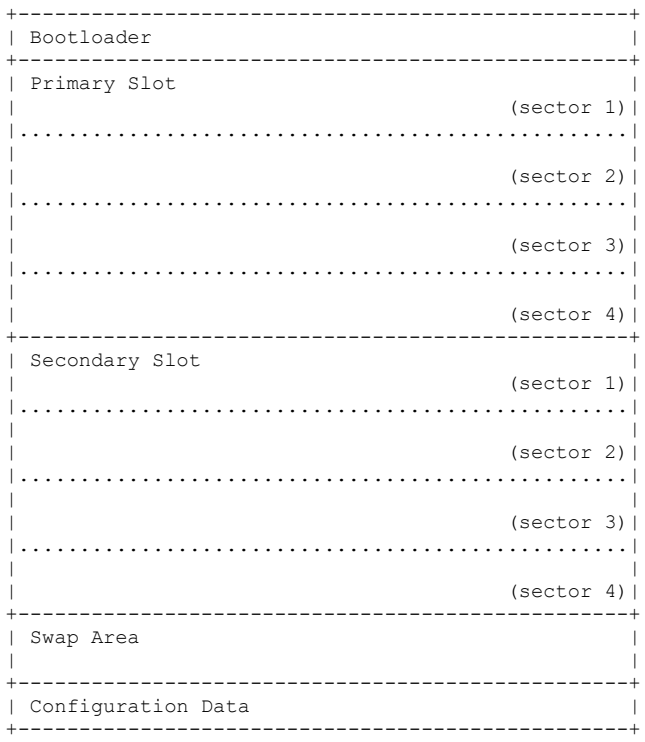


Figure 11: Example Flash Area Layout

Commented [DT21]: Grammar: "... to as the ..."

Commented [DT22]: Often, as in where? A/B testing - Wikipedia doesn't seem like the right concept. I do see wikipedia calls it Blue–green deployment - Wikipedia . If blue/green deployment isn't the right concept either, then at least provide an informative reference if it's "often" referred to as "A/B approach".

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 needs to be encrypted, the use of Authenticated Encryption with Associated Data (AEAD) ciphers is RECOMMENDED.

Commented [DT23]: Grammar: "need" (packages is plural)

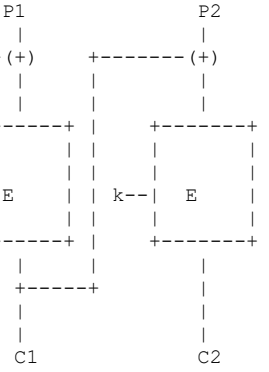
The following sub-sections provide further information about the initialization vector (IV) selection for use with AES-CBC and AES-CTR in the firmware encryption context. An IV MUST NOT be re-used when the same key is used. For this application, the IVs are not random but rather based on the slot/sector-combination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

Commented [DT24]: typo

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.



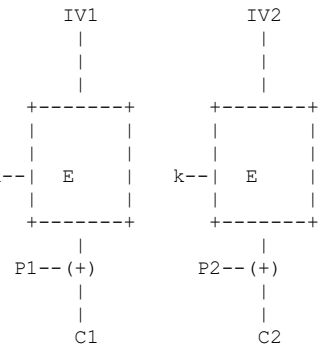
Legend:
Pi = Plaintext blocks
Ci = Ciphertext blocks
E = Encryption function
k = Symmetric key
(+) = XOR operation

Figure 12: AES-CBC Operation

7.2. AES-CTR

Unlike AES-CBC, AES-CTR uses an IV per AES operation, as shown in Figure 13. 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.



Legend:
See previous diagram.

Figure 13: AES-CTR Operation

8. Complete Examples

The following manifests examplify how to deliver encrypted firmware and its encryption info to devices.

Commented [DT25]: Typo. exemplify

The examples are signed using the following ECDSA secp256r1 key:

```
-----BEGIN PRIVATE KEY-----
MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgApZYjZCUGLM50VBC
CjYStX+09jGmnyJPrpDLTz/hiXOhRANCAASEloEarguqq9JhVxie7NomvqqL8Rtv
P+bitWWchdvArTsfKktsCYExwKNtrNHXi9OB3N+wnAUtszmR23M4tKiW
-----END PRIVATE KEY-----
```

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

```
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEhJaBGq4LqqvSYVcYnuzaJr6qi/Eb
bz/m4rVlnIXbwK07HypLbAmBMcCjbazR14vTgdzfsJwFLbM5kdtzOLSolg==
-----END PUBLIC KEY-----
```

Each example uses SHA-256 as the digest function.

8.1. AES Key Wrap Example with Write Directive

The following SUIT manifest requests a parser 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:

```
/ SUIT_Envelope_Tagged / 107({
  / authentication-wrapper / 2: << [
    << [
      / digest-algorithm-id: / -16 / SHA256 /,
      / digest-bytes: / h'5DEFDDB7F175FA20778FFE24BE7B9C36
                          9BD8ED06AA4654F28794CD134CDBA932'
    ] >>,
    << / COSE_Sign1_Tagged / 18([
      / protected: / << {
        / algorithm-id / 1: -7 / ES256 /
      } >>,
      / unprotected: / {},
      / payload: / null,
      / signature: / h'4C4A5FB50738699649BA439237D20ADC
                       ADD6EC634A800A8E093733FC1C64984B
                       F2BFEC583C124B5546BF0CDAC543AB09
                       95589543B434951A29A40000EC56CBE7'
    ]) >>
  ] >>,
  / manifest / 3: << {
    / manifest-version / 1: 1,
    / manifest-sequence-number / 2: 1,
    / common / 3: << {
      / components / 2: [
        ['plaintext-firmware'],
      ]
    } >>,
    / install / 17: << [
      / fetch encrypted firmware /
      / directive-override-parameters / 20, {
        / parameter-content / 18:
          h'CE9AB65E7591EE38669C4CCA7A58FA324C1A0DBFDBC2C7
            C057376AFB805D660048310E8DAB045A2BE0A93F014FC9',
        / parameter-encryption-info / 19: << 96([
          / protected: / << {
            / alg / 1: 1 / AES-GCM-128 /
          } >>,
          / unprotected: / {
            / IV / 5: h'11D40BB56C3836AD44B39835B3ABC7FC'
          },
          / payload: / null / detached ciphertext /,
          / recipients: / [
            [
              / protected: / << {
              } >>,
```

```
                / unprotected: / {
                  / alg / 1: -3 / A128KW /,
                  / kid / 4: 'kid-1'
                },
                / payload: /
                  h'E01F4443C88CA89DF93A9C7E6D79D1C9BC330757C7D2D75A'
                  / 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:

```
D86BA2025873825824822F58205DEFDDB7F175FA20778FFE24BE7B9C369B
D8ED06AA4654F28794CD134CDBA932584AD28443A10126A0F658404C4A5F
B50738699649BA439237D20ADCADD6EC634A800A8E093733FC1C64984BF2
BFEC583C124B5546BF0CDAC543AB0995589543B434951A29A40000EC56CB
E703589DA4010102010357A102818152706C61696E746578742D6669726D
7761726511587C8414A212582ECE9AB65E7591EE38669C4CCA7A58FA324C
1A0DBFDBC2C7C057376AFB805D660048310E8DAB045A2BE0A93F014FC913
5843D8608443A10101A1055011D40BB56C3836AD44B39835B3ABC7FCF681
8341A0A2012204456B69642D315818E01F4443C88CA89DF93A9C7E6D79D1
C9BC330757C7D2D75A120F
```

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 decrypt and stored into another component with the suit-directive-copy directive. This approach works well on constrained devices with execute-in-place flash memory.

Commented [DT26]: Grammar: decrypted?

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'C6A66263CCF4C6FF5992AE4074B30DDD
                          34520AA099F6BAD96B2F60FE79F07EC4'
    ] >>,
    << / COSE_Sign1_Tagged / 18([
      / protected: / << {
        / algorithm-id / 1: -7 / ES256 /
      } >>,
      / unprotected: / {},
      / payload: / null,
      / signature: / h'DA08C3A6455FF30865A97A7F4FBC3BA1
                      5F954E39B57167DEA9FE16EBA12CFE33
                      D58790DB64CB70A08F89513B15CFF995
                      1222868195224E1AB87D46FA37F58864'
    ]) >>
  ] >>,
  / 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'11D40BB56C3836AD44B39835B3ABC7FC'
```

```
          },
          / payload: / null / detached ciphertext /,
          / recipients: / [
            [
              / protected: / << {
              } >>,
              / unprotected: / {
                / alg / 1: -3 / A128KW /,
                / kid / 4: 'kid-1'
              },
              / payload: /
                h'E01F4443C88CA89DF93A9C7E6D79D1C9BC330757C7D2D75A'
                / 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:

```
D86BA2025873825824822F5820C6A66263CCF4C6FF5992AE4074B30DDD34
520AA099F6BAD96B2F60FE79F07EC4584AD28443A10126A0F65840DA08C3
A6455FF30865A97A7F4FBC3BA15F954E39B57167DEA9FE16EBA12CFE33D5
8790DB64CB70A08F89513B15CFF9951222868195224E1AB87D46FA37F588
640358E5A40101020103582BA102828152706C61696E746578742D666972
6D776172658152656E637279707465642D6669726D776172651158AF900C
0114A20E182E157826687474707733A2F2F6578616D706C652E636F6D2F65
6E637279707465642D6669726D77617265150F0C0014A2135843D8608443
A10101A1055011D40BB56C3836AD44B39835B3ABC7FCF6818341A0A20122
04456B69642D315818E01F4443C88CA89DF93A9C7E6D79D1C9BC330757C7
D2D75A1601160F14A2035824822F582036921488FE6680712F734E11F58D
87EEB66D4B21A8A1AD3441060814DA16D50F0E181E030F
```

9. Security 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~~n~~ 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 needs to be provisioned. This specification places no requirements on the structure of the key identifier.

Commented [DT27]: grammar

Commented [DT28]: Grammar: "need" (identifiers is plural)

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 key data protected with that KEK.

Since the CEK is randomly generated, it must be ensured that the guidelines for random number generation in [RFC8937] are followed.

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. 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]:

Label	Name	Reference
TBD1	Encryption Info	Section 4

[Editor's Note: TBD1: Proposed 19]

11. References

11.1. Normative References

[I-D.ietf-cose-aes-ctr-and-cbc]
Housley, R. and H. Tschofenig, "CBOR Object Signing and Encryption (COSE): AES-CTR and AES-CBC", Work in Progress,

Internet-Draft, draft-ietf-cose-aes-ctr-and-cbc-06, 25 May 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-cose-aes-ctr-and-cbc-06>.

[I-D.ietf-suit-manifest]
Moran, B., Tschofenig, H., Birkholz, H., Zandberg, K., and O. Rønningstad, "A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest", Work in Progress, Internet-Draft, draft-ietf-suit-manifest-22, 27 February 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-suit-manifest-22>.

[I-D.ietf-suit-trust-domains]
Moran, B. and K. Takayama, "SUIT Manifest Extensions for Multiple Trust Domains", Work in Progress, Internet-Draft, draft-ietf-suit-trust-domains-04, 7 July 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-suit-trust-domains-04>.

[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>.

11.2. Informative References

[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>.

[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. 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, and Carsten Bormann for their review feedback. Finally, we would like to thank Dick Brooks for making us aware of the challenges firmware encryption imposes on binary analysis.

Appendix B. 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_Value = #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_Value)

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

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