Abstract

Concise Binary Object Representation (CBOR) is a data format designed for small code size and small message size. There is a need for the ability to have the basic security services defined for this data format. This document specifies processing for signatures, message authentication codes, and encryption using CBOR. This document also specifies a representation for cryptographic keys using CBOR.

Contributing to this document

The source for this draft is being maintained in GitHub. Suggested changes should be submitted as pull requests at <https://github.com/cose-wg/cose-spec>. Instructions are on that page as well. Editorial changes can be managed in GitHub, but any substantial issues need to be discussed on the COSE mailing list.

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 http://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 May 25, 2016.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.
1.  Introduction ............................................. 4
    1.1.  Design changes from JOSE ................................. 5
    1.2.  Requirements Terminology .................................. 5
    1.3.  CBOR Grammar ............................................. 6
    1.4.  CBOR Related Terminology ................................. 6
    1.5.  Document Terminology ...................................... 7
2.  Basic COSE Structure .......................................... 7
3.  Header Parameters ........................................... 8
    3.1.  Common COSE Headers Parameters ......................... 9
4.  Signing Structure ............................................ 13
    4.1.  Externally Supplied Data ................................ 14
    4.2.  Signing and Verification Process ...................... 15
    4.3.  Computing Counter Signatures .......................... 17
5.  Encryption Objects ........................................... 17
    5.1.  Enveloped COSE Structure ................................. 18
    5.1.1.  Recipient Algorithm Classes .......................... 19
    5.2.  Encrypted COSE structure .................................. 20
    5.3.  Encryption Algorithm for AEAD algorithms ............... 20
    5.4.  Encryption algorithm for AE algorithms ................ 21
6.  MAC Objects .................................................. 22
    6.1.  How to compute a MAC .................................... 23
7.  Key Structure ................................................ 24
    7.1.  COSE Key Common Parameters .............................. 25
8.  Signature Algorithms .......................................... 27
    8.1.  ECDSA .................................................... 28
    8.1.1.  Security Considerations ............................... 29
    9.1.  Hash-based Message Authentication Codes (HMAC) ........ 30
    9.1.1.  Security Considerations ............................... 31
    9.2.  AES Message Authentication Code (AES-CBC-MAC) .......... 31
    9.2.1.  Security Considerations ............................... 32
10. Content Encryption Algorithms ................................. 32
    10.1.  AES GCM ................................................ 33
    10.1.1.  Security Considerations .............................. 34
    10.2.  AES CCM ................................................ 34
    10.2.1.  Security Considerations .............................. 37
1. Introduction

There has been an increased focus on the small, constrained devices that make up the Internet of Things (IOT). One of the standards that has come out of this process is the Concise Binary Object Representation (CBOR). CBOR extended the data model of the JavaScript Object Notation (JSON) by allowing for binary data among other changes. CBOR is being adopted by several of the IETF working groups dealing with the IOT world as their encoding of data structures. CBOR was designed specifically to be both small in terms of messages transport and implementation size as well having a schema free decoder. A need exists to provide basic message security services for IOT and using CBOR as the message encoding format makes sense.

The JOSE working group produced a set of documents [RFC7515][RFC7516][RFC7517][RFC7518] using JSON [RFC7159] that specified how to process encryption, signatures and message authentication (MAC) operations, and how to encode keys using JSON. This document does the same work for use with the CBOR [RFC7049] document format. While there is a strong attempt to keep the flavor of the original JOSE documents, two considerations are taken into account:
CBOR has capabilities that are not present in JSON and should be used. One example of this is the fact that CBOR has a method of encoding binary directly without first converting it into a base64 encoded string.

COSE is not a direct copy of the JOSE specification. In the process of creating COSE, decisions that were made for JOSE were re-examined. In many cases different results were decided on as the criteria were not always the same as for JOSE.

1.1. Design changes from JOSE

- Define a top level message structure so that encrypted, signed and MACed messages can easily identified and still have a consistent view.
- Signed messages separate the concept of protected and unprotected parameters that are for the content and the signature.
- Recipient processing has been made more uniform. A recipient structure is required for all recipients rather than only for some.
- MAC messages are separated from signed messages.
- MAC messages have the ability to use all recipient algorithms on the MAC authentication key.
- Use binary encodings for binary data rather than base64url encodings.
- Combine the authentication tag for encryption algorithms with the ciphertext.
- Remove the flattened mode of encoding. Forcing the use of an array of recipients at all times forces the message size to be two bytes larger, but one gets a corresponding decrease in the implementation size that should compensate for this. [CREF1]

1.2. Requirements 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 [RFC2119].

When the words appear in lower case, their natural language meaning is used.
1.3. CBOR Grammar

There currently is no standard CBOR grammar available for use by specifications. We therefore describe the CBOR structures in prose. There is a version of a CBOR grammar in the CBOR Data Definition Language (CDDL) [I-D.greevenbosch-appsawg-cbor-cddl]. An informational version of the CBOR grammar that reflects what is in the prose can be found in Appendix A. Since CDDL has not be published as an RFC, this grammar may not work with the final version of CDDL when it is published.

The document was developed by first working on the grammar and then developing the prose to go with it. An artifact of this is that the prose was written using the primitive type strings defined by CDDL. In this specification, the following primitive types are used:

- **bool** - a boolean value (true: major type 7, value 21; false: major type 7, value 20).
- **bstr** - byte string (major type 2).
- **int** - an unsigned integer or a negative integer.
- **nil** - a null value (major type 7, value 22).
- **nint** - a negative integer (major type 1).
- **tstr** - a UTF-8 text string (major type 3).
- **uint** - an unsigned integer (major type 0).

Text from here to start of next section to be removed

NOTE: For the purposes of review, we are currently interlacing the CDDL grammar into the text of document. This is being done for simplicity of comparison of the grammar against the prose. The grammar will be removed to an appendix during WGLC.

```
start = COSE_Untagged_Message / COSE_Tagged_Message /
       COSE_Key / COSE_KeySet / Internal_Types
```

1.4. CBOR Related Terminology

In JSON, maps are called objects and only have one kind of map key: a string. In COSE, we use both strings and integers (both negative and unsigned integers) as map keys. The integers are used for compactness of encoding and easy comparison. (Generally, in this
document the value zero is going to be reserved and not used.) Since the work "key" is mainly used in its other meaning, as a cryptographic key, we use the term "label" for this usage as a map keys.

Text from here to start of next section to be removed

label = int / tstr
values = any

1.5. Document Terminology

In this document we use the following terminology: [CREF2]

Byte is a synonym for octet.

Key management is used as a term to describe how a key at level n is obtained from level n+1 in encrypted and MACed messages. The term is also used to discuss key life cycle management, this document does not discuss key life cycle operations.

2. Basic COSE Structure

The COSE Message structure is designed so that there can be a large amount of common code when parsing and processing the different security messages. All of the message structures are built on a CBOR array type. The first three elements of the array contains the same basic information.

1. The set of protected header parameters wrapped in a bstr.

2. The set of unprotected header parameters as a map.

3. The content of the message. The content is either the plain text or the encrypted text as appropriate. (The content may be absent, but the location is still used.)

Elements after this point are dependent on the specific message type.

Identification of which message is present is done by one of two methods:

- The specific message type is known from the context in which it is placed. This may be defined by a marker in the containing structure or by restrictions specified by the application protocol.
The message type is identified by a CBOR tag. This document defines a CBOR tag for each of the message structures.

COSE_Untagged_Message = COSE_Sign /
  COSE_Enveloped /
  COSE_Encrypted /
  COSE_Mac

COSE_Tagged_Message = COSE_Sign_Tagged /
  COSE_Enveloped_Tagged /
  COSE_Encrypted_Tagged /
  COSE_Mac_Tagged

3. Header Parameters

The structure of COSE has been designed to have two buckets of information that are not considered to be part of the payload itself, but are used for holding information about content, algorithms, keys, or evaluation hints for the processing of the layer. These two buckets are available for use in all of the structures in this document except for keys. While these buckets can be present, they may not all be usable in all instances. For example, while the protected bucket is defined as part of recipient structures, most of the algorithms that are used for recipients do not provide the necessary functionality to provide the needed protection and thus the bucket should not be used.

Both buckets are implemented as CBOR maps. The map key is a 'label' (Section 1.4). The value portion is dependent on the definition for the label. Both maps use the same set of label/value pairs. The integer and string values for labels has been divided into several sections with a standard range, a private range, and a range that is dependent on the algorithm selected. The defined labels can be found in the 'COSE Header Parameters' IANA registry (Section 16.2).

Two buckets are provided for each layer:

protected: Contains parameters about the current layer that are to be cryptographically protected. This bucket MUST be empty if it is not going to be included in a cryptographic computation. This bucket is encoded in the message as a binary object. This value is obtained by CBOR encoding the protected map and wrapping it in a bstr object. Senders SHOULD encode an empty protected map as a zero length binary object (it is shorter). Recipients MUST accept both a zero length binary value and a zero length map encoded in
the binary value. The wrapping allows for the encoding of the protected map to be transported with a greater chance that it will not be altered in transit. (Badly behaved intermediates could decode and re-encode, but this will result in a failure to verify unless the re-encoded byte string is identical to the decoded byte string.) This finesses the problem of all parties needing to be able to do a common canonical encoding.

unprotected: Contains parameters about the current layer that are not cryptographically protected.

Only parameters that deal with the current layer are to be placed at that layer. As an example of this, the parameter ‘content type’ describes the content of the message being carried in the message. As such, this parameter is placed only in the content layer and is not placed in the recipient or signature layers. In principle, one should be able to process any given layer without reference to any other layer. (The only data that should need to cross layers is the cryptographic key.)

The buckets are present in all of the security objects defined in this document. The fields in order are the ‘protected’ bucket (as a CBOR ‘bstr’ type) and then the ‘unprotected’ bucket (as a CBOR ‘map’ type). The presence of both buckets is required. The parameters that go into the buckets come from the IANA "COSE Header Parameters" (Section 16.2). Some common parameters are defined in the next section, but a number of parameters are defined throughout this document.

Text from here to start of next section to be removed [CREF3]

header_map = {+ label => any }

Headers = (  
  protected : bstr, ; Contains a header_map  
  unprotected : header_map
)

3.1. Common COSE Headers Parameters

This section defines a set of common header parameters. A summary of those parameters can be found in Table 1. This table should be consulted to determine the value of label used as well as the type of the value.

The set of header parameters defined in this section are:
alg  This parameter is used to indicate the algorithm used for the security processing. This parameter MUST be present at each level of a signed, encrypted or authenticated message. This parameter MUST be in the protected header bucket. The value is taken from the ‘COSE Algorithm Registry’ (see Section 16.4).

crit  This parameter is used to ensure that applications will take appropriate action based on the values found. The parameter is used to indicate which protected header labels an application that is processing a message is required to understand. When present, this parameter MUST be placed in the protected header bucket.

*  Integer labels in the range of 0 to 10 SHOULD be omitted.

*  Integer labels in the range -1 to -255 can be omitted as they are algorithm dependent. If an application can correctly process an algorithm, it can be assumed that it will correctly process all of the parameters associated with that algorithm. (The algorithm range is -1 to -65536, it is assumed that the higher end will deal with more optional algorithm specific items.)

The header parameter values indicated by ‘crit’ can be processed by either the security library code or by an application using a security library, the only requirement is that the parameter is processed. If the ‘crit’ value list includes a value for which the parameter is not in the protected bucket, this is a fatal error in processing the message.

content type  This parameter is used to indicate the content type of the data in the payload or ciphertext fields. Integers are from the ‘CoAP Content-Formats’ IANA registry table. [CREF4] Strings are from the IANA ‘Media Types’ registry. Applications SHOULD provide this parameter if the content structure is potentially ambiguous.

kid  This parameter one of the ways that can be used to find the key to be used. The value of this parameter is matched against the ‘kid’ member in a COSE_Key structure. Applications MUST NOT assume that ‘kid’ values are unique. There may be more than one key with the same ‘kid’ value, it may be required that all of the keys need to be checked to find the correct one. The internal structure of ‘kid’ values is not defined and generally cannot be relied on by applications. Key identifier values are hints about which key to use. They are not directly a security critical field. For this reason, they can be placed in the unprotected headers bucket.
Initialization Vector  This parameter holds the Initialization Vector (IV) value. For some symmetric encryption algorithms this may be referred to as a nonce. As the IV is authenticated by encryption process, it can be placed in the unprotected header bucket.

Partial Initialization Vector  This parameter holds a part of the IV value. When using the COSE_Encrypted structure, frequently a portion of the IV is part of the context associated with the key value. This field is used to carry the portion of the IV that changes for each message. As the IV is authenticated by the encryption process, it can be placed in the unprotected header bucket.

Some applications may also use this value for doing replay protection. When this is done, the value will normally be defined by the application to be increasing in value for every message.

counter signature  This parameter holds a counter signature value. Counter signatures provide a method of having a second party sign some data. The counter signature can occur as an unprotected attribute in any of the following structures: COSE_Sign, COSE_signature, COSE_Enveloped, COSE_recipient, COSE_Encrypted, COSE_mac. These structures all have the same basic structure so that a consistent calculation of the counter signature can be computed. Details on computing counter signatures are found in Section 4.3.

creation time  This parameter provides the time the content was created. For signatures and recipient structures, this would be the time that the signature or recipient key object was created. For content structures, this would be the time that the content was created. The unsigned integer value is the number of seconds, excluding leap seconds; after midnight UTC, January 1, 1970.
<table>
<thead>
<tr>
<th>name</th>
<th>label</th>
<th>value type</th>
<th>value registry</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alg</td>
<td>1</td>
<td>int / tstr</td>
<td>COSE Algorithm Registry</td>
<td>Integers are taken from table --POINT TO REGISTRY--</td>
</tr>
<tr>
<td>crit</td>
<td>2</td>
<td>[+ label]</td>
<td>COSE Header Label Registry</td>
<td>integer values are from --POINT TO REGISTRY--</td>
</tr>
<tr>
<td>content type</td>
<td>3</td>
<td>tstr / int</td>
<td>CoAP Content- Formats or Media Types registry</td>
<td>Value is either a Media Type or an integer from the CoAP Content Format registry</td>
</tr>
<tr>
<td>kid</td>
<td>4</td>
<td>bstr</td>
<td></td>
<td>key identifier</td>
</tr>
<tr>
<td>IV</td>
<td>5</td>
<td>bstr</td>
<td></td>
<td>Full Initialization Vector</td>
</tr>
<tr>
<td>Partial IV</td>
<td>6</td>
<td>uint</td>
<td></td>
<td>Partial Initialization Vector</td>
</tr>
<tr>
<td>counter signature</td>
<td>7</td>
<td>COSE_signature</td>
<td></td>
<td>CBOR encoded signature structure</td>
</tr>
<tr>
<td>creation time</td>
<td>*</td>
<td>uint</td>
<td></td>
<td>Time the content was created</td>
</tr>
</tbody>
</table>

Table 1: Common Header Parameters
4. Signing Structure

The signature structure allows for one or more signatures to be applied to a message payload. There are provisions for parameters about the content and parameters about the signature to be carried along with the signature itself. These parameters may be authenticated by the signature, or just present. Examples of parameters about the content would be the type of content, when the content was created, and who created the content. \[CREF5\] Examples of parameters about the signature would be the algorithm and key used to create the signature, when the signature was created, and counter-signatures.

When more than one signature is present, the successful validation of one signature associated with a given signer is usually treated as a successful signature by that signer. However, there are some application environments where other rules are needed. An application that employs a rule other than one valid signature for each signer must specify those rules. Also, where simple matching of the signer identifier is not sufficient to determine whether the signatures were generated by the same signer, the application specification must describe how to determine which signatures were generated by the same signer. Support of different communities of recipients is the primary reason that signers choose to include more than one signature. For example, the COSE_Sign structure might include signatures generated with the RSA signature algorithm and with the Elliptic Curve Digital Signature Algorithm (ECDSA) signature algorithm. This allows recipients to verify the signature associated with one algorithm or the other. (The original source of this text is [RFC5652].) More detailed information on multiple signature evaluation can be found in [RFC5752].

A COSE Signing Message is divided into two parts. The CBOR object that carries the body and information about the body is called the COSE_Sign structure. The CBOR object that carries the signature and information about the signature is called the COSE_Signature structure. Examples of COSE Signing Messages can be found in Appendix C.3.

The COSE_Sign structure is a CBOR array. The fields of the array in order are:

- protected is described in Section 3.
- unprotected is described in Section 3.
- payload contains the serialized content to be signed. If the payload is not present in the message, the application is required
to supply the payload separately. The payload is wrapped in a
bstr to ensure that it is transported without changes. If the
payload is transported separately, then a nil CBOR object is
placed in this location and it is the responsibility of the
application to ensure that it will be transported without changes.

signatures is an array of signatures. Each signature is represented
as a COSE_signature structure.

The COSE_signature structure is a CBOR array. The fields of the
array in order are:

protected  is described in Section 3.

unprotected  is described in Section 3.

signature  contains the computed signature value. The type of the
field is a bstr.

Text from here to start of next section to be removed

COSE_Sign_Tagged = #6.999(COSE_Sign) ; Replace 999 with TBD1

COSE_Sign = [ Headers,
                payload : bstr / nil,
                signatures : [+ COSE_signature]
            ]

COSE_signature = [ Headers,
                   signature : bstr
               ]

4.1. Externally Supplied Data

One of the features that we supply in the COSE document is the
ability for applications to provide additional data to be
authenticated as part of the security, but that is not carried as
part of the COSE object. The primary reason for supporting this can
be seen by looking at the CoAP message structure [RFC7252] where the
facility exists for options to be carried before the payload.

[CREF6] An example of data that can be placed in this location would
be transaction ids and nonces to check for replay protection. If the
data is in the options section, then it is available for routers to
help in performing the replay detection and prevention. However, it
may also be desired to protect these values so that they cannot be
modified in transit. This is the purpose of the externally supplied data field.

This document describes the process for using a byte array of externally supplied authenticated data, however the method of constructing the byte array is a function of the application. Applications that use this feature need to define how the externally supplied authenticated data is to be constructed. Such a construction needs to take into account the following issues:

- If multiple items are included, care needs to be taken that data cannot bleed between the items. This is usually addressed by making fields fixed width and/or encoding the length of the field. Using options from CoAP [RFC7252] as an example, these fields use a TLV structure so they can be concatenated without any problems.

- If multiple items are included, a defined order for the items needs to be defined. Using options from CoAP as an example, an application could state that the fields are to be ordered by the option number.

4.2. Signing and Verification Process

In order to create a signature, a consistent byte stream is needed in order to process. This algorithm takes in the body information (COSE_Sign), the signer information (COSE_Signer), and the application data (External). A CBOR array is used to construct the byte stream to be processed. The fields of the array in order are:

1. The protected attributes from the body structure encoded in a bstr type.
2. The protected attributes from the signer structure encoded in a bstr type.
3. The protected attributes from the application encoded in a bstr type. If this field is not supplied, it defaults to a zero length binary string.
4. The payload to be signed encoded in a bstr type. The payload is placed here independent of how it is transported.

How to compute a signature:

1. Create a CBOR array and populate it with the appropriate fields. For body_protected and sign_protected, if the map is empty, a bstr of length zero is used.
2. If the application has supplied external additional authenticated
data to be included in the computation, then it is placed in the
third field. If no data was supplied, then a zero length binary
string is used.

3. Create the value ToBeSigned by encoding the Sig_structure to a
byte string.

4. Call the signature creation algorithm passing in K (the key to
sign with), alg (the algorithm to sign with) and ToBeSigned (the
value to sign).

5. Place the resulting signature value in the 'signature' field of
the map.

How to verify a signature:

1. Create a Sig_structure object and populate it with the
appropriate fields. For body_protected and sign_protected, if
the map is empty, a bstr of length zero is used.

2. If the application has supplied external additional authenticated
data to be included in the computation, then it is placed in the
third field. If no data was supplied, then a zero length binary
string is used.

3. Create the value ToBeSigned by encoding the Sig_structure to a
byte string.

4. Call the signature verification algorithm passing in K (the key
to verify with), alg (the algorithm used sign with), ToBeSigned
(the value to sign), and sig (the signature to be verified).

In addition to performing the signature verification, one must also
perform the appropriate checks to ensure that the key is correctly
paired with the signing identity and that the appropriate
authorization is done.

Text from here to start of next section to be removed

The COSE structure used to create the byte stream to be signed uses
the following CDDL grammar structure:
Sig_structure = [
    body_protected: bstr,
    sign_protected: bstr,
    external_aad: bstr,
    payload: bstr
]

4.3. Computing Counter Signatures

Counter signatures provide a method of having a different signature occur on some piece of content. This is normally used to provide a signature on a signature allowing for a proof that a signature existed at a given time (i.e. a Timestamp). In this document we allow for counter signatures to exist in a greater number of environments. As an example, it is possible to place a counter signature in the unprotected attributes of a COSE_Enveloped object. This would allow for an intermediary to either verify that the encrypted byte stream has not been modified, without being able to decrypt it. Or for the intermediary to assert that an encrypted byte stream either existed at a given time or passed through it in terms of routing (i.e. a proxy signature).

An example of a proxy signature on a signature can be found in Appendix C.3.3. An example of a proxy signature on an encryption object can be found in Appendix C.2.3.

The creation and validation of counter signatures over the different items relies on the fact that the structure all of our objects have the same structure. The elements are a set of protected attributes, a set of unprotected attributes and a body in that order. This means that the Sig_structure can be used for in a uniform manner to get the byte stream for processing a signature. If the counter signature is going to be computed over a COSE_Enveloped structure, the body_protected and payload items can be mapped into the Sig_structure in the same manner as from the COSE_Sign structure.

5. Encryption Objects

COSE supports two different encryption structures. COSE_Enveloped is used when the key needs to be explicitly identified. This structure supports the use of recipient structures to allow for random content encryption keys to be used. COSE_Enveloped is used when a recipient structure is not needed because the key to be used is known implicitly.

Schaad
Expires May 25, 2016
[Page 17]
5.1. Enveloped COSE Structure

The enveloped structure allows for one or more recipients of a message. There are provisions for parameters about the content and parameters about the recipient information to be carried in the message. The parameters associated with the content can be authenticated by the content encryption algorithm. The parameters associated with the recipient can be authenticated by the recipient algorithm (when the algorithm supports it). Examples of parameters about the content are the type of the content, when the content was created, and the content encryption algorithm. Examples of parameters about the recipient are the recipient’s key identifier, the recipient encryption algorithm. [CREF7]

In COSE, the same techniques and structures are used for encrypting both the plain text and the keys used to protect the text. This is different from the approach used by both [RFC5652] and [RFC7516] where different structures are used for the content layer and for the recipient layer. Two structures are defined COSE_Enveloped to hold the encrypted content and COSE_recipient to hold the encrypted keys for recipients. Examples of encrypted messages can be found in Appendix C.2.

The COSE_Enveloped structure is a CBOR array. The fields of the array in order are:

- protected is described in Section 3.
- unprotected is described in Section 3.
- ciphertext contains the encrypted plain text encoded as a bstr. If the ciphertext is to be transported independently of the control information about the encryption process (i.e. detached content) then the field is encoded as a null object.
- recipients contains an array of recipient information structures. The type for the recipient information structure is a COSE_recipient.

The COSE_recipient structure is a CBOR array. The fields of the array in order are:

- protected is described in Section 3.
- unprotected is described in Section 3.
ciphertext contains the encrypted key encoded as a bstr. If there is not an encrypted key, then this field is encoded as a nil value.

recipients contains an array of recipient information structures. The type for the recipient information structure is a COSE_recipient. If there are no recipient information structures, this element is absent.

Text from here to start of next section to be removed

COSE_Enveloped_Tagged = #6.998(COSE_Enveloped) ; Replace 998 with TBD32

COSE_Enveloped = [ COSE_Enveloped_fields recipients: [+COSE_recipient] ]

COSE_Enveloped_fields = ( Headers, ciphertext: bstr / nil, )

COSE_recipient = [ COSE_Enveloped_fields ? recipients: [+COSE_recipient] ]

5.1.1. Recipient Algorithm Classes

A typical encrypted message consists of an encrypted content and an encrypted CEK for one or more recipients. The CEK is encrypted for each recipient, using a key specific to that recipient. The details of this encryption depends on which class the recipient algorithm falls into. Specific details on each of the classes can be found in Section 12. A short summary of the five recipient algorithm classes is:

direct: The CEK is the same as the identified previously distributed symmetric key or derived from a previously distributed secret. No CEK is transported in the message.

symmetric key-encryption keys: The CEK is encrypted using a previously distributed symmetric KEK.

key agreement: The recipient’s public key and a sender’s private key are used to generate a pairwise secret, a KDF is applied to derive
a key, and then the CEK is either the derived key or encrypted by
the derived key.

key transport: The CEK is encrypted with the recipient’s public key.

passwords: The CEK is encrypted in a KEK that is derived from a
password.

5.2. Encrypted COSE structure

The encrypted structure does not have the ability to specify
recipients of the message. The structure assumes that the recipient
of the object will already know the identity of the key to be used in
order to decrypt the message. If a key needs to be identified to the
recipient, the enveloped structure ought to be used.

The structure defined to hold an encrypted message is COSE_Encrypted.
Examples of encrypted messages can be found in Appendix C.2.

The CDDL grammar structure for the COSE_Encrypted type is:

COSE_Encrypted_Tagged = #6.997(COSE_Encrypted) ; Replace 997 with TBD3

COSE_Encrypted = [  
    COSE_Enveloped_fields
]

The COSE_Enveloped structure is a CBOR array. The fields of the
array in order are:

protected is described in Section 3.

unprotected is described in Section 3.

ciphertext contains the encrypted plain text. If the ciphertext is
to be transported independently of the control information about
the encryption process (i.e. detached content) then the field is
encoded as a null value.

5.3. Encryption Algorithm for AEAD algorithms

The encryption algorithm for AEAD algorithms is fairly simple. In
order to get a consistent encoding of the data to be authenticated,
the Enc_structure is used to have canonical form of the AAD. The
Enc_structure is a CBOR array.

1. Copy the protected header field from the message to be sent to
   the first location in the Enc_structure.
2. If the application has supplied external additional authenticated data to be included in the computation, then it is placed in the second location ('external_aad' field) of the Enc_structure. If no data was supplied, then a zero length binary value is used. (See Section 4.1 for application guidance on constructing this field.)

3. Encode the Enc_structure using a CBOR Canonical encoding Section 14 to get the AAD value.

4. Determine the encryption key. This step is dependent on the class of recipient algorithm being used. For:

   No Recipients: The key to be used is determined by the algorithm and key at the current level.

   Direct and Direct Key Agreement: The key is determined by the key and algorithm in the recipient structure. The encryption algorithm and size of the key to be used are inputs into the KDF used for the recipient. (For direct, the KDF can be thought of as the identity operation.)

   Other: The key is randomly generated.

5. Call the encryption algorithm with K (the encryption key to use), P (the plain text) and AAD (the additional authenticated data). Place the returned cipher text into the ‘ciphertext’ field of the structure.

6. For recipients of the message, recursively perform the encryption algorithm for that recipient using the encryption key as the plain text.

   Text from here to start of next section to be removed

   Enc_structure = [
       protected: bstr,
       external_aad: bstr
   ]

5.4. Encryption algorithm for AE algorithms

   1. Verify that the ‘protected’ field is absent.

   2. Verify that there was no external additional authenticated data supplied for this operation.
3. Determine the encryption key. This step is dependent on the class of recipient algorithm being used. For:

No Recipients: The key to be used is determined by the algorithm and key at the current level.

Direct and Direct Key Agreement: The key is determined by the key and algorithm in the recipient structure. The encryption algorithm and size of the key to be used are inputs into the KDF used for the recipient. (For direct, the KDF can be thought of as the identity operation.)

Other: The key is randomly generated.

4. Call the encryption algorithm with K (the encryption key to use) and the P (the plain text). Place the returned cipher text into the ‘ciphertext’ field of the structure.

5. For recipients of the message, recursively perform the encryption algorithm for that recipient using the encryption key as the plain text.

6. MAC Objects

In this section we describe the structure and methods to be used when doing MAC authentication in COSE. This document allows for the use of all of the same classes of recipient algorithms as are allowed for encryption.

When using MAC operations, there are two modes in which it can be used. The first is just a check that the content has not been changed since the MAC was computed. Any class of recipient algorithm can be used for this purpose. The second mode is to both check that the content has not been changed since the MAC was computed, and to use the recipient algorithm to verify who sent it. The classes of recipient algorithms that support this are those that use a pre-shared secret or do static-static key agreement (without the key wrap step). In both of these cases, the entity that created and sent the message MAC can be validated. (The knowledge of sender assumes that there are only two parties involved and you did not send the message yourself.)

The MAC message uses two structures, the COSE_Mac structure defined in this section for carrying the body and the COSE_recipient structure (Section 5.1) to hold the key used for the MAC computation. Examples of MAC messages can be found in Appendix C.1.
The COSE_Mac structure is a CBOR array. The fields of the array in order are:

- protected is described in Section 3.
- unprotected is described in Section 3.
- payload contains the serialized content to be MACed. If the payload is not present in the message, the application is required to supply the payload separately. The payload is wrapped in a bstr to ensure that it is transported without changes. If the payload is transported separately, then a null CBOR object is placed in this location and it is the responsibility of the application to ensure that it will be transported without changes.
- tag contains the MAC value.
- recipients contains the recipient information. See the description under COSE_Encryption for more info.

Text from here to start of next section to be removed

COSE_Mac_Tagged = #6.996(COSE_Mac) ; Replace 996 with TBD4

COSE_Mac = [Headers, payload: bstr / nil, tag: bstr, recipients: [+COSE_recipient]]

6.1. How to compute a MAC

In order to get a consistent encoding of the data to be authenticated, the MAC_structure is used to have a canonical form. The MAC_structure is a CBOR array.

The steps to compute a MAC are:

1. Create a MAC_structure and copy the protected and payload fields from the COSE_Mac structure.

2. If the application has supplied external authenticated data, encode it as a binary value and place in the MAC_structure. If there is no external authenticated data, then use a zero length ‘bstr’. (See Section 4.1 for application guidance on constructing this field.)
3. Encode the MAC_structure using a canonical CBOR encoder. The resulting bytes is the value to compute the MAC on.

4. Compute the MAC and place the result in the ‘tag’ field of the COSE_Mac structure.

5. Encrypt and encode the MAC key for each recipient of the message.

MAC_structure = [ 
  protected: bstr,  
  external_aad: bstr,  
  payload: bstr 
]

7. Key Structure

A COSE Key structure is built on a CBOR map object. The set of common parameters that can appear in a COSE Key can be found in the IANA registry ‘COSE Key Common Parameter Registry’ (Section 16.5). Additional parameters defined for specific key types can be found in the IANA registry ‘COSE Key Type Parameters’ (Section 16.6).

A COSE Key Set uses a CBOR array object as its underlying type. The values of the array elements are COSE Keys. A Key Set MUST have at least one element in the array.

The element "kty" is a required element in a COSE_Key map.

COSE_Key = { 
  key_kty => tstr / int,  
  ? key_ops => [+ (tstr / int) ],  
  ? key_alg => tstr / int,  
  ? key_kid => bstr,  
  * label => values 
}

COSE_KeySet = [+COSE_Key]
This document defines a set of common parameters for a COSE Key object. Table 2 provides a summary of the parameters defined in this section. There are also a set of parameters that are defined for a specific key type. Key type specific parameters can be found in Section 13.

<table>
<thead>
<tr>
<th>name</th>
<th>label</th>
<th>CBOR type</th>
<th>registry</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kty</td>
<td>1</td>
<td>tstr / int</td>
<td>COSE</td>
<td>Identification of the key type</td>
</tr>
<tr>
<td>key_ops</td>
<td>4</td>
<td>[* (tstr/int)]</td>
<td></td>
<td>Restrict set of permissible operations</td>
</tr>
<tr>
<td>alg</td>
<td>3</td>
<td>tstr / int</td>
<td>COSE</td>
<td>Key usage restriction to this algorithm</td>
</tr>
<tr>
<td>kid</td>
<td>2</td>
<td>bstr</td>
<td></td>
<td>Key Identification value - match to kid in message</td>
</tr>
<tr>
<td>use</td>
<td>*</td>
<td>tstr</td>
<td></td>
<td>deprecated - don’t use</td>
</tr>
</tbody>
</table>

Table 2: Key Map Labels

kty: This parameter is used to identify the family of keys for this structure, and thus the set of key type specific parameters to be found. The set of values defined in this document can be found in Table 18. This parameter MUST be present in a key object. Implementations MUST verify that the key type is appropriate for the algorithm being processed. The key type MUST be included as part of a trust decision process.

alg: This parameter is used to restrict the algorithms that are to be used with this key. If this parameter is present in the key structure, the application MUST verify that this algorithm matches the algorithm for which the key is being used. If the algorithms do not match, then this key object MUST NOT be used to perform the cryptographic operation. Note that the same key can be in a different key structure with a different or no algorithm.
specified, however this is considered to be a poor security practice.

kid:  This parameter is used to give an identifier for a key.  The identifier is not structured and can be anything from a user provided string to a value computed on the public portion of the key.  This field is intended for matching against a ‘kid’ parameter in a message in order to filter down the set of keys that need to be checked.

key_ops:  This parameter is defined to restrict the set of operations that a key is to be used for.  The value of the field is an array of values from Table 3.

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>1</td>
<td>The key is used to create signatures. Requires private key fields.</td>
</tr>
<tr>
<td>verify</td>
<td>2</td>
<td>The key is used for verification of signatures.</td>
</tr>
<tr>
<td>encrypt</td>
<td>3</td>
<td>The key is used for key transport encryption.</td>
</tr>
<tr>
<td>decrypt</td>
<td>4</td>
<td>The key is used for key transport decryption.</td>
</tr>
<tr>
<td>wrap key</td>
<td>5</td>
<td>The key is used for key wrapping.</td>
</tr>
<tr>
<td>unwrap key</td>
<td>6</td>
<td>The key is used for key unwrapping. Requires private key fields.</td>
</tr>
<tr>
<td>key agree</td>
<td>7</td>
<td>The key is used for key agreement.</td>
</tr>
</tbody>
</table>

Table 3: Key Operation Values

The following provides a CDDL fragment which duplicates the assignment labels from Table 2 and Table 3.
8. Signature Algorithms

There are two basic signature algorithm structures that can be used. The first is the common signature with appendix. In this structure, the message content is processed and a signature is produced, the signature is called the appendix. This is the message structure used by our common algorithms such as ECDSA and RSASSA-PSS. (In fact the SSA in RSASSA-PSS stands for Signature Scheme with Appendix.) The basic structure becomes:

\[
\text{signature} = \text{Sign}(\text{message content}, \text{key})
\]

\[
\text{valid} = \text{Verification}(\text{message content}, \text{key}, \text{signature})
\]

The second is a signature with message recovery. (An example of such an algorithm is [PVSig].) In this structure, the message content is processed, but part of it is included in the signature. Moving bytes of the message content into the signature allows for an effectively smaller signature, the signature size is still potentially large, but the message content is shrunk. This has implications for systems implementing these algorithms and for applications that use them. The first is that the message content is not fully available until after a signature has been validated. Until that point the part of the message contained inside of the signature is unrecoverable. The second is that the security analysis of the strength of the signature is very much based on the structure of the message content. Messages which are highly predictable require additional randomness to be supplied as part of the signature process. In the worst case, it becomes the same as doing a signature with appendix. Thirdly, in the event that multiple signatures are applied to a message, all of the signature algorithms are going to be required to consume the same number of bytes of message content.
signature, message sent = Sign(message content, key)

valid, message content = Verification(message sent, key, signature)

At this time, only signatures with appendixes are defined for use with COSE, however considerable interest has been expressed in using a signature with message recovery algorithm due to the effective size reduction that is possible. Implementations will need to keep this in mind for later possible integration.

8.1. ECDSA

ECDSA [DSS] defines a signature algorithm using ECC.

The ECDSA signature algorithm is parameterized with a hash function \( h \). In the event that the length of the hash function output is greater than the group of the key, the left-most bytes of the hash output are used.

The algorithms defined in this document can be found in Table 4.

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>hash</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES256</td>
<td>-7</td>
<td>SHA-256</td>
<td>ECDSA w/ SHA-256</td>
</tr>
<tr>
<td>ES384</td>
<td>-8</td>
<td>SHA-384</td>
<td>ECDSA w/ SHA-384</td>
</tr>
<tr>
<td>ES512</td>
<td>-9</td>
<td>SHA-512</td>
<td>ECDSA w/ SHA-512</td>
</tr>
</tbody>
</table>

Table 4: ECDSA Algorithm Values

This document defines ECDSA to work only with the curves P-256, P-384 and P-521. This document requires that the curves be encoded using the ‘EC2’ key type. Implementations need to check that the key type and curve are correct when creating and verifying a signature. Other documents can defined it to work with other curves and points in the future.

In order to promote interoperability, it is suggested that SHA-256 be used only with curve P-256, SHA-384 be used only with curve P-384 and SHA-512 be used with curve P-521. This is aligned with the recommendation in Section 4 of [RFC5480].

The signature algorithm results in a pair of integers \( (R, S) \). These integers will be of the same order as length of the key used for the...
signature process. The signature is encoded by converting the integers into byte strings of the same length as the key size. The length is rounded up to the nearest byte and is left padded with zero bits to get to the correct length. The two integers are then concatenated together to form a byte string that is the resulting signature.

Using the function defined in [RFC3447] the signature is:
Signature = I2OSP(R, n) | I2OSP(S, n)
where n = ceiling(key_length / 8)

8.1.1. Security Considerations

The security strength of the signature is no greater than the minimum of the security strength associated with the bit length of the key and the security strength of the hash function.

System which have poor random number generation can leak their keys by signing two different messages with the same value ‘k’ (the per-message random value). [RFC6979] provides a method to deal with this problem by making ‘k’ be deterministic based on the message content rather than randomly generated. Applications that specify ECDSA should evaluate the ability to get good random number generation and require this when it is not possible.

Note: Use of this technique a good idea even when good random number generation exists. Doing so both reduces the possibility of having the same value of ‘k’ in two signature operations, but allows for reproducible signature values which helps testing.

There are two substitution attacks that can theoretically be mounted against the ECDSA signature algorithm.

- Changing the curve used to validate the signature: If one changes the curve used to validate the signature, then potentially one could have a two messages with the same signature each computed under a different curve. The only requirement on the new curve is that its order be the same as the old one and it be acceptable to the client. An example would be to change from using the curve secp256r1 (aka P-256) to using secp256k1. (Both are 256 bit curves.) We current do not have any way to deal with this version of the attack except to restrict the overall set of curves that can be used.

- Change the hash function used to validate the signature: If one has either two different hash functions of the same length, or one can truncate a hash function down, then one could potentially find collisions between the hash functions rather than within a single
hash function. (For example, truncating SHA-512 to 256 bits might collide with a SHA-256 bit hash value.) This attack can be mitigated by including the signature algorithm identifier in the data to be signed.


Message Authentication Codes (MACs) provide data authentication and integrity protection. They provide either no or very limited data origination. (One cannot, for example, be used to prove the identity of the sender to a third party.)

MACs use the same basic structure as signature with appendix algorithms. The message content is processed and an authentication code is produced. The authentication code is frequently called a tag. The basic structure becomes:

\[
tag = \text{MAC\_Create}(\text{message content, key})\]

\[
\text{valid} = \text{MAC\_Verify}(\text{message content, key, tag})\]

MAC algorithms can be based on either a block cipher algorithm (i.e. AES-MAC) or a hash algorithm (i.e. HMAC). This document defines a MAC algorithm for each of these two constructions.

9.1. Hash-based Message Authentication Codes (HMAC)

The Hash-based Message Authentication Code algorithm (HMAC) \([\text{RFC2104}] [\text{RFC4231}]\) was designed to deal with length extension attacks. The algorithm was also designed to allow for new hash algorithms to be directly plugged in without changes to the hash function. The HMAC design process has been vindicated as, while the security of hash algorithms such as MD5 has decreased over time, the security of HMAC combined with MD5 has not yet been shown to be compromised \([\text{RFC6151}]\).

The HMAC algorithm is parameterized by an inner and outer padding, a hash function (h) and an authentication tag value length. For this specification, the inner and outer padding are fixed to the values set in \([\text{RFC2104}]\). The length of the authentication tag corresponds to the difficulty of producing a forgery. For use in constrained environments, we define a set of HMAC algorithms that are truncated. There are currently no known issues when truncating, however the security strength of the message tag is correspondingly reduced in strength. When truncating, the left-most tag length bits are kept and transmitted.
The algorithm defined in this document can be found in Table 5.

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>Hash</th>
<th>Length</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HMAC 256/64</td>
<td></td>
<td>SHA-256</td>
<td>64</td>
<td>HMAC w/ SHA-256 truncated to 64 bits</td>
</tr>
<tr>
<td>HMAC 256/256</td>
<td>4</td>
<td></td>
<td>256</td>
<td>HMAC w/ SHA-256</td>
</tr>
<tr>
<td>HMAC 384/384</td>
<td>5</td>
<td></td>
<td>384</td>
<td>HMAC w/ SHA-384</td>
</tr>
<tr>
<td>HMAC 512/512</td>
<td>6</td>
<td></td>
<td>512</td>
<td>HMAC w/ SHA-512</td>
</tr>
</tbody>
</table>

Table 5: HMAC Algorithm Values

Some recipient algorithms carry the key while others derive a key from secret data. For those algorithms that carry the key (i.e. RSA-OAEP and AES-KeyWrap), the size of the HMAC key SHOULD be the same size as the underlying hash function. For those algorithms that derive the key, the derived key MUST be the same size as the underlying hash function.

If the key is obtained from a key structure, the key type MUST be 'Symmetric'. Implementations creating and validating MAC values MUST validate that the key type, key length, and algorithm are correct and appropriate for the entities involved.

9.1.1. Security Considerations

HMAC has proved to be resistant to attack even when used with weakening hash algorithms. The current best method appears to be a brute force attack on the key. This means that key size is going to be directly related to the security of an HMAC operation.

9.2. AES Message Authentication Code (AES-CBC-MAC)

AES-CBC-MAC is defined in [MAC].

AES-CBC-MAC is parameterized by the key length, the authentication tag length and the IV used. For all of these algorithms, the IV is fixed to all zeros. We provide an array of algorithms for various key lengths and tag lengths. The algorithms defined in this document are found in Table 6.
<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>key length</th>
<th>tag length</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-MAC</td>
<td>*</td>
<td>128</td>
<td>64</td>
<td>AES-MAC 128 bit key, 64-bit tag</td>
</tr>
<tr>
<td>128/64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AES-MAC</td>
<td>*</td>
<td>256</td>
<td>64</td>
<td>AES-MAC 256 bit key, 64-bit tag</td>
</tr>
<tr>
<td>256/64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AES-MAC</td>
<td>*</td>
<td>128</td>
<td>128</td>
<td>AES-MAC 128 bit key, 128-bit tag</td>
</tr>
<tr>
<td>128/128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AES-MAC</td>
<td>*</td>
<td>256</td>
<td>128</td>
<td>AES-MAC 256 bit key, 128-bit tag</td>
</tr>
<tr>
<td>256/128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6: AES-MAC Algorithm Values

Keys may be obtained either from a key structure or from a recipient structure. If the key obtained from a key structure, the key type MUST be ‘Symmetric’. Implementations creating and validating MAC values MUST validate that the key type, key length and algorithm are correct and appropriate for the entities involved.

9.2.1. Security Considerations

A number of attacks exist against CBC-MAC that need to be considered.

- A single key must only be used for messages of a fixed and known length. If this is not the case, an attacker will be able to generate a message with a valid tag given two message, tag pairs. This can be addressed by using different keys for different length messages. (CMAC mode also addresses this issue.)

- If the same key is used for both encryption and authentication operations, using CBC modes an attacker can produce messages with a valid authentication code.

- If the IV can be modified, then messages can be forged. This is addressed by fixing the IV to all zeros.

10. Content Encryption Algorithms

Content Encryption Algorithms provide data confidentiality for potentially large blocks of data using a symmetric key. They provide integrity on the data that was encrypted, however they provide either no or very limited data origination. (One cannot, for example, be
used to prove the identity of the sender to a third party.) The ability to provide data origination is linked to how the symmetric key is obtained.

We restrict the set of legal content encryption algorithms to those that support authentication both of the content and additional data. The encryption process will generate some type of authentication value, but that value may be either explicit or implicit in terms of the algorithm definition. For simplicity sake, the authentication code will normally be defined as being appended to the cipher text stream. The basic structure becomes:

ciphertext = Encrypt(message content, key, additional data)
valid, message content = Decrypt(cipher text, key, additional data)

Most AEAD algorithms are logically defined as returning the message content only if the decryption is valid. Many but not all implementations will follow this convention. The message content MUST NOT be used if the decryption does not validate.

10.1. AES GCM

The GCM mode is a generic authenticated encryption block cipher mode defined in [AES-GCM]. The GCM mode is combined with the AES block encryption algorithm to define an AEAD cipher.

The GCM mode is parameterized with by the size of the authentication tag and the size of the nonce. This document fixes the size of the nonce at 96-bits. The size of the authentication tag is limited to a small set of values. For this document however, the size of the authentication tag is fixed at 128 bits.

The set of algorithms defined in this document are in Table 7.

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A128GCM</td>
<td>1</td>
<td>AES-GCM mode w/ 128-bit key, 128-bit tag</td>
</tr>
<tr>
<td>A192GCM</td>
<td>2</td>
<td>AES-GCM mode w/ 192-bit key, 128-bit tag</td>
</tr>
<tr>
<td>A256GCM</td>
<td>3</td>
<td>AES-GCM mode w/ 256-bit key, 128-bit tag</td>
</tr>
</tbody>
</table>

Table 7: Algorithm Value for AES-GCM
Keys may be obtained either from a key structure or from a recipient structure. If the key obtained from a key structure, the key type MUST be ‘Symmetric’. Implementations encrypting and decrypting MUST validate that the key type, key length and algorithm are correct and appropriate for the entities involved.

10.1.1. Security Considerations

When using AES-CCM, the following restrictions MUST be enforced:

- The key and nonce pair MUST be unique for every message encrypted.
- The total amount of data encrypted MUST NOT exceed \(2^{39} - 256\) bits. An explicit check is required only in environments where it is expected that it might be exceeded.

Consideration was given to supporting smaller tag values, the constrained community would desire tag sizes in the 64-bit range. Doing show drastically changes both the maximum messages size (generally not an issue) and the number of times that a key can be used. Given that CCM is the usual mode for constrained environments restricted modes are not supported.

10.2. AES CCM

Counter with CBC-MAC (CCM) is a generic authentication encryption block cipher mode defined in [RFC3610]. The CCM mode is combined with the AES block encryption algorithm to define a commonly used content encryption algorithm used in constrained devices.

The CCM mode has two parameter choices. The first choice is \(M\), the size of the authentication field. The choice of the value for \(M\) involves a trade-off between message expansion and the probably that an attacker can undetectably modify a message. The second choice is \(L\), the size of the length field. This value requires a trade-off between the maximum message size and the size of the Nonce.

It is unfortunate that the specification for CCM specified \(L\) and \(M\) as a count of bytes rather than a count of bits. This leads to possible misunderstandings where AES-CCM-8 is frequently used to refer to a version of CCM mode where the size of the authentication is 64 bits and not 8 bits. These values have traditionally been specified as bit counts rather than byte counts. This document will follow the tradition of using bit counts so that it is easier to compare the different algorithms presented in this document.

We define a matrix of algorithms in this document over the values of \(L\) and \(M\). Constrained devices are usually operating in situations...
where they use short messages and want to avoid doing recipient specific cryptographic operations. This favors smaller values of M and larger values of L. Less constrained devices do will want to be able to use larger messages and are more willing to generate new keys for every operation. This favors larger values of M and smaller values of L. (The use of a large nonce means that random generation of both the key and the nonce will decrease the chances of repeating the pair on two different messages.)

The following values are used for L:

16 bits (2) limits messages to $2^{16}$ bytes (64 KiB) in length. This sufficiently long for messages in the constrained world. The nonce length is 13 bytes allowing for $2^{(13\times8)}$ possible values of the nonce without repeating.

64 bits (8) limits messages to $2^{64}$ bytes in length. The nonce length is 7 bytes allowing for $2^{56}$ possible values of the nonce without repeating.

The following values are used for M:

64 bits (8) produces a 64-bit authentication tag. This implies that there is a 1 in $2^{64}$ chance that a modified message will authenticate.

128 bits (16) produces a 128-bit authentication tag. This implies that there is a 1 in $2^{128}$ chance that a modified message will authenticate.
<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>L</th>
<th>M</th>
<th>k</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-CCM-16-64-128</td>
<td>10</td>
<td>16</td>
<td>64</td>
<td>128</td>
<td>AES-CCM mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>128-bit key, 64-bit tag, 13-byte nonce</td>
</tr>
<tr>
<td>AES-CCM-16-64-256</td>
<td>11</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>AES-CCM mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>256-bit key, 64-bit tag, 13-byte nonce</td>
</tr>
<tr>
<td>AES-CCM-64-64-128</td>
<td>30</td>
<td>64</td>
<td>64</td>
<td>128</td>
<td>AES-CCM mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>128-bit key, 64-bit tag, 7-byte nonce</td>
</tr>
<tr>
<td>AES-CCM-64-64-256</td>
<td>31</td>
<td>64</td>
<td>64</td>
<td>256</td>
<td>AES-CCM mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>256-bit key, 64-bit tag, 7-byte nonce</td>
</tr>
<tr>
<td>AES-CCM-16-128-128</td>
<td>12</td>
<td>16</td>
<td>128</td>
<td>128</td>
<td>AES-CCM mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>128-bit key, 128-bit tag, 13-byte nonce</td>
</tr>
<tr>
<td>AES-CCM-16-128-256</td>
<td>13</td>
<td>16</td>
<td>128</td>
<td>256</td>
<td>AES-CCM mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>256-bit key, 128-bit tag, 13-byte nonce</td>
</tr>
<tr>
<td>AES-CCM-64-128-128</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>128</td>
<td>AES-CCM mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>128-bit key, 128-bit tag, 7-byte nonce</td>
</tr>
<tr>
<td>AES-CCM-64-128-256</td>
<td>33</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>AES-CCM mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>256-bit key, 128-bit tag, 7-byte nonce</td>
</tr>
</tbody>
</table>

Table 8: Algorithm Values for AES-CCM

Keys may be obtained either from a key structure or from a recipient structure. If the key obtained from a key structure, the key type MUST be ‘Symmetric’. Implementations encrypting and decrypting MUST validate that the key type, key length and algorithm are correct and appropriate for the entities involved.
10.2.1. Security Considerations

When using AES-CCM, the following restrictions MUST be enforced:

- The key and nonce pair MUST be unique for every message encrypted.
- The total number of times the AES block cipher is used MUST NOT exceed $2^{61}$ operations. This limitation is the sum of times the block cipher is used in computing the MAC value and in performing stream encryption operations. An explicit check is required only in environments where it is expected that it might be exceeded.

[RFC3610] additionally calls out one other consideration of note. It is possible to do a pre-computation attack against the algorithm in cases where the portions encryption content is highly predictable. This reduces the security of the key size by half. Ways to deal with this attack include adding a random portion to the nonce value and/or increasing the key size used. Using a portion of the nonce for a random value will decrease the number of messages that a single key can be used for. Increasing the key size may require more resources in the constrained device. See sections 5 and 10 of [RFC3610] for more information.

10.3. ChaCha20 and Poly1305

ChaCha20 and Poly1305 combined together is a new AEAD mode that is defined in [RFC7539]. This is a new algorithm defined to be a cipher that is not AES and thus would not suffer from any future weaknesses found in AES. These cryptographic functions are designed to be fast in software-only implementations.

The ChaCha20/Poly1305 AEAD construction defined in [RFC7539] has no parameterization. It takes a 256-bit key and a 96-bit nonce as well as the plain text and additional data as inputs and produces the cipher text as an option. We define one algorithm identifier for this algorithm in Table 9.

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChaCha20/Poly1305</td>
<td>24</td>
<td>ChaCha20/Poly1305 w/ 256-bit key, 128-bit tag</td>
</tr>
</tbody>
</table>

Table 9: Algorithm Value for AES-GCM

Keys may be obtained either from a key structure or from a recipient structure. If the key obtained from a key structure, the key type...
MUST be ‘Symmetric’. Implementations encrypting and decrypting MUST validate that the key type, key length and algorithm are correct and appropriate for the entities involved.

10.3.1. Security Considerations

The pair of key, nonce MUST be unique for every invocation of the algorithm. Nonce counters are considered to be an acceptable way of ensuring that they are unique.

11. Key Derivation Functions (KDF)

Key Derivation Functions (KDFs) are used to take some secret value and generate a different one. The original secret values come in three basic flavors:

- Secrets that are uniformly random: This is the type of secret which is created by a good random number generator.
- Secrets that are not uniformly random: This is type of secret which is created by operations like key agreement.
- Secrets that are not random: This is the type of secret that people generate for things like passwords.

General KDF functions work well with the first type of secret, can do reasonable well with the second type of secret and generally do poorly with the last type of secret. None of the KDF functions in this section are designed to deal with the type of secrets that are used for passwords. Functions like PBSE2 [RFC2898] need to be used for that type of secret.

Many functions are going to handle the first two type of secrets differently. The KDF function defined in Section 11.1 can use different underlying constructions if the secret is uniformly random than if the secret is not uniformly random. This is reflected in the set of algorithms defined for HKDF.

When using KDF functions, one component that is generally included is context information. Context information is used to allow for different keying information to be derived from the same secret. The use of context based keying material is considered to be a good security practice. This document defines a single context structure and a single KDF function.
11.1.  HMAC-based Extract-and-Expand Key Derivation Function (HKDF)

The HKDF key derivation algorithm is defined in [RFC5869].

The HKDF algorithm takes these inputs:

secret - a shared value that is secret. Secrets may be either previously shared or derived from operations like a DH key agreement.

salt - an optional public value that is used to change the generation process. If specified, the salt is carried using the ‘salt’ algorithm parameter. While [RFC5869] suggests that the length of the salt be the same as the length of the underlying hash value, any amount of salt will improve the security as different key values will be generated. A parameter to carry the salt is defined in Table 11. This parameter is protected by being included in the key computation and does not need to be separately authenticated.

length - the number of bytes of output that need to be generated.

context information - Information that describes the context in which the resulting value will be used. Making this information specific to the context that the material is going to be used ensures that the resulting material will always be unique. The context structure used is encoded into the algorithm identifier.

PRF - The underlying pseudo-random function to be used in the HKDF algorithm. The PRF is encoded into the HKDF algorithm selection.

HKDF is defined to use HMAC as the underlying PRF. However, it is possible to use other functions in the same construct to provide a different KDF function that may be more appropriate in the constrained world. Specifically, one can use AES-CBC-MAC as the PRF for the expand step, but not for the extract step. When using a good random shared secret of the correct length, the extract step can be skipped. The extract cannot be skipped if the secret is not uniformly random, for example if it is the result of an ECDH key agreement step.

The algorithms defined in this document are found in Table 10.
<table>
<thead>
<tr>
<th>name</th>
<th>PRF</th>
<th>Skip extract</th>
<th>context</th>
</tr>
</thead>
<tbody>
<tr>
<td>HKDF</td>
<td>HMAC with</td>
<td>no</td>
<td>HKDF using HMAC SHA-256 as the PRF</td>
</tr>
<tr>
<td>SHA-256</td>
<td>SHA-256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HKDF</td>
<td>HMAC with</td>
<td>no</td>
<td>HKDF using HMAC SHA-512 as the PRF</td>
</tr>
<tr>
<td>SHA-512</td>
<td>SHA-512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HKDF AES-</td>
<td>AES-CBC-128</td>
<td>yes</td>
<td>HKDF using AES-MAC as the PRF with 128-bit key</td>
</tr>
<tr>
<td>MAC-128</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HKDF AES-</td>
<td>AES-CBC-256</td>
<td>yes</td>
<td>HKDF using AES-MAC as the PRF with 256-bit key</td>
</tr>
<tr>
<td>MAC-256</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 10: HKDF algorithms

<table>
<thead>
<tr>
<th>name</th>
<th>label</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>salt</td>
<td>-20</td>
<td>bstr</td>
<td>Random salt</td>
</tr>
</tbody>
</table>

Table 11: HKDF Algorithm Parameters

11.2. Context Information Structure

The context information structure is used to ensure that the derived keying material is "bound" to the context of the transaction. The context information structure used here is based on that defined in [SP800-56A]. By using CBOR for the encoding of the context information structure, we automatically get the same type and length separation of fields that is obtained by the use of ASN.1. This means that there is no need to encode the lengths for the base elements as it is done by the encoding used in JOSE (Section 4.6.2 of [RFC7518]). [CREF9]

The context information structure refers to PartyU and PartyV as the two parties which are doing the key derivation. Unless the application protocol defines differently, we assign PartyU to the entity that is creating the message and PartyV to the entity that is receiving the message. By doing this association, different keys will be derived for each direction as the context information is different in each direction.
Application protocols are free to define the roles differently. For example, they could assign the PartyU role to the entity that initiates the connection and allow directly sending multiple messages over the connection in both directions without changing the role information. It is still recommended that different keys be derived in each direction to avoid reflection problems.

The context structure is built from information that is known to both entities. This information can be obtained from a variety of sources:

- Fields can be defined by the application. This is commonly used to assign names to parties.
- Fields can be defined by usage of the output. Examples of this are the algorithm and key size that are being generated.
- Fields can be defined by parameters from the message. We define a set of parameters in Table 12 which can be used to carry the values associated with the context structure. Examples of this are identities and nonce values. These parameters are designed to be placed in the unprotected bucket of the recipient structure. (They do not need to be in the protected bucket since they already are included in the cryptographic computation by virtue of being included in the context structure.)

We define a CBOR object to hold the context information. This object is referred to as CBOR_KDF_Context. The object is based on a CBOR array type. The fields in the array are:

- AlgorithmID: This field indicates the algorithm for which the key material will be used. This field is required to be present and is a copy of the algorithm identifier in the message. The field exists in the context information so that if the same environment is used for different algorithms, then completely different keys will be generated each of those algorithms. (This practice means if algorithm A is broken and thus can is easier to find, the key derived for algorithm B will not be the same as the key for algorithm B.)

- PartyUIInfo: This field holds information about party U. The PartyUIInfo is encoded as a CBOR structure. The elements of PartyUIInfo are encoded in the order presented, however if the element does not exist no element is placed in the array. The elements of the PartyUIInfo array are:

  - identity: This contains the identity information for party U. The identities can be assigned in one of two manners. Firstly, a
protocol can assign identities based on roles. For example, the roles of "client" and "server" may be assigned to different entities in the protocol. Each entity would then use the correct label for the data they send or receive. The second way is for a protocol to assign identities is to use a name based on a naming system (i.e. DNS, X.509 names).

We define an algorithm parameter 'PartyU identity' that can be used to carry identity information in the message. However, identity information is often known as part of the protocol and can thus be inferred rather than made explicit. If identity information is carried in the message, applications SHOULD have a way of validating the supplied identity information. The identity information does not need to be specified and can be left as absent.

nonce  This contains a one time nonce value. The nonce can either be implicit from the protocol or carried as a value in the unprotected headers.

We define an algorithm parameter 'PartyU nonce' that can be used to carry this value in the message. However, the nonce value could be determined by the application and the value determined from elsewhere.

This item is optional and can be absent.

other  This contains other information that is defined by the protocol.

This item is optional and can be absent.

PartyVInfo  This field holds information about party V. The PartyVInfo is encoded as a CBOR structure. For store and forward environments, the party V information may be minimal or even absent. The elements of PartyVInfo are encoded in the order presented, however if the element does not exist no element is placed in the array. The elements of the PartyVInfo array are:

identity  This contains the identity information for party V. The identities can be assigned in one of two manners. Firstly, a protocol can assign identities based on roles. For example, the roles of "client" and "server" may be assigned to different entities in the protocol. Each entity would then use the correct label for the data they send or receive. The second way is for a protocol to assign identities is to use a name based on a naming system (i.e. DNS, X.509 names).

We define an algorithm parameter 'PartyU identity' that can be used to carry identity information in the message. However, identity information is often known as part of the protocol and can thus be inferred rather than made explicit. If identity information is carried in the message, applications SHOULD have
a way of validating the supplied identity information. The identity information does not need to be specified and can be left as absent.

nonce  This contains a one time nonce value. The nonce can either be implicit from the protocol or carried as a value in the unprotected headers. We define an algorithm parameter 'PartyU nonce' that can be used to carry this value in the message. However, the nonce value could be determined by the application and the value determined from elsewhere. This item is optional and can be absent.

other  This contains other information that is defined by the protocol. This item is optional and can be absent.

SuppPubInfo  This field contains public information that is mutually known to both parties.

keyDataLength  This is set to the number of bits of the desired output value. (This practice means if algorithm A can use two different key lengths, the key derived for longer key size will not contain the key for shorter key size as a prefix.)

protected  This field contains the protected parameter field.

other  The field other is for free form data defined by the application. An example is that an application could define two different strings to be placed here to generate different keys for a data stream vs a control stream. This field is optional and will only be present if the application defines a structure for this information. Applications that define this SHOULD use CBOR to encode the data so that types and lengths are correctly included.

SuppPrivInfo  This field contains private information that is mutually known information. An example of this information would be a pre-existing shared secret. The field is optional and will only be present if the application defines a structure for this information. Applications that define this SHOULD use CBOR to encode the data so that types and lengths are correctly included.
<table>
<thead>
<tr>
<th>name</th>
<th>label</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PartyU identity</td>
<td>-21</td>
<td>bstr</td>
<td>Party U identity Information</td>
</tr>
<tr>
<td>PartyU nonce</td>
<td>-22</td>
<td>bstr / int</td>
<td>Party U provided nonce</td>
</tr>
<tr>
<td>PartyU other</td>
<td>-23</td>
<td>bstr</td>
<td>Party U other provided information</td>
</tr>
<tr>
<td>PartyV identity</td>
<td>-24</td>
<td>bstr</td>
<td>Party V identity Information</td>
</tr>
<tr>
<td>PartyV nonce</td>
<td>-25</td>
<td>bstr / int</td>
<td>Party V provided nonce</td>
</tr>
<tr>
<td>PartyV other</td>
<td>-26</td>
<td>bstr</td>
<td>Party V other provided information</td>
</tr>
</tbody>
</table>

Table 12: Context Algorithm Parameters

Text from here to start of next section to be removed

COSE_KDF_Context = [
    AlgorithmID : int / tstr,
    PartyUIInfo : [
        ? nonce : bstr / int,
        ? identity : bstr,
        ? other : bstr,
    ],
    PartyVInfo : [
        ? nonce : bstr,
        ? identity : bstr / tstr,
        ? other : bstr,
    ],
    SuppPubInfo : [
        keyDataLength : uint,
        protected : bstr,
        ? other : bstr,
    ],
    ? SuppPrivInfo : bstr
]
12. Recipient Algorithm Classes

Recipient algorithms can be defined into a number of different classes. COSE has the ability to support many classes of recipient algorithms. In this section, a number of classes are listed and then a set of algorithms are specified for each of the classes. The names of the recipient algorithm classes used here are the same as are defined in [RFC7516]. Other specifications use different terms for the recipient algorithm classes or do not support some of our recipient algorithm classes.

12.1. Direct Encryption

The direct encryption class algorithms share a secret between the sender and the recipient that is used either directly or after manipulation as the content key. When direct encryption mode is used, it MUST be the only mode used on the message.

The COSE_Enveloped structure for the recipient is organized as follows:

- The ‘protected’ field MUST be a zero length item unless it is used in the computation of the content key.
- The ‘alg’ parameter MUST be present.
- A parameter identifying the shared secret SHOULD be present.
- The ‘ciphertext’ field MUST be a zero length item.
- The ‘recipients’ field MUST be absent.

12.1.1. Direct Key

This recipient algorithm is the simplest, the identified key is directly used as the key for the next layer down in the message. There are no algorithm parameters defined for this algorithm. The algorithm identifier value is assigned in Table 13.

When this algorithm is used, the protected field MUST be zero length. The key type MUST be ‘Symmetric’.
### Table 13: Direct Key

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>direct</td>
<td>-6</td>
<td>Direct use of CEK</td>
</tr>
</tbody>
</table>

#### 12.1.1.1. Security Considerations

This recipient algorithm has several potential problems that need to be considered:

- These keys need to have some method to be regularly updated over time. All of the content encryption algorithms specified in this document have limits on how many times a key can be used without significant loss of security.

- These keys need to be dedicated to a single algorithm. There have been a number of attacks developed over time when a single key is used for multiple different algorithms. One example of this is the use of a single key both for CBC encryption mode and CBC-MAC authentication mode.

- Breaking one message means all messages are broken. If an adversary succeeds in determining the key for a single message, then the key for all messages is also determined.

#### 12.1.2. Direct Key with KDF

These recipient algorithms take a common shared secret between the two parties and applies the HKDF function (Section 11.1) using the context structure defined in Section 11.2 to transform the shared secret into the necessary key. The ‘protected’ field can be of non-zero length. The ‘protected’ field is copied into the SuppPubInfo.protected field of the context structure. Either the ‘salt’ parameter of HKDF or the partyU ’nonce’ parameter of the context structure MUST be present. The salt/nonce parameter can be generated either randomly or deterministically. The requirement is that it be a unique value for the key pair in question.

If the salt/nonce value is generated randomly, then it is suggested that the length of the random value be the same length as the hash function underlying HKDF. While there is no way to guarantee that it will be unique, there is a high probability that it will be unique. If the salt/nonce value is generated deterministically, it can be guaranteed to be unique and thus there is no length requirement.
A new IV must be used if the same key is used in more than one message. The IV can be modified in a predictable manner, a random manner or an unpredictable manner. One unpredictable manner that can be used is to use the HKDF function to generate the IV. If HKDF is used for generating the IV, the algorithm identifier is set to "IV-GENERATION".

When these algorithms are used, the key type MUST be 'symmetric'.

The set of algorithms defined in this document can be found in Table 14.

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>KDF</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>direct+HKDF-SHA-256</td>
<td>*</td>
<td>HKDF</td>
<td>Shared secret w/ HKDF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHA-256</td>
<td>and SHA-256</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>direct+HKDF-SHA-512</td>
<td>*</td>
<td>HKDF</td>
<td>Shared secret w/ HKDF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHA-512</td>
<td>and SHA-512</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>direct+HKDF-AES-128</td>
<td>*</td>
<td>HKDF AES-</td>
<td>Shared secret w/ AES-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MAC-128</td>
<td>MAC 128-bit key</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>direct+HKDF-AES-256</td>
<td>*</td>
<td>HKDF AES-</td>
<td>Shared secret w/ AES-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MAC-256</td>
<td>MAC 256-bit key</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 14: Direct Key

12.1.2.1. Security Considerations

The shared secret needs to have some method to be regularly updated over time. The shared secret forms the basis of trust. Although not used directly, it should still be subject to scheduled rotation.

12.2. Key Wrapping

In key wrapping mode, the CEK is randomly generated and that key is then encrypted by a shared secret between the sender and the recipient. All of the currently defined key wrapping algorithms for COSE are AE algorithms. Key wrapping mode is considered to be superior to direct encryption if the system has any capability for doing random key generation. This is because the shared key is used to wrap random data rather than data has some degree of organization and may in fact be repeating the same content. The use of Key Wrapping loses the weak data origination that is provided by the direct encryption algorithms.
The COSE_Enveloped structure for the recipient is organized as follows:

- The ‘protected’ field MUST be absent if the key wrap algorithm is an AE algorithm.
- The ‘recipients’ field is normally absent, but can be used. Applications MUST deal with a recipients field present, not being able to decrypt that recipient is an acceptable way of dealing with it. Failing to process the message is not an acceptable way of dealing with it.
- The plain text to be encrypted is the key from next layer down (usually the content layer).
- At a minimum, the ‘unprotected’ field MUST contain the ‘alg’ parameter and SHOULD contain a parameter identifying the shared secret.

12.2.1. AES Key Wrapping

The AES Key Wrapping algorithm is defined in [RFC3394]. This algorithm uses an AES key to wrap a value that is a multiple of 64 bits. As such, it can be used to wrap a key for any of the content encryption algorithms defined in this document. The algorithm requires a single fixed parameter, the initial value. This is fixed to the value specified in Section 2.2.3.1 of [RFC3394]. There are no public parameters that vary on a per invocation basis.

Keys may be obtained either from a key structure or from a recipient structure. If the key obtained from a key structure, the key type MUST be ‘Symmetric’. Implementations encrypting and decrypting MUST validate that the key type, key length and algorithm are correct and appropriate for the entities involved.

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>key size</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A128KW</td>
<td>-3</td>
<td>128</td>
<td>AES Key Wrap w/ 128-bit key</td>
</tr>
<tr>
<td>A192KW</td>
<td>-4</td>
<td>192</td>
<td>AES Key Wrap w/ 192-bit key</td>
</tr>
<tr>
<td>A256KW</td>
<td>-5</td>
<td>256</td>
<td>AES Key Wrap w/ 256-bit key</td>
</tr>
</tbody>
</table>

Table 15: AES Key Wrap Algorithm Values
12.2.1.1. Security Considerations for AES-KW

The shared secret need to have some method to be regularly updated over time. The shared secret is the basis of trust.

12.3. Key Encryption

Key Encryption mode is also called key transport mode in some standards. Key Encryption mode differs from Key Wrap mode in that it uses an asymmetric encryption algorithm rather than a symmetric encryption algorithm to protect the key. This document defines one Key Encryption mode algorithm.

When using a key encryption algorithm, the COSE_Enveloped structure for the recipient is organized as follows:

- The 'protected' field MUST be absent.
- The plain text to be encrypted is the key from next layer down (usually the content layer).
- At a minimum, the 'unprotected' field MUST contain the 'alg' parameter and SHOULD contain a parameter identifying the asymmetric key.

12.4. Direct Key Agreement

The 'direct key agreement' class of recipient algorithms uses a key agreement method to create a shared secret. A KDF is then applied to the shared secret to derive a key to be used in protecting the data. This key is normally used as a CEK or MAC key, but could be used for other purposes if more than two layers are in use (see Appendix B).

The most commonly used key agreement algorithm used is Diffie-Hellman, but other variants exist. Since COSE is designed for a store and forward environment rather than an on-line environment, many of the DH variants cannot be used as the receiver of the message cannot provide any key material. One side-effect of this is that perfect forward secrecy (see [RFC4949]) is not achievable. A static key will always be used for the receiver of the COSE message.

Two variants of DH that are easily supported are:

- Ephemeral-Static DH: where the sender of the message creates a one time DH key and uses a static key for the recipient. The use of the ephemeral sender key means that no additional random input is needed as this is randomly generated for each message.
Static-Static DH: where a static key is used for both the sender and the recipient. The use of static keys allows for recipient to get a weak version of data origination for the message. When static-static key agreement is used, then some piece of unique data is required to ensure that a different key is created for each message.

In this specification, both variants are specified. This has been done to provide the weak data origination option for use with MAC operations.

When direct key agreement mode is used, there MUST be only one recipient in the message. This method creates the key directly and that makes it difficult to mix with additional recipients. If multiple recipients are needed, then the version with key wrap needs to be used.

The COSE_Enveloped structure for the recipient is organized as follows:

- The ‘protected’ field MUST be absent.
- At a minimum, the ‘unprotected’ field MUST contain the ‘alg’ parameter and SHOULD contain a parameter identifying the recipient’s asymmetric key.
- The ‘unprotected’ field MUST contain the ‘epk’ parameter.

12.4.1. ECDH

The basic mathematics for Elliptic Curve Diffie-Hellman can be found in [RFC6090].

ECDH is parameterized by the following:

- Curve Type/Curve: The curve selected controls not only the size of the shared secret, but the mathematics for computing the shared secret. The curve selected also controls how a point in the curve is represented and what happens for the identity points on the curve. In this specification, we allow for a number of different curves to be used. A set of curves are defined in Table 19. Since the only the math is changed by changing the curve, the curve is not fixed for any of the algorithm identifiers we define. Instead, it is defined by the points used.

- Ephemeral-static or static-static: The key agreement process may be done using either a static or an ephemeral key for the sender’s side. When using ephemeral keys, the sender MUST generate a new
ephemeral key for every key agreement operation. The ephemeral key is placed in the ‘ephemeral key’ parameter and MUST be present for all algorithm identifiers that use ephemeral keys. When using static keys, the sender MUST either generate a new random value or otherwise create a unique value to be placed in either in the KDF parameters or the context structure. For the KDF functions used, this means either in the ‘salt’ parameter for HKDF (Table 11) or in the ‘PartyU nonce’ parameter for the context structure (Table 12) MUST be present. (Both may be present if desired.) The value in the parameter MUST be unique for the key pair being used. It is acceptable to use a global counter that is incremented for every static-static operation and use the resulting value. When using static keys, the static key needs to be identified to the recipient. The static key can be identified either by providing the key (‘static key’) or by providing a key identifier for the static key (‘static key id’). Both of these parameters are defined in Table 17.

- Key derivation algorithm: The result of an ECDH key agreement process does not provide a uniformly random secret. As such, it needs to be run through a KDF in order to produce a usable key. Processing the secret through a KDF also allows for the introduction of both context material, how the key is going to be used, and one time material in the even to of a static-static key agreement.

- Key Wrap algorithm: The key wrap algorithm can be ‘none’ if the result of the KDF is going to be used as the key directly. This option, along with static-static, should be used if knowledge about the sender is desired. If ‘none’ is used, then the content layer encryption algorithm size is value fed to the context structure. Support is also provided for any of the key wrap algorithms defined in Section 12.2.1. If one of these options is used, the input key size to the key wrap algorithm is the value fed into the context structure as the key size.

The set of direct ECDH algorithms defined in this document are found in Table 16.

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>KDF</th>
<th>Ephemeral-Static</th>
<th>Key Wrap</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECDH-ES + HKDF-256</td>
<td>50</td>
<td>HKDF - SHA-256</td>
<td>yes</td>
<td>none</td>
<td>ECDH ES w/ HKDF - generate key directly</td>
</tr>
<tr>
<td>ECDH-ES + HKDF-512</td>
<td>51</td>
<td>HKDF - SHA -256</td>
<td>yes</td>
<td>none</td>
<td>ECDH ES w/ HKDF - generate key directly</td>
</tr>
<tr>
<td>ECDH-SS + HKDF-256</td>
<td>52</td>
<td>HKDF - SHA -256</td>
<td>no</td>
<td>none</td>
<td>ECDH ES w/ HKDF - generate key directly</td>
</tr>
<tr>
<td>ECDH-SS + HKDF-512</td>
<td>53</td>
<td>HKDF - SHA -256</td>
<td>no</td>
<td>none</td>
<td>ECDH ES w/ HKDF - generate key directly</td>
</tr>
<tr>
<td>ECDH-ES+A128KW</td>
<td>54</td>
<td>HKDF - SHA -256</td>
<td>yes</td>
<td>A128KW</td>
<td>ECDH ES w/ Concat KDF and AES Key wrap w/ 128 bit key</td>
</tr>
<tr>
<td>ECDH-ES+A192KW</td>
<td>55</td>
<td>HKDF - SHA -256</td>
<td>yes</td>
<td>A192KW</td>
<td>ECDH ES w/ Concat KDF and AES Key wrap w/ 192 bit key</td>
</tr>
<tr>
<td>ECDH-ES+A256KW</td>
<td>56</td>
<td>HKDF - SHA -256</td>
<td>yes</td>
<td>A256KW</td>
<td>ECDH ES w/ Concat KDF and AES Key wrap w/ 256 bit key</td>
</tr>
<tr>
<td>ECDH-SS+A128KW</td>
<td>57</td>
<td>HKDF - SHA -256</td>
<td>no</td>
<td>A128KW</td>
<td>ECDH SS w/ Concat KDF and AES Key wrap w/ 128 bit key</td>
</tr>
<tr>
<td>ECDH-</td>
<td>58</td>
<td>HKDF</td>
<td>no</td>
<td>A192KW</td>
<td>ECDH SS w/</td>
</tr>
<tr>
<td>SS+A192KW</td>
<td></td>
<td>- SHA</td>
<td></td>
<td>Concat KDF and AES Key wrap w/ 192 bit key</td>
<td></td>
</tr>
<tr>
<td>ECDH-59 HKDF no A256KW</td>
<td></td>
<td>- SHA</td>
<td></td>
<td>Encrypt KDF and AES Key wrap w/ 256 bit key</td>
<td></td>
</tr>
</tbody>
</table>

Table 16: ECDH Algorithm Values

<table>
<thead>
<tr>
<th>name</th>
<th>label</th>
<th>type</th>
<th>algorithm</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ephemeral</td>
<td>-1</td>
<td>COSE_Key</td>
<td>ECDH-ES</td>
<td>Ephemeral Public key for the sender</td>
</tr>
<tr>
<td>static key</td>
<td>-2</td>
<td>COSE_Key</td>
<td>ECDH-ES</td>
<td>Static Public key for the sender</td>
</tr>
<tr>
<td>static key id</td>
<td>-3</td>
<td>bstr</td>
<td>ECDH-SS</td>
<td>Static Public key identifier for the sender</td>
</tr>
</tbody>
</table>

Table 17: ECDH Algorithm Parameters

This document defines these algorithms to be used with the curves P-256, P-384, P-521. Implementations MUST verify that the key type and curve are correct. Different curves are restricted to different key types. Implementations MUST verify that the curve and algorithm are appropriate for the entities involved.

12.5. Key Agreement with KDF

Key Agreement with Key Wrapping uses a randomly generated CEK. The CEK is then encrypted using a Key Wrapping algorithm and a key derived from the shared secret computed by the key agreement algorithm.

The COSE_Enveloped structure for the recipient is organized as follows:
The ‘protected’ field is fed into the KDF context structure.

The plain text to be encrypted is the key from next layer down (usually the content layer).

The ‘alg’ parameter MUST be present in the layer.

A parameter identifying the recipient’s key SHOULD be present. A parameter identifying the sender’s key SHOULD be present.

12.5.1. ECDH

These algorithms are defined in Table 16.

13. Keys

The COSE_Key object defines a way to hold a single key object. It is still required that the members of individual key types be defined. This section of the document is where we define an initial set of members for specific key types.

For each of the key types, we define both public and private members. The public members are what is transmitted to others for their usage. We define private members mainly for the purpose of archival of keys by individuals. However, there are some circumstances in which private keys may be distributed by various entities in a protocol. Examples include: entities that have poor random number generation, centralized key creation for multi-cast type operations, and protocols in which a shared secret is used as a bearer token for authorization purposes.

Key types are identified by the ‘kty’ member of the COSE_Key object. In this document, we define four values for the member:

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC2</td>
<td>2</td>
<td>Elliptic Curve Keys w/ X,Y Coordinate pair</td>
</tr>
<tr>
<td>Symmetric</td>
<td>4</td>
<td>Symmetric Keys</td>
</tr>
<tr>
<td>Reserved</td>
<td>0</td>
<td>This value is reserved</td>
</tr>
</tbody>
</table>

Table 18: Key Type Values
13.1. Elliptic Curve Keys

Two different key structures could be defined for Elliptic Curve keys. One version uses both an x and a y coordinate, potentially with point compression. This is the traditional EC point representation that is used in [RFC5480]. The other version uses only the x coordinate as the y coordinate is either to be recomputed or not needed for the key agreement operation. Currently no algorithms are defined using this key structure.

<table>
<thead>
<tr>
<th>name</th>
<th>key type</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-256</td>
<td>EC2</td>
<td>1</td>
<td>NIST P-256 also known as secp256r1</td>
</tr>
<tr>
<td>P-384</td>
<td>EC2</td>
<td>2</td>
<td>NIST P-384 also known as secp384r1</td>
</tr>
<tr>
<td>P-521</td>
<td>EC2</td>
<td>3</td>
<td>NIST P-521 also known as secp521r1</td>
</tr>
</tbody>
</table>

Table 19: EC Curves

13.1.1. Double Coordinate Curves

The traditional way of sending EC curves has been to send either both the x and y coordinates, or the x coordinate and a sign bit for the y coordinate. The latter encoding has not been recommended in the IETF due to potential IPR issues. However, for operations in constrained environments, the ability to shrink a message by not sending the y coordinate is potentially useful.

For EC keys with both coordinates, the ‘kty’ member is set to 2 (EC2). The key parameters defined in this section are summarized in Table 20. The members that are defined for this key type are:

- **crv** contains an identifier of the curve to be used with the key. The curves defined in this document for this key type can be found in Table 19. Other curves may be registered in the future and private curves can be used as well.

- **x** contains the x coordinate for the EC point. The integer is converted to an octet string as defined in [SEC1]. Zero octets MUST NOT be removed from the front of the octet string.

- **y** contains either the sign bit or the value of y coordinate for the EC point. When encoding the value y, the integer is converted to an octet string (as defined in [SEC1]) and encoded as a CBOR bstr.
Leading zero octets MUST be preserved. When encoding the sign of y, the expression ‘y > 0’ is evaluated and encoded a CBOR boolean.

d contains the private key.

For public keys, it is REQUIRED that ‘crv’, ‘x’ and ‘y’ be present in the structure. For private keys, it is REQUIRED that ‘crv’ and ‘d’ be present in the structure. For private keys, it is RECOMMENDED that ‘x’ and ‘y’ also be present, but they can be recomputed from the required elements and omitting them saves on space.

<table>
<thead>
<tr>
<th>name</th>
<th>key type</th>
<th>value</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>crv</td>
<td>2</td>
<td>-1</td>
<td>int / tstr</td>
<td>EC Curve identifier - Taken from the COSE General Registry</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
<td>-2</td>
<td>bstr</td>
<td>X Coordinate</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
<td>-3</td>
<td>bstr / bool</td>
<td>Y Coordinate</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>-4</td>
<td>bstr</td>
<td>Private key</td>
</tr>
</tbody>
</table>

Table 20: EC Key Parameters

13.2. Symmetric Keys

Occasionally it is required that a symmetric key be transported between entities. This key structure allows for that to happen.

For symmetric keys, the ‘kty’ member is set to 3 (Symmetric). The member that is defined for this key type is:

k contains the value of the key.

This key structure contains only private key information, care must be taken that it is never transmitted accidentally. For public keys, there are no required fields. For private keys, it is REQUIRED that ‘k’ be present in the structure.
14. CBOR Encoder Restrictions

There has been an attempt to limit the number of places where the document needs to impose restrictions on how the CBOR Encoder needs to work. We have managed to narrow it down to the following restrictions:

- The restriction applies to the encoding the Sig_structure, the Enc_structure, and the MAC_structure.

- The rules for Canonical CBOR (Section 3.9 of RFC 7049) MUST be used in these locations. The main rule that needs to be enforced is that all lengths in these structures MUST be encoded such that they are encoded using definite lengths and the minimum length encoding is used.

- Applications MUST not generate messages with the same label used twice as a key in a single map. Applications MUST not parse and process messages with the same label used twice as a key in a single map. Applications can enforce the parse and process requirement by using parsers that will fail the parse step or by using parsers that will pass all keys to the application and the application can perform the check for duplicate keys.

15. Application Profiling Considerations

One of the issues that needs to be addressed is a requirement that a standard specify a set of algorithms that are required to be implemented. [CREF10] This is done to promote interoperability as it provides a minimal set of algorithms that all devices can be sure will exist at both ends. However, we have elected not to specify a set of mandatory algorithms in this document.

It is expected that COSE is going to be used in a wide variety of applications and on a wide variety of devices. Many of the constrained devices are going to be setup to use a small fixed set of algorithms, and this set of algorithms may not match those available on a device. We therefore have deferred to the application protocols the decision of what to specify for mandatory algorithms.
Since the set of algorithms in an environment of constrained devices may depend on what the set of devices are and how long they have been in operation, we want to highlight that application protocols will need to specify some type of discovery method of algorithm capabilities. The discovery method may be as simple as requiring preconfiguration of the set of algorithms to providing a discovery method built into the protocol. S/MIME provided a number of different ways to approach the problem:

- Advertising in the message (S/MIME capabilities) [RFC5751].
- Advertising in the certificate (capabilities extension) [RFC4262]
- Minimum requirements for the S/MIME, which have been updated over time [RFC2633][RFC5751]

16. IANA Considerations

16.1. CBOR Tag assignment

It is requested that IANA assign the following tags from the "Concise Binary Object Representation (CBOR) Tags" registry. It is requested that the tags be assigned in the 24 to 255 value range.

The tags to be assigned are:

<table>
<thead>
<tr>
<th>Tag Value</th>
<th>Data Item</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBD1</td>
<td>COSE_Sign</td>
<td>COSE Signed Data Object</td>
</tr>
<tr>
<td>TBD2</td>
<td>COSE_Enveloped</td>
<td>COSE Enveloped Data Object</td>
</tr>
<tr>
<td>TBD3</td>
<td>COSE_Encrypted</td>
<td>COSE Encrypted Data Object</td>
</tr>
<tr>
<td>TBD4</td>
<td>COSE_Mac</td>
<td>COSE Mac-ed Data Object</td>
</tr>
<tr>
<td>TBD5</td>
<td>COSE_Key, COSE_KeySet</td>
<td>COSE Key or COSE Key Set</td>
</tr>
</tbody>
</table>

16.2. COSE Header Parameter Registry

It is requested that IANA create a new registry entitled "COSE Header Parameters". The registry is to be created as Expert Review Required.

The columns of the registry are:
name  The name is present to make it easier to refer to and discuss
the registration entry. The value is not used in the protocol.
Names are to be unique in the table.

label  This is the value used for the label. The label can be either
an integer or a string. Registration in the table is based on the
value of the label requested. Integer values between 1 and 255
and strings of length 1 are designated as Standards Track Document
required. Integer values from 256 to 65535 and strings of length
2 are designated as Specification Required. Integer values of
greater than 65535 and strings of length greater than 2 are
designated as first come, first served. Integer values in the
range -1 to -65536 are delegated to the "COSE Header Algorithm
Label" registry. Integer values beyond -65536 are marked as
private use.

value  This contains the CBOR type for the value portion of the
label.

value registry  This contains a pointer to the registry used to
contain values where the set is limited.

description  This contains a brief description of the header field.

specification  This contains a pointer to the specification defining
the header field (where public).

The initial contents of the registry can be found in Table 1. The
specification column for all rows in that table should be this
document.

Additionally, the label of 0 is to be marked as ‘Reserved’.

16.3.  COSE Header Algorithm Label Table

It is requested that IANA create a new registry entitled "COSE Header
Algorithm Labels". The registry is to be created as Expert Review
Required.

The columns of the registry are:

name  The name is present to make it easier to refer to and discuss
the registration entry. The value is not used in the protocol.

algorithm  The algorithm(s) that this registry entry is used for.
This value is taken from the "COSE Algorithm Value" registry.
Multiple algorithms can be specified in this entry. For the
table, the algorithm, label pair MUST be unique.
label  This is the value used for the label. The label is an integer in the range of -1 to -65536.

value  This contains the CBOR type for the value portion of the label.

value registry  This contains a pointer to the registry used to contain values where the set is limited.

description  This contains a brief description of the header field.

specification  This contains a pointer to the specification defining the header field (where public).

The initial contents of the registry can be found in Table 11, Table 12, and Table 17. The specification column for all rows in that table should be this document.

16.4. COSE Algorithm Registry

It is requested that IANA create a new registry entitled "COSE Algorithm Registry". The registry is to be created as Expert Review Required.

The columns of the registry are:

value  The value to be used to identify this algorithm. Algorithm values MUST be unique. The value can be a positive integer, a negative integer or a string. Integer values between 0 and 255 and strings of length 1 are designated as Standards Track Document required. Integer values from 256 to 65535 and strings of length 2 are designated as Specification Required. Integer values of greater than 65535 and strings of length greater than 2 are designated as first come, first served. Integer values in the range -1 to -65536 are delegated to the "COSE Header Algorithm Label" registry. Integer values beyond -65536 are marked as private use.

description  A short description of the algorithm.

specification  A document where the algorithm is defined (if publicly available).

The initial contents of the registry can be found in Table 8, Table 7, Table 9, Table 4, Table 5, Table 6, Table 13, Table 14, Table 15, and Table 16. The specification column for all rows in that table should be this document.
16.5. COSE Key Common Parameter Registry

It is requested that IANA create a new registry entitled "COSE Key Common Parameter" Registry. The registry is to be created as Expert Review Required.

The columns of the registry are:

- name: This is a descriptive name that enables easier reference to the item. It is not used in the encoding.
- label: The value to be used to identify this algorithm. Key map labels MUST be unique. The label can be a positive integer, a negative integer or a string. Integer values between 0 and 255 and strings of length 1 are designated as Standards Track Document required. Integer values from 256 to 65535 and strings of length 2 are designated as Specification Required. Integer values of greater than 65535 and strings of length greater than 2 are designated as first come, first served. Integer values in the range -1 to -65536 are used for key parameters specific to a single algorithm delegated to the "COSE Key Parameter Label" registry. Integer values beyond -65536 are marked as private use.
- CBOR Type: This field contains the CBOR type for the field.
- registry: This field denotes the registry that values come from, if one exists.
- description: This field contains a brief description for the field.
- specification: This contains a pointer to the public specification for the field if one exists.

This registry will be initially populated by the values in Section 7.1. The specification column for all of these entries will be this document.

16.6. COSE Key Type Parameter Registry

It is requested that IANA create a new registry "COSE Key Type Parameters". The registry is to be created as Expert Review Required.

The columns of the table are:

- key type: This field contains a descriptive string of a key type. This should be a value that is in the COSE General Values table and is placed in the ‘kty’ field of a COSE Key structure.
name  This is a descriptive name that enables easier reference to the item. It is not used in the encoding.

label  The label is to be unique for every value of key type. The range of values is from -256 to -1. Labels are expected to be reused for different keys.

CBOR type  This field contains the CBOR type for the field

description  This field contains a brief description for the field

specification  This contains a pointer to the public specification for the field if one exists

This registry will be initially populated by the values in Table 20 and Table 21. The specification column for all of these entries will be this document.

16.7.  COSE Elliptic Curve Registry

It is requested that IANA create a new registry "COSE Elliptic Curve Parameters". The registry is to be created as Expert Review Required.

The columns of the table are:

name  This is a descriptive name that enables easier reference to the item. It is not used in the encoding.

value  This is the value used to identify the curve. These values MUST be unique. The integer values from -256 to 255 are designated as Standards Track Document Required. The integer values from 256 to 65535 and -65536 to -257 are designated as Specification Required. Integer values over 65535 are designated as first come, first served. Integer values less than -65536 are marked as private use.

key type  This designates the key type(s) that can be used with this curve.

description  This field contains a brief description of the curve.

specification  This contains a pointer to the public specification for the curve if one exists.

This registry will be initially populated by the values in Table 18. The specification column for all of these entries will be this document.
16.8. Media Type Registrations

16.8.1. COSE Security Message

This section registers the "application/cose" and "application/cose+cbor" media types in the "Media Types" registry. [CREF11] These media types are used to indicate that the content is a COSE_MSG. [CREF12]

Type name: application

Subtype name: cose

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: See the Security Considerations section of RFC TBD.

Interoperability considerations: N/A

Published specification: RFC TBD

Applications that use this media type: To be identified

Fragment identifier considerations: N/A

Additional information:

* Magic number(s): N/A

* File extension(s): cbor

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

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

Intended usage: COMMON

Restrictions on usage: N/A

Author: Jim Schaad, ietf@augustcellars.com

Change Controller: IESG
Provisional registration? No
Type name: application
Subtype name: cose+cbor
Required parameters: N/A
Optional parameters: N/A
Encoding considerations: binary
Security considerations: See the Security Considerations section of RFC TBD.
Interoperability considerations: N/A
Published specification: RFC TBD
Applications that use this media type: To be identified
Fragment identifier considerations: N/A
Additional information:
* Magic number(s): N/A
* File extension(s): cbor
* Macintosh file type code(s): N/A
Person & email address to contact for further information: iesg@ietf.org
Intended usage: COMMON
Restrictions on usage: N/A
Author: Jim Schaad, ietf@augustcellars.com
Change Controller: IESG
Provisional registration? No
16.8.2. COSE Key media type

This section registers the "application/cose-key+cbor" and "application/cose-key-set+cbor" media types in the "Media Types" registry. These media types are used to indicate, respectively, that content is a COSE_Key or COSE_KeySet object.

Type name: application

Subtype name: cose-key+cbor

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: See the Security Considerations section of RFC TBD.

Interoperability considerations: N/A

Published specification: RFC TBD

Applications that use this media type: To be identified

Fragment identifier considerations: N/A

Additional information:

* Magic number(s): N/A

* File extension(s): cbor

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

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

Intended usage: COMMON

Restrictions on usage: N/A

Author: Jim Schaad, ietf@augustcellars.com

Change Controller: IESG

Provisional registration? No
Type name: application
Subtype name: cose-key-set+cbor
Required parameters: N/A
Optional parameters: N/A
Encoding considerations: binary
Security considerations: See the Security Considerations section of RFC TBD.
Interoperability considerations: N/A
Published specification: RFC TBD
Applications that use this media type: To be identified
Fragment identifier considerations: N/A
Additional information:
* Magic number(s): N/A
* File extension(s): cbor
* Macintosh file type code(s): N/A
Person & email address to contact for further information: iesg@ietf.org
Intended usage: COMMON
Restrictions on usage: N/A
Author: Jim Schaad, ietf@augustcellars.com
Change Controller: IESG
Provisional registration? No

16.9. CoAP Content Format Registrations

This section registers a set of content formats for CoAP. ID assignment in the 24–255 range is requested.
17. Security Considerations

There are security considerations:

1. Protect private keys.

2. MAC messages with more than one recipient means one cannot figure out which party sent the message.

3. Use of a direct key with other recipient structures hands the key to the other recipients.

4. Use of direct ECDH direct encryption is easy for people to leak information on if there are other recipients in the message.

5. Considerations about protected vs unprotected header fields. Why the algorithm parameter needs to be protected.

6. Need to verify that: 1) the kty field of the key matches the key and algorithm being used, 2) the kty field needs to be included in the trust decision as well as the other key fields, and 3) the algorithm is included in the trust decision.

18. References

18.1. Normative References


18.2. Informative References


[RFC3610] Whiting, D., Housley, R., and N. Ferguson, "Counter with
CBC-MAC (CCM)", RFC 3610, DOI 10.17487/RFC3610, September

[RFC4231] Nystrom, M., "Identifiers and Test Vectors for HMAC-SHA-
224, HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512",
RFC 4231, DOI 10.17487/RFC4231, December 2005,

Multipurpose Internet Mail Extensions (S/MIME)
Capabilities", RFC 4262, DOI 10.17487/RFC4262, December

FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007,

Multipurpose Internet Mail Extensions (S/MIME)
Capabilities", RFC 5480, DOI 10.17487/RFC5480, March 2009,

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70,
RFC 5652, DOI 10.17487/RFC5652, September 2009,

[RFC5751] Ramsdell, B. and S. Turner, "Secure/Multipurpose Internet
Mail Extensions (S/MIME) Version 3.2 Message

[RFC5752] Turner, S. and J. Schaad, "Multiple Signatures in
Cryptographic Message Syntax (CMS)", RFC 5752,
DOI 10.17487/RFC5752, January 2010,

Key Derivation Function (HKDF)", RFC 5869,
DOI 10.17487/RFC5869, May 2010,

[RFC5990] Randall, J., Kaliski, B., Brainard, J., and S. Turner,
"Use of the RSA-KEM Key Transport Algorithm in the
Cryptographic Message Syntax (CMS)", RFC 5990,
DOI 10.17487/RFC5990, September 2010,

Schaad                    Expires May 25, 2016                 [Page 69]


Appendix A. CDDL Grammar

For people who prefer using a formal language to describe the syntax of the CBOR, in this section a CDDL grammar is given that corresponds to [I-D.greevenbosch-appsawg-cbor-cddl]. This grammar is informational. In the event of differences between this grammar and the prose, the prose is considered to be authoritative.

The collected CDDL can be extracted from the XML version of this document via the following XPath expression below. (Depending on the XPath evaluator one is using, it may be necessary to deal with &gt; as an entity.)

```
//artwork[@type='CDDL']/text()
```

; This is define to make the tool quieter

```
Internal_Types = Sig_structure / Enc_structure / MAC_structure / COSE_KDF_Context
```

Appendix B. Three Levels of Recipient Information

All of the currently defined recipient algorithms classes only use two levels of the COSE_Encrypt structure. The first level is the message content and the second level is the content key encryption. However, if one uses a recipient algorithm such as RSA-KEM (see Appendix A of RSA-KEM [RFC5990], then it make sense to have three levels of the COSE_Encrypt structure.

These levels would be:

- Level 0: The content encryption level. This level contains the payload of the message.
- Level 1: The encryption of the CEK by a KEK.
- Level 2: The encryption of a long random secret using an RSA key and a key derivation function to convert that secret into the KEK.

This is an example of what a triple layer message would look like.

The message has the following layers:
Level 0: Has a content encrypted with AES-GCM using a 128-bit key.
Level 1: Uses the AES Key wrap algorithm with a 128-bit key.
Level 2: Uses ECDH Ephemeral-Static direct to generate the level 1 key.

In effect this example is a decomposed version of using the ECDH-ES+A128KW algorithm.

Size of binary file is 216 bytes
998{
  / protected / h’a10101’ / {
    \ alg \ 1:1 \ AES-GCM 128 \
  } / ,
  / unprotected / {
    / iv / 5:h’02d1f7e6f26c43d4868d87ce’
  },
  / ciphertext / h’64f84d913ba60a76070a9a48f26e97e863e285295a44320878caceb0763a334806857c67’,
  / recipients / [ ]
  } 
}

Appendix C. Examples

The examples can be found at https://github.com/cose-wg/Examples. The file names in each section correspond the same file names in the repository. I am currently still in the process of getting the examples up there along with some control information for people to be able to check and reproduce the examples.
Examples may have some features that are in question but not yet incorporated in the document.

To make it easier to read, the examples are presented using the CBOR’s diagnostic notation rather than a binary dump. A ruby based tool exists to convert between a number of formats. This tool can be installed with the command line:

```
gem install cbor-diag
```

The diagnostic notation can be converted into binary files using the following command line:

```
diag2cbor < inputfile > outputfile
```

The examples can be extracted from the XML version of this document via an XPath expression as all of the artwork is tagged with the attribute type=’CBORdiag’.

C.1. Examples of MAC messages

C.1.1. Shared Secret Direct MAC

This example uses the following:

- MAC: AES-CMAC, 256-bit key, truncated to 64 bits
- Recipient class: direct shared secret
- File name: Mac-04

Size of binary file is 73 bytes
C.1.2. ECDH Direct MAC

This example uses the following:

- MAC: HMAC w/SHA-256, 256-bit key
- Recipient class: ECDH key agreement, two static keys, HKDF w/ context structure

Size of binary file is 217 bytes
996(
  [  
    / protected / h’a10104’ / {  
      alg / 1:4 \ HMAC 256/256 \  
    } / ,  
    / unprotected / {},  
    / payload / ’This is the content.’,  
    / tag / h’2ba937ca03d76c3dbad30cfcbaeeef586f9c0f9ba616ad67e9205d38576ad9930’,  
    / recipients / [  
      [  
        / protected / h’,  
        / unprotected / {  
          / alg / 1:52 / ECDH-SS + HKDF-256 /,  
          / kid / 4:’meriadoc.brandybuck@buckland.example’,  
          / static kid / -3:’peregrin.took@tuckborough.example’,  
          / apu”:h’4d8553e7e74f3c6a3a9dd3ef286a8195cbf8a23d19558ccfec7d34b824f42d92bd06bd2c7f0271f0214e141fb779ae2856abf585a58368b017e7f2a9e5ce4db5’  
        ]  
      ]  
    ]
  ]
)

C.1.3. Wrapped MAC

This example uses the following:

- MAC: AES-MAC, 128-bit key, truncated to 64 bits
- Recipient class: AES keywrap w/ a pre-shared 256-bit key

Size of binary file is 124 bytes
C.1.4. Multi-recipient MAC message

This example uses the following:

- MAC: HMAC w/ SHA-256, 128-bit key
- Recipient class: Uses three different methods
  1. ECDH Ephemeral-Static, Curve P-521, AES-Key Wrap w/ 128-bit key
  2. AES-Key Wrap w/ 256-bit key

Size of binary file is 374 bytes
C.2. Examples of Encrypted Messages

```c
996(
  [  
    / protected / h’a10104’/ {  
      alg \ 1:4 \ HMAC 256/256 \  
    }/,  
    / unprotected / {},  
    / payload / ’This is the content.’,  
    / tag / h’7aaa6e7456873061f0a7de21ff0c0658d401a68da738dd8937486
51983celd0’,  
    / recipients / [  
      [  
        / protected / h’”,  
        / unprotected / {},  
        / alg / 1:55 / ECHD-ES+A192KW /,  
        / kid / 4:’bilbo.baggins@hobbiton.example’,  
        -1:{  
          1:2,  
          -1:3,  
          -2:h’43b12669acac3fd27898ffba0bcd2e6c366d53bc4db71f909a7
59304acfb5e18cdd7ba0b13f8c7636271a6924b1ac63c02688075b55ef2d613574e
7dc242f79c3’,  
          -3:h’812dd649f4ef32b11014d74010a954689c6b6e8785b333d1ab4
4f22b9d1091ae8fc8ae40b687e5cfbe7ee6f8b47918a07bb04e9f5b1a51a334a16bc
09777434113’  
        }},  
      ],  
      / ciphertext / h’f20ad9c96134f3c6be4f75e7101c0ecc5efa071ff20
a87fd1ac2851094lee0376573e2b384b56b99’,  
    ]},  
    [  
      / protected / h’”,  
      / unprotected / {},  
      / alg / 1:-5 / A256KW /,  
      / kid / 4:’018c0ae5-4d9b-471b-bfd6-eef314bc7037’  
    },  
    / ciphertext / h’0b2c7cfc0e04e98276342d6476a7723c090dfdd15f9a
518e7736549e998370695e6d6a83b4ae50bb’  
  ]
)
```

Schaad  Expires May 25, 2016  [Page 78]
C.2.1. Direct ECDH

This example uses the following:

- CEK: AES-GCM w/ 128-bit key
- Recipient class: ECDH Ephemeral-Static, Curve P-256

Size of binary file is 184 bytes

```
998,
  / protected / h’a10101’ / {
    \ alg \ 1:1 \ AES-GCM \ 128 \ }
  / unprotected / {
    / iv / 5:h’c9cf4df2fe6c632bf7886413’
  },
  / ciphertext / h’45fce2814311024d3a479e7d3eed063850f3f0b9f3f9486
  77e3ae9869bcf9ff4e1763812’,
  / recipients / {
    / protected / h’,
    / unprotected / {
      / alg / 1:50 / ECDH-ES + HKDF-256 /,
      / kid / 4:’meriadoc.brandybuck@buckland.example’,
      / ephemeral / -1:{
        / kty / 1:2,
        / crv / -1:1,
        / x / -2:h’98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbf
        bf054e1c7b4d91d6280’,
        / y / -3:h’f01400b089867804b8e9fc96c3932161f1934f4223069
        170d924b7e03bf822bb’
      }
    },
    / ciphertext / h’
  }
}
```

C.2.2. Direct plus Key Derivation

This example uses the following:

- CEK: AES-CCM w/128-bit key, truncate the tag to 64 bits

```
Schaad                Expires May 25, 2016                 [Page 79]```
Recipient class: Use HKDF on a shared secret with the following implicit fields as part of the context.

- APU identity: "lighting-client"
- APV identity: "lighting-server"
- Supplementary Public Other: "Encryption Example 02"

Size of binary file is 97 bytes

```
998(
  / protected / h'1010a' / {
    \ alg \ 1:10 \ AES-CCM-16-64-128 \
  }) /,
/ unprotected / {
  / iv / 5:h'89f52f65a1c580933b5261a7'
},
/ ciphertext / h'7b9dcfa42c4e1d3182c402dc18ef8b5637de4fb62cf1dd156ea6e60',
/ recipients / [ [ / protected / h'', / unprotected / { / alg / 1:"dir+kdf", / kid / 4:'our-secret', -20:'aabbccddeeffgghh' }, / ciphertext / h'' ] ]
)
```

C.2.3. Counter Signature on Encrypted Content

This example uses the following:

- CEK: AES-GCM w/ 128-bit key
- Recipient class: ECDH Ephemeral-Static, Curve P-256

C.2.4. Encrypted Content w/ Implicit Recipient

This example uses the following:

- CEK: AES-GCM w/ 128-bit key
C.3. Examples of Signed Message

C.3.1. Single Signature

This example uses the following:

- Signature Algorithm: ECDSA w/ SHA-256, Curve P-256-1

Size of binary file is 105 bytes

```
999(
  [  
  / protected / h’’, 
  / unprotected / {}, 
  / payload / ’This is the content.’, 
  / signatures / [{ 
    / protected / h’a10126’ / { 
      / alg \ 1:-7 \ ES256 \ }, 
    / unprotected / { 
      / kid / 4:’11’ 
    }, 
    / signature / h’4358e9e92b46d45134548b6e3b4eae3d2f801bce85236c7aab42968ad8e3e92400873ed761735222a6d1f442c4bb3a3151946b16900048572455e65451d89aab7’ 
  } ]
)
```

C.3.2. Multiple Signers

This example uses the following:

- Signature Algorithm: ECDSA w/ SHA-256, Curve P-256-1
- Signature Algorithm: ECDSA w/ SHA-512, Curve P-521

Size of binary file is 277 bytes
C.3.3. Counter Signature

This example uses the following:

- Signature Algorithm: ECDSA w/ SHA-256, Curve P-256-1

C.4. COSE Keys

C.4.1. Public Keys

This is an example of a COSE Key set. This example includes the public keys for all of the previous examples.

In order the keys are:
- An EC key with a kid of "meriadoc.brandybuck@buckland.example"
- An EC key with a kid of "peregrin.took@tuckborough.example"
- An EC key with a kid of "bilbo.baggins@hobbiton.example"
- An EC key with a kid of "11"

Size of binary file is 481 bytes
C.4.2. Private Keys

This is an example of a COSE Key set. This example includes the private keys for all of the previous examples.

In order the keys are:

Schaad                    Expires May 25, 2016                 [Page 84]
- An EC key with a kid of "meriadoc.brandybuck@buckland.example"
- A shared-secret key with a kid of "our-secret"
- An EC key with a kid of "peregrin.took@tuckborough.example"
- A shared-secret key with a kid of "018c0ae5-4d9b-471b-bfd6-eef314bc7037"
- An EC key with a kid of "bilbo.baggins@hobbiton.example"
- An EC key with a kid of "11"

Size of binary file is 782 bytes

```
[
  {
    /kty/ 1:2,
    /kid/ 2:’meriadoc.brandybuck@buckland.example’,
    /crv/ -1:1,
    /x/ -2:h’65eda5a12577c2bae829437fe338701a10aa375e1bb5b5de108d
e439c08551d’,
    /y/ -3:h’1e52ed75701163f7f9e40ddf9f341b3dc9ba860af7e0ca7ca7e9e
ced0084d19c’,
    /d/ -4:h’aff907c99f9ad3aae6c4cdef21122bce2b68b5283e6907154ad91
1840fa208cf’
  },
  {
    /kty/ 1:4,
    /kid/ 2:’our-secret’,
    /k/ -1:h’849b576219da48de646d07dbb533566e976686457c1491be3a76d
cea6c427188’
  },
  {
    /kty/ 1:2,
    /kid/ 2:’bilbo.baggins@hobbiton.example’,
    /crv/ -1:3,
    /x/ -2:h’0072992cb3ac08ecf3e5c63de0dc0d51a8c1f79ef2f82f9f3c73
7bf5de786671ecac625fe8257bbd0394644caaa3aaaf8f27a4585fbbcad0f24576200
85e5c8f42ad’,
    /y/ -3:h’01dca6947bce88bc5790485ac97427342bc35f887d86d65a08937
7e247e60bba55e4e8501e2ada5724ac51d6909008033ebe10ac999b9d7f5cc2519f3
fe1ad9475’,
    /d/ -4:h’00085138ddabf5ca975f5860f91a08e91d65f9a76ad4018766a4
76680b55cd339e8ab6c72b5facdb2a2a50ac25bd086647dd3e2e6e99e84ca2c3609f
df177feb26d’
  }
]
```
/ kty / 1:2,
/ crv / -1:1,
/ kid / 2:'peregrin.took@tuckborough.example',
/ x / -2:h'98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbbf054e1c7b4d91d6280',
/ y / -3:h'f01400b089867804b8e9fc96c3932161f1934f4223069170d924be03bf822bb',
/ d / -4:h'02d1f7e6f2643d4868d87ce2b2353161740aacf1f7163647984b522a848df1c3'
}
{
/ kty / 1:4,
/ kid / 2:'018c0ae5-4d9b-471b-bfd6-eeef314bc7037',
/ k / -1:h'849b57219daae864d6d07dbb5533566e976686457c1491be3a76dcea6c427188'
}
{
/ kty / 1:2,
/ kid / 2:'11',
/ crv / -1:1,
/ x / -2:h'bac5b11cad8f99f9c72b05cf4b9e264244dc189f745228255a219a86d6a09ef',
/ y / -3:h'20138bf82d1b6d562be0fa54ab7804a3a64b6d72ccfed6bf6e6d28bbf6c117e',
/ d / -4:h'57c92077664146e87676c9520d054aa93c3afb04e306705db6090308507b4d3'
}

Appendix D.  Document Updates

D.1.  Version -06 to -08
  o Redefine sequence number into a the Partial IV.

D.2.  Version -06 to -07
  o Editorial Changes
    o Make new IANA registries be Expert Review

D.3.  Version -05 to -06
  o Remove new CFRG Elliptical Curve key agreement algorithms.
    o Remove RSA algorithms
    o Define a creation time and sequence number for discussions.
Remove message type field from all structures.

Define CBOR tagging for all structures with IANA registrations.

D.4. Version -04 to -05

- Removed the jku, x5c, x5t, x5t#256, x5u, and jwk headers.
- Add enveloped data vs encrypted data structures.
- Add counter signature parameter.

D.5. Version -03 to -04

- Change top level from map to array.
- Eliminate the term "key management" from the document.
- Point to content registries for the 'content type' attribute
- Push protected field into the KDF functions for recipients.
- Remove password based recipient information.
- Create EC Curve Registry.

D.6. Version -02 to -03

- Make a pass over all of the algorithm text.
- Alter the CDDL so that Keys and KeySets are top level items and the key examples validate.
- Add sample key structures.
- Expand text on dealing with Externally Supplied Data.
- Update the examples to match some of the renumbering of fields.

D.7. Version -02 to -03

- Add a set of straw man proposals for algorithms. It is possible/expected that this text will be moved to a new document.
- Add a set of straw man proposals for key structures. It is possible/expected that this text will be moved to a new document.
- Provide guidance on use of externally supplied authenticated data.
o Add external authenticated data to signing structure.

D.8. Version -01 to -2

o Add first pass of algorithm information

o Add direct key derivation example.

D.9. Version -00 to -01

o Add note on where the document is being maintained and contributing notes.

o Put in proposal on MTI algorithms.

o Changed to use labels rather than keys when talking about what indexes a map.

o Moved nonce/IV to be a common header item.

o Expand section to discuss the common set of labels used in COSE_Key maps.

o Start marking element 0 in registries as reserved.

o Update examples.

Editorial Comments

[CREF1] JLS: Need to check this list for correctness before publishing.

[CREF2] JLS: I have not gone through the document to determine what needs to be here yet. We mostly want to grab terms that are used in unusual ways or are not generally understood.

[CREF3] JLS: A completest version of this grammar would list the options available in the protected and unprotected headers. Do we want to head that direction?

[CREF4] JLS: Expand CoAP?

[CREF5] Hannes: Ensure that the list of examples only includes items that are implemented in this specification. Check the other places where such lists occur and ensure that they also follow this rule.

[CREF6] JLS: Don’t talk about items which we do not define in this specification? Only talk about...
[CREF7] JLS: Restrict to the set of supported parameters.

[CREF8] JLS: We can really simplify the grammar for COSE_Key to be just the kty (the one required field) and the generic item. The reason to do this is that it makes things simpler. The reason not to do this says that we really need to add a lot more items so that a grammar check can be done that is more tightly enforced.

[CREF9] Ilari: Look to see if we need to be clearer about how the fields defined in the table are transported and thus why they have labels.

[CREF10] JLS: It would be possible to extend this section to talk about those decisions that an application needs to think about rather than just talking about MTI algorithms.

[CREF11] JLS: Should we register both or just the cose+cbor one?

[CREF12] JLS: Should we create the equivalent of the smime-type parameter to identify the inner content type?

Author’s Address

Jim Schaad
August Cellars

Email: ietf@augustcellars.com
Abstract

This memo defines Object Security of CoAP (OSCOAP), a method for protection of request and response message exchanges of the Constrained Application Protocol (CoAP) using data object security. OSCOAP provides end-to-end encryption, integrity and replay protection to CoAP payload, options and header fields, and a secure binding between CoAP request and response messages. The use of OSCOAP is signaled with the Object-Security option, also defined in this memo.

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 http://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 April 21, 2016.

Copyright Notice

Copyright (c) 2015 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 (http://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 Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction .................................................. 3
   1.1. Terminology .............................................. 4
2. Background .................................................... 5
3. The Object-Security Option .................................... 6
4. Secure Message Format ........................................ 7
   4.1. Secure Message Header ................................... 7
   4.2. Secure Message Body and Tag ............................ 8
   4.2.1. Integrity Protection Only ............................ 8
   4.2.2. Encryption and Integrity Protection ................ 8
5. CoAP Message Protection ....................................... 9
   5.1. Integrity Protection Only ............................... 9
   5.1.1. Protected CoAP message formatting ................ 9
   5.1.2. Secure Message formatting .......................... 10
   5.1.3. Integrity Protection and Verification ............. 10
   5.2. Encryption and Integrity Protection .................. 10
   5.2.1. Protected CoAP message formatting ................ 10
   5.2.2. Secure Message formatting .......................... 11
6. Protected CoAP Message Fields ............................... 12
   6.1. Protected CoAP Header Fields ........................... 12
   6.2. Protected CoAP Options ................................ 12
   6.2.1. Integrity Protection ............................... 13
   6.2.2. Encryption ......................................... 15
7. Replay Protection and Freshness ............................. 15
   7.1. Replay Protection ...................................... 15
   7.2. Freshness ............................................. 16
8. Security Considerations ....................................... 16
9. Privacy Considerations ....................................... 18
10. IANA Considerations ......................................... 18
11. Acknowledgments ............................................. 19
12. References .................................................. 19
   12.1. Normative References .................................. 19
   12.2. Informative References ............................... 20
Appendix A. COSE Profile of SM .................................. 21
   A.1. Integrity Protection Only .............................. 21
   A.1.1. COSE_Sign ........................................ 21
   A.1.2. COSE_mac ........................................ 22
   A.2. Encryption and Integrity Protection: COSE_enveloped ... 22
   A.3. COSE Optimizations .................................... 23
Appendix B. Comparison of message sizes ...................... 25
1. Introduction

The Constrained Application Protocol CoAP [RFC7252] was designed with a constrained RESTful environment in mind. CoAP references DTLS [RFC6347] for securing the message exchanges. Two prominent features of CoAP, store-and-forward and publish-subscribe exchanges, are problematic to secure with DTLS and transport layer security. As DTLS offers hop-by-hop security, in case of store-and-forward exchanges it necessitates a trusted intermediary. Securing publish-subscribe CoAP exchanges with DTLS requires the use of the keep-alive mechanism which incurs additional overhead and actually takes away most of the benefits of asynchronous communication.

The pervasive monitoring debate has illustrated the need to protect data also from trustworthy intermediary nodes as they can be compromised. The community has reacted strongly to the revelations, and new solutions must consider this attack [RFC7258] and include encryption by default.

This memo defines Object Security of CoAP (OSCOAP) a data object based communication security solution complementing DTLS and supporting secure messaging end-to-end across intermediary nodes. OSCOAP may be used in very constrained settings where DTLS cannot be supported. OSCOAP can also be combined with DTLS thus enabling, for example, end-to-end security of CoAP payload in combination with hop-by-hop protection of the entire CoAP message during transport between end-point and intermediary node.

OSCOAP provides end-to-end encryption, integrity and replay protection to CoAP payload, options and header fields, and a secure binding between CoAP request and response messages. Using this method the unprotected CoAP message is transformed into a protected
CoAP message, which contains a secure data object protecting the unprotected message, and which is sent instead of the unprotected message. The use of OSCOAP is signaled with the Object-Security option, also defined in this memo.

1.1. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119]. These words may also appear in this document in lowercase, absent their normative meanings.

Certain security-related terms are to be understood in the sense defined in [RFC4949]. These terms include, but are not limited to, "authentication", "authorization", "confidentiality", "(data) integrity", "message authentication code", and "verify". For "signature", see below.

RESTful terms, such as "resource" or "representation", are to be understood as used in HTTP [RFC7231] and CoAP.

Terminology for constrained environments, such as "constrained device", "constrained-node network", is defined in [RFC7228].

Terminology for authentication and authorization in constrained environments, such as "Authorization Server", "Resource Server", etc, is defined in [I-D.ietf-ace-actors].

The CoAP option Object-Security and the Secure Message (SM) format are defined in this memo.

Two different scopes of object security are defined:

- OSCOAP = object security of CoAP, signaled with the Object-Security option
- OSCON = object security of content, signaled with Content Format/Media Type set to application/oscon.

OSCON is defined in Appendix C and included for comparison with OSCOAP.

The COSE message format is defined in [I-D.ietf-cose-msg].
2. Background

The background for this work is provided by the use cases and architecture in [I-D.ietf-ace-usecases] and [I-D.ietf-ace-actors]. The focus of this memo is on end-to-end security in constrained environments in the presence of intermediary nodes.

For constrained-node networks there may be several reasons for messages to be cached or stored in one node and later forwarded.

For example, connectivity between the nodes may be intermittent, or some node may be sleeping at the time when the message should have been forwarded (see e.g. [I-D.ietf-ace-usecases] sections 2.1.1, and 2.5.1). Also, the architectural model or protocol applied may require an intermediary node which breaks security on transport layer (see e.g. [I-D.ietf-ace-usecases] sections 2.1.1, and 2.5.2). Examples of intermediary nodes include forward proxies, reverse proxies, pub-sub brokers, HTTP-CoAP cross-proxies, and SMS servers.

Based on these examples the following security requirements have been identified:

1. The payload shall be integrity protected and should be encrypted end-to-end from sender to receiver.

2. It shall be possible for an intended receiver to detect if it has received this message previously, i.e. replay protection.

3. The CoAP options which are not intended to be changed by an intermediary node shall be integrity protected between Client and Server.

4. The CoAP options which are not intended to be read by an intermediary node shall be encrypted between Client and Server.

5. The CoAP header fields "Code" and "Version" shall be integrity protected between Client and Server.

6. A Client shall be able to verify that a message is the response to a particular request the Client made.

In this list above, requirements 1-2 deals essentially with protecting the CoAP payload only, whereas 3-6 deals with protecting an entire CoAP request-response exchange, including also CoAP options and header fields.

Object Security of CoAP (OSCOAP), which is the main focus of this memo, addresses all requirements above by defining a method for
encryption, integrity protection and replay protection of CoAP payload, options and header fields, and a secure binding between CoAP request and response messages. OSCOAP consists of:

- the Object-Security option, indicating that OSCOAP is being used;
- a compact cryptographic message format called "Secure Message", based on the COSE message format ([I-D.ietf-cose-msg]); and
- a scheme for transforming an unprotected CoAP message into a protected CoAP message, which contains the Object-Security option and a Secure Message protecting CoAP payload, options and header fields.

The same method can be applied to payload only of individual messages, targeting only requirements 1-2 above. We call this object security of content (OSCON) and it is defined in Appendix C.

Examples of the use of OSCOAP and OSCON are given in Appendix D.

3. The Object-Security Option

In order to end-to-end protect CoAP message exchanges including options and headers, a new CoAP option is introduced: the Object-Security option. The Object-Security option indicates that OSCOAP is used, i.e. that certain CoAP Header fields, Options and Payload (if present) are integrity and replay protected and potentially encrypted, using a cryptographic message format called the Secure Message format Section 4.

This option is critical, safe to forward, it is not part of a cache key, and it is not repeatable. Figure 1 illustrates the structure of this option.

<table>
<thead>
<tr>
<th>No.</th>
<th>C</th>
<th>U</th>
<th>N</th>
<th>R</th>
<th>Name</th>
<th>Format</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBD</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>Object-Security</td>
<td>opaque</td>
<td>0, TBD</td>
</tr>
</tbody>
</table>

C=Critical, U=Unsafe, N=NoCacheKey, R=Repeatable

Figure 1: The Object-Security Option

The length of the option depends on the specific choice of the Secure Message format. Length 0 indicates that the Secure Message is the CoAP Payload of the message, and is used when the CoAP message type used supports payload.
4. Secure Message Format

There exist already standardized and draft content formats for encryption and integrity protection of data such as CMS [RFC5652], JWS [RFC7515], JWE [RFC7516], and COSE [I-D.ietf-cose-msg].

Current CMS and JWx objects are undesirably large for very constrained devices. Large messages have a negative impact on memory and storage in constrained devices, packet fragmentation in constrained-node networks due to limited frame sizes, and increased energy consumption due to more data transmission and reception. The candidate for use with object security of CoAP messages is the COSE message format [I-D.ietf-cose-msg].

Pending an optimized and stable version of the COSE message format this memo defines the SM format to refer to a content format for encrypted and integrity protected data, and also includes a unique transaction identifier for replay protection. Appendix A shows a profile of the COSE message format which complies with the Secure Message format.

A Secure Message (SM) SHALL consist of Header, Body and Tag.

4.1. Secure Message Header

The following parameters SHALL be included in the SM Header:

- Context Identifier (CID). This parameter identifies the sender security context including the cipher suite, key(s) and additional algorithm specific parameters used to protect the message. Each client and server communicating using OSCOAP has two contexts, one for sending and one for receiving.

- Sequence Number (SEQ). The Sequence Number parameter enumerates the Secure Messages sent associated to a Context Identifier, and is used for replay protection and uniqueness of nonce. The start sequence number SHALL be 0. For a given key, any Sequence Number MUST NOT be used more than once.

The granularity of "sender" - what is being identified with the Context Identifier - is defined by the application. With OSCOAP the Context Identifier typically identifies the sending party and different resources may be identified by the Uri-Path in the request. (Compare Appendix C.)

The ordered sequence (SEQ, CID) is called Transaction Identifier (TID), and SHALL be unique for each SM.
4.2. Secure Message Body and Tag

The use cases require support for two message types, one for Encryption and Integrity Protection, and another for integrity protection only. The SM Body and the SM Tag are different depending on message type.

For Integrity Protection Only we denote by Authenticated Data (AD) the data which is integrity protected in the Secure Message. For Encryption and Integrity Protection we denote by Plaintext and Additional Authenticated Data (AAD), the data which is encrypted and integrity protected, and integrity protected only, respectively, in the Secure Message.

The message type SHALL be explicit to allow an intermediate node to distinguish between the two types and read the SM Body of an Integrity Protected Only message.

4.2.1. Integrity Protection Only

In the case of integrity protection only, the SM Body SHALL consist of the payload of the CoAP message.

The SM Tag SHALL consist of the Signature / Message Authentication Code (MAC) as defined by the cipher suite calculated over the Authenticated Data (AD). The AD for OSCOAP is defined in Section 5.1.2.

4.2.2. Encryption and Integrity Protection

The use cases require support for two kinds of cipher suites: Authenticated Encryption with Additional Data (AEAD) as well as Symmetric Encryption and Asymmetric Signature (SEAS).

In case of AEAD, the SM Body and SM Tag SHALL consist of the Ciphertext as defined by the cipher suite calculated over the Plaintext and the Additional Authenticated Data (AAD).

In case of SEAS, the SM Body SHALL be the Ciphertext as defined by the symmetric encryption algorithm, given by the cipher suite, calculated over the Plaintext. The SM Tag SHALL be the Signature defined by the cipher suite calculated over Ciphertext and AAD.

The Plaintext and the AAD for OSCOAP are defined in Section 5.2.2.
5. CoAP Message Protection

This section presents how OSCOAP protects individual CoAP messages including payload, options and header fields, as well as request-response message exchanges, using the Object-Security option (Section 3) and the Secure Message format (Section 4).

The basic idea is that the significant parts of an unprotected CoAP message - including payload, certain header field and options - are protected using the Secure Message format and sent in a CoAP message with the Object-Security option, in what we then call a "protected" CoAP message. As much as possible of the CoAP message should be protected, but not all CoAP header fields or options can be encrypted and integrity protected, because some are intended to be read or changed by an intermediary node, see Section 6.1 and Section 6.2.

The use of OSCOAP is signaled with the Object-Security option. Endpoints supporting the Object-Security option MUST verify the SM as described in this section before accepting a message as valid. An endpoint receiving a CoAP request with the Object-Security option MUST respond with a CoAP message with the Object-Security option.

The differences between Encryption and Integrity Protection vs Integrity Protection Only is described below. Encryption and Integrity Protection SHALL be used by default.

5.1. Integrity Protection Only

5.1.1. Protected CoAP message formatting

The protected CoAP message is formatted as an ordinary CoAP message, with the following Header, Options and Payload based on the unprotected CoAP message:

- The CoAP header SHALL be the same as the unprotected CoAP message.
- The CoAP options SHALL consist of the same options as the unprotected CoAP message, and the Object-Security option.
- If the unprotected CoAP message has no Payload then the Object-Security option SHALL contain the SM. If the unprotected CoAP message has Payload, then the Object-Security option SHALL be empty and the Payload of the CoAP message SHALL be the SM.
5.1.2. Secure Message formatting

The SM Header, Body and Tag are specified in Section 4.1 and Section 4.2.

The Authenticated Data SHALL consist of the following data, in this order:

- the SM Header;

- the two first bytes of the CoAP header (including Version and Code) with Type and Token Length bits set to 0;

- all CoAP options present which are marked as IP in Figure 2 (Section 6.2), in the order as given by the option number (each Option with Option Header including delta to previous IP-marked Option which is present);

- the CoAP Payload (if any); and

- the Transaction Identifier of the associated CoAP Request, if the message is a CoAP Response (see Section 4.1).

5.1.3. Integrity Protection and Verification

A CoAP endpoint protecting a CoAP message with the Object-Security option using a cipher suite for integrity protection only SHALL generate a protected CoAP message and SM based on the unprotected CoAP message as described in Section 5.1.1 and Section 5.1.2. In addition, the sending endpoint SHALL process the Sequence Number as described in Section 7.

A CoAP endpoint receiving a message containing the Object-Security option SHALL first recreate the Authenticated Data as described in Section 5.1.2, and then verify the SM Tag as defined by the cipher suite associated to the Context Identifier. In addition, the receiving endpoint SHALL process the Sequence Number as described in Section 7.

5.2. Encryption and Integrity Protection

5.2.1. Protected CoAP message formatting

The protected CoAP message is formatted as an ordinary CoAP message, with the following Header, Options and Payload based on the unprotected CoAP message:

- The CoAP header SHALL be the same as the unprotected CoAP message.
The CoAP options SHALL consist of the unencrypted options of the unprotected CoAP message (those not marked as E in Figure 2 (Section 6.2)), and the Object-Security option. The options shall be formatted as in a CoAP message (each Option with Options Header including delta to previous unencrypted Option).

If the unprotected CoAP message has no Payload then the Object-Security option SHALL contain the SM. If the unprotected CoAP message has Payload, then the Object-Security option SHALL be empty and the Payload of the CoAP message SHALL be the SM.

5.2.2. Secure Message formatting

The SM Header, Body and Tag are specified in Section 4.1 and Section 4.2.

The Additional Authenticated Data SHALL consist of the following data, in this order:

- the SM Header;
- the two first bytes of the CoAP header (including Version and Code) with Type and Token Length bits set to 0;
- all CoAP options present which are marked as IP but not marked as E in Figure 2 (Section 6.2), in the order as given by the option number (each Option with Option Header including delta to previous IP-marked Option which is present); and
- the Transaction Identifier of the associated CoAP Request, if the message is a CoAP Response (see Section 4.1).

The Plaintext SHALL consist of the following data, formatted as a CoAP message without Header consisting of:

- all CoAP Options present which are marked as E in Figure 2 (see Section 6.2), in the order as given by the Option number (each Option with Option Header including delta to previous E-marked Option); and
- the CoAP Payload, if present, and in that case prefixed by the one-byte Payload Marker (0xFF).

5.2.2.1. Encryption and Decryption

A CoAP endpoint protecting a CoAP message with the Object-Security option using a cipher suite for encryption and integrity protection SHALL generate a protected CoAP message and SM based on the
unprotected CoAP message as described in Section 5.2.1 and Section 5.2.2. In addition, the sending endpoint SHALL process the Sequence Number as described in Section 7.

A CoAP endpoint receiving a message containing the Object-Security option SHALL recreate the Additional Authenticated Data as described in Section 5.1.2 and verify the integrity of, and decrypt the message as defined by the cipher suite associated to the Context Identifier. In addition, the receiving endpoint SHALL process the Sequence Number as described in Section 7.

6. Protected CoAP Message Fields

The CoAP payload SHALL be integrity protected. The CoAP payload SHOULD be encrypted by default.

How CoAP Options and Header Fields shall be protected is described in the remainder of this section.

6.1. Protected CoAP Header Fields

This section describes which CoAP header fields are encrypted or integrity protected end-to-end in OSCOAP.

The CoAP Message Layer parameters, Type and Message ID, as well as Token and Token Length may be changed by a proxy and thus SHALL neither be integrity protected nor encrypted.

The Version and Code fields SHALL be integrity protected, see security considerations.

6.2. Protected CoAP Options

This section describes which CoAP options are encrypted and integrity protected, if present in the unprotected CoAP message.

All CoAP options SHALL be encrypted by default, unless intended to be read by an intermediate node; and SHALL be integrity protected, unless intended to be changed by an intermediate node.

However, some special considerations are necessary because CoAP defines certain legitimate proxy operations, because the security information itself may be transported as an option, and because different processing is performed depending on whether encryption is applied or not.

The details are presented in Section 6.2.1 and Section 6.2.2, and summarized in Figure 2.
<table>
<thead>
<tr>
<th>No.</th>
<th>C</th>
<th>U</th>
<th>N</th>
<th>R</th>
<th>Name</th>
<th>Format</th>
<th>Length</th>
<th>E</th>
<th>IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>If-Match</td>
<td>opaque</td>
<td>0-8</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td></td>
<td>Uri-Host</td>
<td>string</td>
<td>1-255</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>ETag</td>
<td>opaque</td>
<td>1-8</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>If-None-Match</td>
<td>empty</td>
<td>0</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>6</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td></td>
<td>Observe</td>
<td>uint</td>
<td>0-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>Uri-Port</td>
<td>uint</td>
<td>0-2</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>x</td>
<td>Location-Path</td>
<td>string</td>
<td>0-255</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>Uri-Path</td>
<td>string</td>
<td>0-255</td>
<td>x</td>
<td>b</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>Content-Format</td>
<td>uint</td>
<td>0-2</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>Max-Age</td>
<td>uint</td>
<td>0-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>Uri-Query</td>
<td>string</td>
<td>0-255</td>
<td>x</td>
<td>b</td>
</tr>
<tr>
<td>17</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>Accept</td>
<td>uint</td>
<td>0-2</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>x</td>
<td></td>
<td>Location-Query</td>
<td>string</td>
<td>0-255</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>Block2</td>
<td>uint</td>
<td>0-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>Block1</td>
<td>uint</td>
<td>0-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>x</td>
<td></td>
<td>Size2</td>
<td>uint</td>
<td>0-4</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>Proxy-Uri</td>
<td>string</td>
<td>1-1034</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>Proxy-Scheme</td>
<td>string</td>
<td>1-255</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>x</td>
<td></td>
<td>Size1</td>
<td>uint</td>
<td>0-4</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* C=Critical, U=Unsafe, N=NoCacheKey, R=Repeatable, E=Encrypt, IP=Integrity Protect.

Figure 2: Protected CoAP options in OSCOAP

CoAP options marked "i" indicate that they are used as invariants in the authenticated data (AD/AAD) as described in Section 6.2.1.1 and Section 6.2.1.2.

In case of Integrity Protection Only, options marked with "a" and "b" are composed into a URI as described in Section 6.2.1.2 and included as invariant in the Proxy-Uri option in the Authenticated Data.

In case of Encryption and Integrity Protection, options marked "a" are composed into a URI as described in Section 6.2.2 and included as the Proxy-Uri option in the Additional Authenticated Data. (Options marked "b" are included in the Plaintext.)

6.2.1. Integrity Protection

CoAP options which are not intended to be changed by an intermediate node MUST be integrity protected.

- CoAP options of the unprotected message which are Safe-to-Forward SHALL be integrity protected. See Figure 2.
Note: The Object-Security option in itself is Safe-to-Forward but is added to the protected message.

CoAP options which are intended to be modified by a proxy can be divided into two categories, those that are intended to change in a predictable way, and those which are not. The following options are of the latter kind and SHALL NOT be integrity protected:

- Max-Age, Observe, Block1, Block2: These options may be modified by a proxy in a way that is not predictable for client and server.

The remaining options may be modified by a proxy, but when they are, the change is predictable. Therefore it is possible to define "invariants" which can be integrity protected.

6.2.1.1. Proxy-Scheme

A Forward Proxy is intended to replace the URI scheme with the content of the Proxy-Scheme option. The Proxy-Scheme option is defined in this memo to be an invariant with respect to the following processing:

- If there is a Proxy-Scheme present in the unprotected message, then the client SHALL integrity protect the Proxy-Scheme option.

- If there is no Proxy-Scheme option present the client SHALL include the Proxy-Scheme option in the authenticated data (AD/AAD) set to the URI scheme. (The sent message does not include the Proxy-Scheme option.)

- The server SHALL insert the Proxy-Scheme option with the name of the URI scheme the message was received in the authenticated data (AD/AAD).

6.2.1.2. Uri- *

For options related to URI of resource (Uri-Host, Uri-Port, Uri-Path, Uri-Query, Proxy-Uri) a Forward Proxy is intended to replace the Uri-* options with the content of the Proxy-Uri option.

The Proxy-Uri option is defined in this memo to be an invariant with respect to the following processing (applied to Integrity Protection only, for Encryption see next section):

- If there is a Proxy-Uri present, then the client MUST integrity protect the Proxy-Uri option and the Uri-* options MUST NOT be integrity protected.
If there is no Proxy-Uri option present, then the client SHALL compose the full URI from Uri-* options according to the method described in section 6.5 of [RFC7252]. The Authenticated Data contains the following options, modified compared to what is sent:

- All Uri-* options removed
- A Proxy-Uri option with the full URI included

The server SHALL compose the URI from the Uri-* options according to the method described in section 6.5 of [RFC7252]. The so obtained URI is placed into a Proxy-Uri option, which is included in the Authenticated Data.

### 6.2.2. Encryption

All CoAP options MUST be encrypted, except the options below which MUST NOT be encrypted:

- Max-Age, Observe, Block1, Block2, Proxy-Uri, Proxy-Scheme: This information is intended to be read by a proxy.
- Uri-Host, Uri-Port: This information can be inferred from destination IP address and port.
- Object-Security: This is the security-providing option.

In the case of encryption, the Proxy-Uri of the Additional Authenticated Data MUST only contain Uri-Host and Uri-Port and MUST NOT contain Uri-Path and Uri-Query because the latter options are not necessarily available to a Forward Proxy.

### 7. Replay Protection and Freshness

In order to protect from replay of messages and verify freshness of responses, a CoAP endpoint using object security SHALL maintain Sequence Numbers (SEQs) of sent and received Secure Messages (see Section 4.1), associated to the respective security context identified with the Context Identifier (CID).

#### 7.1. Replay Protection

An endpoint SHALL maintain a SEQ for each security context it uses to receive messages, and one SEQ for each security context for protecting sent messages. Depending on use case, an endpoint MAY maintain a sliding receive window for Sequence Numbers in received messages associated to each CID, equivalent to the functionality described in section 4.1.2.6 of [RFC6347].
Before composing a new message a sending endpoint SHALL step the SEQ of the associated CID. However, if the Sequence Number counter wraps, the endpoint must first acquire a new CID and associated security context/key(s). The latter is out of scope of this memo.

A receiving endpoint SHALL verify that the Sequence Number received in the SM Header is greater than the Sequence Number of the associated CID (or within the sliding window and not previously received) and update the SEQ (window) accordingly.

7.2. Freshness

OSCOAP is a challenge-response protocol, where the response is verified to match a prior request by including the unique transaction identifier TID (concatenation of SEQ and CID) of the request in the integrity calculation of the response message.

If a CoAP server receives a request with the Object-Security option, then the authenticated data (AD or AAD) of the response SHALL include the TID of the request as described in Section 5.1.2 and Section 5.2.2.

If the CoAP client receives a response with the Object-Security option, then the client SHALL verify the integrity of the response using the TID of its own associated request in the authenticated data (AD or AAD) as described in Section 5.1.2 and Section 5.2.2.

8. Security Considerations

In scenarios with proxies, gateways, or caching, DTLS only protects data hop-by-hop meaning that these intermediary nodes can read and modify information. The trust model where all participating nodes are considered trustworthy is problematic not only from a privacy perspective but also from a security perspective as the intermediaries are free to delete resources on sensors and falsify commands to actuators (such as "unlock door", "start fire alarm", "raise bridge"). Even in the rare cases where all the owners of the intermediary nodes are fully trusted, attacks and data breaches make such an architecture weak.

DTLS protects the entire CoAP message including Header, Options and Payload, whereas OSCOAP protects the payload and message fields described in Section 6.1 and Section 6.2. The cost for DTLS providing this protection is the overhead in e.g. additional messages, processing, memory incurred by the DTLS Handshake protocol, which can be omitted in use cases where key establishment can be provided by other means.
CoAP specifies how messages should be acknowledged on message layer. The CoAP message layer, however, cannot be protected by application layer security end-to-end since the parameters Type and Message ID, as well as Token and Token Length may be changed by a proxy. Moreover, messages that are not possible to verify should for security reasons not always be acknowledged but in some cases be silently dropped. This would not comply with CoAP message layer, but does not have an impact on the object security solution, since message layer is excluded from that.

The CoAP Header field Code needs to be integrity protected end-to-end. For example, if a malicious man-in-the-middle would replace the client requested GET with a DELETE, this must be detected by the server. The CoAP Header field Version needs also to be integrity protected to prevent from potential cross-version attacks, such as bidding-down.

Blockwise transfers as defined [I-D.ietf-core-block] cannot be protected with application layer security end-to-end because the Block1/Block2 options may be changed in an unpredictable way by an intermediate node.

However, it is possible to define end-to-end block options analogous to Block1 and Block2 which are safe-to-forward, integrity protected and not supposed to be changed by intermediate devices. With such an option each individual block can be securely verified by the receiver, retransmission securely requested etc. Since the blocks are enumerated sequentially and carry information about last block, when all blocks have been securely received, this proves that the entire message has been securely transferred.

The Observe option cannot be integrity protected since it is allowed to change in an unpredictable way. But since message sequence numbers are integrity protected a client can verifies that a GET response has not been received before.

The use of sequence numbers for replay protection introduces the problem related to wrapping of the counter. The alternatives also have issues: very constrained devices may not be able to support accurate time or generate and store large numbers of random nonces. The requirement to change key at counter wrap is a complication, but it also forces the user of this specification to think about implementing key renewal.

This specification needs to be complemented with a procedure whereby the client and the server establish the keys used for wrapping and unwrapping the Secure Message. One way to address key establishment is to assume that there is a trusted third party which can support
client and server, such as the Authorization Server in [I-D.ietf-ace-actors]. The Authorization Server may, for example, authenticate the client on behalf of the server, or provide cryptographic keys or credentials to the client and/or server which can be use to derive the keys used in the Secure Message exchange. Similarly, the Authorization Server may, on behalf of the server, notify the client of server supported ciphers, in order to facilitate the usage of OSCOAP in deployments with multiple supported cryptographic algorithms.

The security contexts required are different for different cipher suites. For an AEAD or SEAS it is required to have a unique Initialization Vector for each message, for which the Sequence Number is used. The Initialization Vector SHALL be the concatenation of a Salt (4 bytes unsigned integer) and the Sequence Number. The Salt SHOULD be established between sender and receiver before the message is sent, to avoid the overhead of sending it in each message. For example, the Salt may be established by the same means as keys are established.

9. Privacy Considerations

End-to-end integrity protection provides certain privacy properties, e.g. protection of communication with sensor and actuator from manipulation which may affect the personal sphere. End-to-end encryption of payload and certain CoAP options provides additional protection as to the content and nature of the message exchange.

The headers sent in plaintext allow for example matching of CON and ACK (CoAP Message Identifier), matching of request and response (Token). Plaintext options could also reveal information, e.g. lifetime of measurement (Max-age), or that this message contains one data point in a sequence (Observe).

10. IANA Considerations

Note to RFC Editor: Please replace all occurrences of "[this document]" with the RFC number of this specification.

The following entry is added to the CoAP Option Numbers registry:

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBD</td>
<td>Object-Security</td>
<td>[[this document]]</td>
</tr>
</tbody>
</table>

This document registers the following value in the CoAP Content Format registry established by [RFC7252].

Media Type: application/oscon

Encoding: -

Id: 70

Reference: [this document]

11. Acknowledgments

Klaus Hartke has independently been working on the same problem and a similar solution: establishing end-to-end security across proxies by adding a CoAP option. We are grateful to Malisa Vucinic for providing helpful and timely reviews of new versions of the draft.

12. References

12.1. Normative References


12.2. Informative References

[I-D.ietf-ace-actors]

[I-D.ietf-ace-usecases]

[I-D.ietf-core-block]

[I-D.ietf-cose-msg]

[I-D.seitz-ace-core-authz]


Appendix A. COSE Profile of SM

This section defines a profile of the 05-version of COSE [I-D.ietf-cose-msg] complying with the Secure Message format (see Section 4) and supporting the two scopes of object security OSCOAP and OSCON (Appendix C). In the last subsection we elaborate on possible optimizations.

- The "COSE_MSG" top level object as defined in COSE corresponds to the Secure Message object.
- The "msg_type" parameter corresponds to the Secure Message type, as defined in Section 4.2. Depending on the use case, this field can take the values msg_type_mac, msg_type_signed or msg_type_encryptData.
- The "Header" field of the COSE object corresponds to the Header field of the Secure Message.
  * The "protected" field includes:
    + the new "seq" parameter corresponding to the parameter Sequence Number of the Secure Message (see Section 4.1).
  * The "unprotected" field is empty.

A.1. Integrity Protection Only

When Integrity Protection only needs to be provided, the Secure Message object corresponds to a COSE_MSG with msg_type equal to msg_type_signed (COSE_Sign) or msg_type_mac (COSE_mac).

The Externally Supplied Data ("external_aad" field), as defined in Section 4.1 of [I-D.ietf-cose-msg] include the Authenticated Data as defined in Section 5.1.2 with the exception of SM Header and CoAP Payload.

A.1.1. COSE_Sign

A COSE_MSG of type COSE_Sign is a Secure Message if its fields are defined as follows (see example in Appendix B.2).

- The "Headers" field of COSE_Sign as defined in Appendix A.
- The "payload" field contains the CoAP Payload (if any).
- The "signatures" array contains one "COSE_signature" item. The "Headers" field of the COSE_signature object is defined as follows:
The "protected" field includes:

* the new "cid" parameter which corresponds to the parameter Context Identifier of the Secure Message (see Section 4.1);

The "unprotected" field is empty.

The "signature" field contains the computed signature value as described in Section 4.2 of [I-D.ietf-cose-msg].

A Secure Message with digital signature and Detached Content corresponds to COSE_sign with "Headers" and "signatures" fields; i.e. no "payload" field.

A.1.2. COSE_mac

A COSE_MSG of type COSE_mac is a Secure Message if its fields are defined as follows (see example in Appendix B.1).

The "Headers" field of COSE_mac as defined in Appendix A.

The "payload" field contains the CoAP Payload (if any).

The "tag" field contains the MAC value, computed as defined in Section 6.1 of [I-D.ietf-cose-msg].

The "recipients" array contains one "COSE_recipient" item (section 5 of [I-D.ietf-cose-msg]). The "COSE_recipient" item contains one "COSE_encrypt_fields" object. The "Headers" field of the COSE_encrypt_fields object is defined as follows:

* the new "cid" parameter which corresponds to the parameter Context Identifier of the Secure Message (see Section 4.1);

The "unprotected" field is empty.

A Secure Message with MAC and Detached Content corresponds to a COSE_sign with "Headers", "recipients" and "tag" fields; i.e. no "payload" field.

A.2. Encryption and Integrity Protection: COSE_enveloped

When Encryption and Integrity Protection need to be provided, the Secure Message object corresponds to a COSE_MSG with msg_type equal to msg_type_enveloped (COSE_enveloped).
The Additional Authenticated Data ("Enc_structure") as described is Section 5.3 of [I-D.ietf-cose-msg] is defined in Section 5.2.2: * the "protected" parameters includes the SM Header; * the "external_aad" includes the other fields (CoAP Version, Code, Options to integrity protect and TID).

The plain text, as mentioned in Sections 5.3 and 5.4 of [I-D.ietf-cose-msg] is defined in Section 5.2.2 and contains CoAP Options to encrypt and the CoAP Payload.

A COSE_MSG of type COSE_enveloped [I-D.ietf-cose-msg] is a Secure Message if its fields are defined as follows (see example in Appendix B.3).

The "Headers" field of COSE_encrypt_fields item as defined in Appendix A.

The "ciphertext" field is encoded as a nil type, following the specifications in Section 5.1 of [I-D.ietf-cose-msg].

The "recipients" array contains one "COSE_recipient" item (Section 5.1 of [I-D.ietf-cose-msg]). The "COSE_recipient" item contains one "COSE_encrypt_fields" object. The "Headers" field of the COSE_encrypt_fields object is defined as follows:

- The "protected" field includes:
  - the new "cid" parameter which corresponds to the parameter Context Identifier of the Secure Message (see Section 4.1);
- The "unprotected" field is empty.

The "ciphertext" field of the COSE_encrypt_fields object contains the encrypted plain text, as defined in section 5 of [I-D.ietf-cose-msg].

A.3. COSE Optimizations

For constrained environments it is important that the message expansion due to security overhead is kept at a minimum.

This section lists potential optimizations of COSE [I-D.ietf-cose-msg] for the purpose of reducing message size and improving performance in constrained node networks. The message sizes resulting from the first four optimizations are presented in Appendix B (as "modified COSE").

1. The first improvement proposed is to flatten the structure of the COSE_msg, following the Encrypted COSE structure defined in
Section 5.2 of [I-D.ietf-cose-msg]. In fact, there is little need to support multiple signatures or recipients in the use cases targeting the most constrained devices. Two different structures inspired by the COSE_encryptData are defined: COSE_ip and COSE_en. COSE_ip is used for the Integrity Protection Only use case (Section 5.1), COSE_en is used for Encryption (Section 5.2).

2. In general, the security context defines uniquely the cipher suite, and hence the "alg" parameter of COSE_msg can be removed.

3. The "unprotected" field is not used since it is assumed that all parameters should be protected when possible. Thus the "Headers" structure can be flattened into a "protectedHeader" field, containing the "cid" parameter and the "seq" parameter.

4. Analogous to other key values, one-byte keys/labels can be assigned to the new parameters defined in this document and cipher suites adapted to constrained device processing. For example: "cid" = 11 and "seq" = 12.

5. Digitally signed messages have the largest absolute overhead due to the size of the signature (see Appendix B.2 and Appendix B.4). Whereas certain MACs can be securely truncated, signatures cannot. Signature schemes with message recovery allow some remedy since they allow part of the message to be recovered from the signature itself and thus need not be sent. The effective size of the signature could in this way be considerably reduced, which would have a large impact on the message size (compare size of signature and total overhead in Figure 5 and Figure 6). A valuable optimization would thus be to support signature schemes with message recovery.

Combining the first 4 points, the resulting structures and their fields are defined as follows: COSE_ip top level object corresponds to the Secure Message object.

- The "msg_type" parameter takes a new value, msg_type_integrityprotection=5.

- The "protectedHeader" field, analogous to the "protected" field of the "Headers", includes:
  * the new "cid" parameter which corresponds to the parameter Context Identifier of the Secure Message (see Section 4.1);
  * the new "seq" parameter corresponding to the parameter Sequence Number of the Secure Message (see Section 4.1).
- The "payload" field (as described in Appendix A.1.1 and Appendix A.1.2).
- The "tag" field (as described in Appendix A.1.1 and Appendix A.1.2).

COSE_en top level object corresponds to the Secure Message object.

- The "msg_type" parameter takes a new value, msg_type_encryption=6.
- The "protectedHeader" field, analogous to the "protected" field of the "Headers", includes:
  * the new "cid" parameter which corresponds to the parameter Context Identifier of the Secure Message (see Section 4.1);
  * the new "seq" parameter corresponding to the parameter Sequence Number of the Secure Message (see Section 4.1).
- The "ciphertext" field (as described in Appendix A.2).
- The "tag" field contains the tag value in case Integrity Protection is also provided.

Appendix B. Comparison of message sizes

This section gives some examples of overhead incurred with the current proposal for COSE at the time of writing [I-D.ietf-cose-msg]. Message sizes are also listed for a modified version of COSE implementing some of the optimizations described in Appendix A.3 and for a lower bound CBOR encoding of the Secure Message with structure [seq, cid, body, tag].

Motivated by the use cases, there are four different kinds of protected messages that need to be supported: message authentication code, digital signature, authenticated encryption, and symmetric encryption + digital signature. The latter is relevant e.g. for proxy-caching and publish-subscribe with untrusted intermediary (see Appendix D.2). The sizes estimated for selected algorithms are detailed in the subsections.

The size of the header is shown separately from the size of the MAC/signature. An 8-byte Context Identifier and a 3-byte Sequence Number are used throughout all examples, with these value:

- cid: 0xa1534e3c5fdc09bd
- seq: 0x112233

For each scheme, we indicate the fixed length of these two parameters ("seq+cid" column) and of the tag ("MAC"/"SIG"/"TAG"). The "Total Size" column shows the total Secure Message size, while the "Overhead" column is calculated from the previous columns following this equation:

\[
\text{Overhead} = \text{Total Size} - (\text{MAC} + \text{seq+cid})
\]

This means that overhead incurring from CBOR encoding is also included in the Overhead count.

To make it easier to read, COSE objects are represented using CBOR’s diagnostic notation rather than a binary dump.

B.1. MAC Only

This example is based on HMAC-SHA256, with truncation to 16 bytes.

The object in COSE encoding gives:

```
[ 3,            # msg_type
  h’a201046373657143112233’,  # protected:
    {1: 4,  
      "seq": h’112233’
    }
  ],               # unprotected
  h’’, # payload
  MAC,           # truncated 16-byte MAC
  [               # recipients
    [                      # recipient structure
      h’’,                 # protected
        {1:-6, "cid":h’a1534e3c5f6c09bd’}, # unprotected
        h’’                 # ciphertext
    ]
  ]
]
```

The COSE object encodes to a total size of 53 bytes.

In the modified version of COSE defined in Appendix A.3, the equivalent COSE object would be:
This modified COSE object encodes to a total size of 37 bytes.

The low-bound CBOR encoding of this same object is encoded by:

```plaintext
[ 
  h'112233',   # seq
  h'a1534e3c5f0d09bd', # cid
  h'',        # payload
  MAC         # truncated 16-byte MAC
]
```

This object encodes to a total size of 32 bytes.

Figure 3 summarizes these results.

```
+--------+---------+------+------------+----------+
| Scheme | seq+cid |  MAC | Total Size | Overhead |
|--------+---------+------+------------+----------|
| COSE   |   11 B  |  16 B |  53 bytes  | 26 bytes |
| mod-COSE|   11 B  |  16 B |  37 bytes  | 10 bytes |
| bound  |   11 B  |  16 B |  32 bytes  |  5 bytes |
```

**Figure 3**: Comparison of COSE, modified COSE and CBOR lower bound for HMAC-SHA256.

### B.2. Signature Only

This example is based on ECDSA, with a signature of 64 bytes.

The object in COSE encoding gives:

```plaintext
[ 
  5,       # msg_type
  h'a20b48a1534e3c5f0d0c43112233', # protected:
    11:h'a1534e3c5f0d09bd',
    12:h'112233' )
  h'',   # payload
  MAC    # truncated 16-byte MAC
]
```
The COSE object encodes to a total size of 100 bytes.

In the modified version of COSE defined in Appendix A.3, the equivalent COSE object would be:

```
[  
  5, # msg_type
  h’a20b48a1534e3c5fdc09bd0c43112233’, # protected:
    {11:h’a1534e3c5fdc09bd’,
     12:h’112233’}
  h’’, # payload
  SIG
] # 64-byte signature
```

The COSE object encodes to a total size of 86 bytes.

The low-bound CBOR encoding of this same object is encoded by:

```
[  
  h’112233’, # seq
  h’a1534e3c5fdc09bd’, # cid
  h’’, # payload
  SIG # 64-byte signature
] # 64-byte signature
```

This object encodes to a total size of 81 bytes.

Figure 4 summarizes these results.
<table>
<thead>
<tr>
<th>Scheme</th>
<th>seq+cid</th>
<th>SIG</th>
<th>Total Size</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>COSE</td>
<td>11 B</td>
<td>64 B</td>
<td>100 bytes</td>
<td>25 bytes</td>
</tr>
<tr>
<td>mod-COSE</td>
<td>11 B</td>
<td>64 B</td>
<td>86 bytes</td>
<td>11 bytes</td>
</tr>
<tr>
<td>bound</td>
<td>11 B</td>
<td>64 B</td>
<td>81 bytes</td>
<td>6 bytes</td>
</tr>
</tbody>
</table>

Figure 4: Comparison of COSE, modified COSE and CBOR lower bound for 64 byte ECDSA signature.

B.3. Authenticated Encryption with Additional Data (AEAD)

This example is based on AES-128-CCM-8.

It is assumed that the IV is generated from the Sequence Number and some previously agreed upon Salt. This means it is not required to explicitly send the whole IV in the message.

The object in COSE encoding gives:

```
[  2,  # msg_type
  h’a201046373657143112233’,  # protected:
    {1: 4,  # 8byte authentication tag
      "seq": h’112233’}
  ],  # unprotected
  TAG,  # 8byte authentication tag
  [  # recipients
    [  # recipient structure
      h’’,  # protected
        {1:-6, "cid":h’a1534e3c5f09b9’},  # unprotected
        h’’  # ciphertext
    ]
  ]
```

The COSE object encodes to a total size of 44 bytes.

In the modified version of COSE defined in Appendix A.3, the equivalent COSE object would be:
The modified COSE object encodes to a total size of 29 bytes.

The low-bound CBOR encoding of this same object is encoded by:

```json
[
  h'112233', # seq
  h'a1534e3c5fbd09bd', # cid
  h'', # ciphertext
  TAG # 8byte authentication tag
]
```

This object encodes to a total size of 24 bytes.

Figure 5 summarizes these results.

```
+--------+---------+-----+------------+----------+
| Scheme | seq+cid | TAG | Total Size | Overhead |
+--------+---------+-----+------------+----------+
| COSE   | 11 B    | 8 B | 44 bytes   | 25 bytes |
+--------+---------+-----+------------+----------+
| mod-COSE | 11 B    | 8 B | 29 bytes   | 10 bytes |
+--------+---------+-----+------------+----------+
| bound  | 11 B    | 8 B | 24 bytes   | 5 bytes  |
+--------+---------+-----+------------+----------+
```

Figure 5: Comparison of COSE, modified COSE and CBOR lower bound for AES-CCM.

B.4. Symmetric Encryption with Asymmetric Signature (SEAS)

This example is based on AES-128-CTR and ECDSA with 64 bytes signature. COSE requires this to be a nested encapsulation of one object into another, here illustrated with a digitally signed AEAD protected object.

The object in COSE encoding gives:
The COSE object encodes to a total size of 134 bytes.

In the modified version of COSE defined in Appendix A.3, the equivalent COSE object would be:

```
[ 6,  # msg_type
  h’a20b48a1534e3c5f0dc09bd0c43112233’,  # protected:
    {11: h’a1534e3c5f0dc09bd’,
     12: h’112233’})
  h’,  # ciphertext
  SIG
]  # 64-byte signature
```

This modified COSE object encodes to a total size of 86 bytes.

The low-bound CBOR encoding of this same object is encoded by:

```
[ h’112233’,  # seq
  h’a1534e3c5f0dc09bd’,  # cid
  h’,  # ciphertext
  SIG  # 64-byte signature
]
This object encodes to a total size of 81 bytes.

Figure 6 summarizes these results.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>seq+cid</th>
<th>SIG</th>
<th>Total Size</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>COSE</td>
<td>11 B</td>
<td>64 B</td>
<td>134 bytes</td>
<td>59 bytes</td>
</tr>
<tr>
<td>mod-COSE</td>
<td>11 B</td>
<td>64 B</td>
<td>86 bytes</td>
<td>11 bytes</td>
</tr>
<tr>
<td>bound</td>
<td>11 B</td>
<td>64 B</td>
<td>81 bytes</td>
<td>6 bytes</td>
</tr>
</tbody>
</table>

Figure 6: Comparison of nested AES-CCM within ECDSA (COSE) and combined AES-ECDSA (modified COSE and CBOR lower bound).

Appendix C. Object Security of Content (OSCON)

In this section we define how to only protect the payload/content of individual messages using the Secure Message format (Section 4) to comply with the requirements 1 and 2 in Section 2. This is referred to as Object Security of Content (OSCON).

Note that by only protecting the content of a message it may be verified by multiple recipients. For example, in the case of a proxy that supports caching, a recent response for a certain resource can be cached and used to serve multiple clients. Or, in a publish-subscribe setting, multiple subscribers can be served the same publication. The use of content protection also decouples the binding to the underlying transfer protocol, so the same protected content object can be freely move between CoAP, HTTP, BlueTooth or whatever application layer protocol.

The use of OSCON is signaled with the Content-Format/Media Type set to application/oscon (Section 10). Since the actual format of the content which is protected is lost, that information needs to be added to the message header or known to the recipient.

The sending endpoint SHALL wrap the Payload, and the receiving endpoint unwrap the Payload in the SM format as described in this section. A CoAP client MAY request a response in the OSCON format by setting the option Accept to application/oscon.

In case of cipher suite for integrity protection only, the Authenticated Data SHALL be the concatenation of the SM Header and the CoAP Payload. If case of cipher suite for both encryption and integrity protection, then the AAD SHALL be the SM Header and the
Plaintext SHALL be the CoAP Payload. By default, cipher suites for encryption and integrity protection SHALL be used.

The SM SHALL be protected (encrypted) and verified (decrypted) as described in Section 5.1.3 (Section 5.2.2.1), including replay protection as described in Section 7.1.

Whereas in OSCOAP, the Context Identifier of the SM Header (Section 4.1) typically identifies the sending party, with OSCON (Appendix C) the Context Identifier may well identify the sender and resource.

C.1. Security Considerations of OSCON

OSCON (Appendix C) only protects payload and only gives replay protection (not freshness of response), but allows additional use cases such as point to multi-point interactions including publish-subscribe, reverse proxies and proxy caching of responses. In case of symmetric keys the receiver does not get data origin authentication, which requires a digital signature using a private asymmetric key.

OSCON SHALL NOT be used in cases where CoAP header fields (such as Code or Version) or CoAP options need to be integrity protected. The request for a response in OSCON using the CoAP option Accept set to "application/oscon" is not secured since OSCON does not integrity protect any options. Hence the exchange of OSCON request-response messages is vulnerable to a man-in-the-middle attack where response is exchanged for another response, but since there is replay protection only messages with higher sequence numbers will be accepted.

Blockwise transfers in CoAP as defined in [I-D.ietf-core-block] can be applied with OSCON, i.e. the entire payload is encapsulated in a Secure Message which is partitioned into blocks which are sent with unprotected CoAP. The receiver is able to verify the integrity of the payload but only after the last block containing the signature/MAC is received, and if the verification fails the entire message needs to be resent. However, if the verification succeeds, then the transmission in OSCON has less computational and packet overhead since only one signature/MAC was generated and sent. As CoAP blockwise transfer with OSCON is prone to Denial of Service attacks, it should only be used for exchanges where this threat can be mitigated, for example within a local area network where link-layer security is activated.
Appendix D.  Examples

This section gives examples of how to use the Object-Security option and the message formats defined in this memo.

D.1.  CoAP Message Protection

This section illustrates Object Security of CoAP (OSCOAP). The message exchange assumes there is a security context established between client and server. One key is used for each direction of the message transfer. The intermediate node detects that the CoAP message contains an OSCOAP object (Object-Security option is set) and thus forwards the message as it cannot serve a cached response.

D.1.1.  Integrity Protection of CoAP Message Exchange

Here is an example of a PUT request/response message exchange passing an intermediate node protected with the Object-Security option. The example illustrates a client closing a lock (PUT 1) and getting a confirmation that the lock is closed. Code, Uri-Path and Payload of the request and Code of the response are integrity protected (and other message fields, see Section 6.1 and Section 6.2).
Figure 7: CoAP PUT protected with OSCOAP

Since the request message (PUT) supports payload, the OSCOAP object is carried in the CoAP payload. Since the response message (Changed) does not support payload the Object-Security option carries the OSCOAP object.

The Header contains Sequence Number ("seq":"a6") and Context Identifier ("cid":"5fda09bda15343c"), the latter is an identifier indicating which security context was used to integrity protect the message, and may be used as an identifier for a secret key or a public key. (It may e.g. be the hash of a public key.)

The server and client can verify that the Sequence Number has not been received and used with this key before. With OSCOAP, the client additionally verifies the freshness of the response, i.e. that the response message is generated as an answer to the received request message (see Section 7).
This example deviates from encryption by default (see Section 8) just to illustrate the case of Integrity Protection only. If there is no compelling reason why the CoAP message should be in plaintext, then it MUST be encrypted.

D.1.2. Additional Encryption of CoAP Message

Here is an example of a GET request/response message exchange passing an intermediate node protected with the Enc option. The example illustrates a client requesting a blood sugar measurement resource (GET /glucose) and receiving the value 220 mg/dl. Uri-Path and Payload are encrypted and integrity protected. Code is integrity protected only (see Section 6.1 and Section 6.2).

Client  Proxy  Server

```
+-----> | GET | Code: 0.01 (GET)  
|       |     | Token: 0x83        
|       |     | Object-Security:  
|       |     |   \"seq\":\"15b7\",  
|       |     |     \"cid\":\"34e3c5fdca1509bd\",  
|       |     |       \{"glucose\}\}, \<Tag\> 

+-----> | GET | Code: 0.01 (GET)  
|       |     | Token: 0xbe        
|       |     | Object-Security:  
|       |     |   \"seq\":\"15b7\",  
|       |     |     \"cid\":\"34e3c5fdca1509bd\",  
|       |     |       \{"glucose\}\}, \<Tag\> 

<-----+ | 2.05 | Code: 2.05 (Content)  
|      |     | Token: 0xbe  
|      |     | Object-Security:  
|      |     | Payload:  
|      |     |   \"seq\":\"32c9\",  
|      |     |     \"cid\":\"c09bda155fd34e3c\",  
|      |     |       \{220\}\}, \<Tag\> 

<-----+ | 2.05 | Code: 2.05 (Content)  
|      |     | Token: 0x83  
|      |     | Object-Security:  
|      |     | Payload:  
|      |     |   \"seq\":\"32c9\",  
|      |     |     \"cid\":\"c09bda155fd34e3c\",  
|      |     |       \{220\}\}, \<Tag\> 
```

Figure 8: CoAP GET protected with OSCOAP. The bracket \{ ... \} indicates encrypted data.
Since the request message (GET) does not support payload, the OSCOAP object is carried in the Object-Security option. Since the response message (Content) supports payload, the Object-Security option is empty and the OSCOAP object is carried in the payload.

The Context Identifier is a hint to the receiver indicating which security context was used to encrypt and integrity protect the message, and may be used as an identifier for the AEAD secret key. One key is used for each direction of the message transfer.

The server and client can verify that the Sequence Number has not been received and used with this key before, and the client additionally verifies the freshness of the response, i.e. that the response message is generated as an answer to the received request message (see Section 7).

D.2. Payload Protection

This section gives examples that illustrate Object Security of Content (OSCON), see Appendix C). The assumption here is that only the intended receiver(s) has the relevant security context related to the resource. In case of a closed group of recipients of the same object, e.g. in Information-Centric Networking or firmware update distribution, it may be necessary to support symmetric key encryption in combination with digital signature.

D.2.1. Proxy Caching

This example outlines how a proxy forwarding request and response of one client can cache a response whose payload is a OSCON object, and serve this response to another client request, such that both clients can verify integrity and non-replay.
D.2.2. Publish-Subscribe

This example outlines a publish-subscribe setting where the payload is encrypted, integrity and replay protected end-to-end between Publisher and Subscriber. The example applies for example to closed
user groups of a single data source and illustrates a subscription registration and a later publication of birch pollen count of 300 per cubic meters. The PubSub Broker can define the Observe count arbitrarily (as could any intermediary node, even in OSCOAP), but cannot manipulate the Sequence Number without being possible to detect.

<table>
<thead>
<tr>
<th>Sub-</th>
<th>PubSub-</th>
<th>Pub-</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Broker</td>
<td>lisher</td>
</tr>
<tr>
<td>+-----+</td>
<td>Code: 0.01 (GET)</td>
<td></td>
</tr>
<tr>
<td>GET</td>
<td>Token: 0x72</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uri-Path: ps</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uri-Path: birch-pollen</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Observe: 0 (register)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>Code: 2.05 (Content)</td>
<td></td>
</tr>
<tr>
<td>2.05</td>
<td>Token: 0x72</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Observe: 1</td>
<td></td>
</tr>
</tbody>
</table>
|       | Payload: ["seq":"15b7",
|       | "cid":"c09bda155fd34e3c",
|       | ("270"), <Tag>]

| ------| Code: 0.03 (PUT) |
| PUT   | Token: 0x1f |
|       | Uri-Path: ps |
|       | Uri-Path: birch-pollen |
|       | Payload: ["seq":"15b8",
|       | "cid":"c09bda155fd34e3c",
|       | ("300"), <Tag>]

| +-----+ | Code: 2.04 (Changed) |
| 2.04  | Token: 0x1f |
|       | |
| ------| Code: 2.05 (Content) |
| 2.05  | Token: 0x72 |
|       | Observe: 2 |
|       | Payload: ["seq":"15b8",
|       | "cid":"c09bda155fd34e3c",
|       | ("300"), <Tag>]

Figure 10: Publish-subscribe protected with OSCON. The bracket { ... } indicates encrypted data.
This example deviates from encryption by default (see Section 8) just to illustrate Integrity Protection only in the case of OSCON. If there is no compelling reason why the payload should be in plaintext, then encryption MUST be used.

D.2.3. Transporting Authorization Information

This example outlines the transportation of authorization information from a node producing (Authorization Server, AS) to a node consuming (Resource Server, RS) such information. Authorization information may for example be an authorization decision with respect to a Client (C) accessing a Resource to be enforced by RS, see e.g. [I-D.ietf-ace-actors] or [I-D.seitz-ace-core-authz]. Here, C is clearly not trusted with modifying the information, but may need to be involved in mediating the authorization information to the RS, for example, because AS and RS does not have direct connectivity. So end-to-end security is required and object security ("access tokens") is the natural candidate.

This example considers the authorization information to be encapsulated in a OSCON object, generated by AS. How C accesses the OSCON object is out of scope for this example, it may e.g. be using CoAP. C then requests RS to configure the authorization information in the OSCON object by doing POST to /authz-info. This particular resource has a default access policy that only new messages signed by AS are authorized. RS thus verifies the integrity and sequence number by using the existing security context for the AS, and responds accordingly, a) or b), see Figure 11.
Figure 11: Protected Transfer of Access Token using OSCON

Authors’ Addresses

Goeran Selander
Ericsson
Farogatan 6
Kista 16480
Sweden

Email: goran.selander@ericsson.com

John Mattsson
Ericsson
Farogatan 6
Kista 16480
Sweden

Email: john.mattsson@ericsson.com
Francesca Palombini
Ericsson
Farogatan 6
Kista 16480
Sweden

Email: francesca.palombini@ericsson.com

Ludwig Seitz
SICS Swedish ICT
Scheelvagen 17
Lund 22370
Sweden

Email: ludwig@sics.se