COSE Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 26, 2017

J. Schaad
August Cellars
November 22, 2016

CBOR Object Signing and Encryption (COSE)
draft-ietf-cose-msg-24

Abstract

Concise Binary Object Representation (CBOR) is data format designed for small code size and small message size. There is a need for the ability to have basic security services defined for this data format. This document defines the CBOR Object Signing and Encryption (COSE) specification. This specification describes how to create and process signature, message authentication codes and encryption using CBOR for serialization. This specification additionally specifies how to represent 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

Copyright Notice

Table of Contents

1. Introduction   4
  1.1. Design changes from JOSE   5
  1.2. Requirements Terminology   6
  1.3. CBOR Grammar   6
  1.4. CBOR Related Terminology   7
  1.5. Document Terminology   8
2. Basic COSE Structure   8
3. Header Parameters 10
  3.1. Common COSE Headers Parameters 12
4. Signing Objects 16
  4.1. Signing with One or More Signers 16
  4.2. Signing with One Signer 18
  4.3. Externally Supplied Data 19
  4.4. Signing and Verification Process 20
  4.5. Computing Counter Signatures 22
5. Encryption Objects 22
  5.1. Enveloped COSE Structure 22
    5.1.1. Content Key Distribution Methods 24
  5.2. Single Recipient Encrypted 25
  5.3. How to encrypt and decrypt for AEAD Algorithms 25
  5.4. How to encrypt and decrypt for AE Algorithms 28
6. MAC Objects 29
  6.1. MACed Message with Recipients 30
  6.2. MACed Messages with Implicit Key 31
  6.3. How to compute and verify a MAC 31
7. Key Objects 33
  7.1. COSE Key Common Parameters 33
8. Signature Algorithms 36
  8.1. ECDSA 37
    8.1.1. Security Considerations 39
  8.2. Edwards-curve Digital Signature Algorithms (EdDSA) . . . 40

```
      8.2.1.  Security Considerations . . . . . . . . . . . . . . .  41
   9.  Message Authentication (MAC) Algorithms . . . . . . . . . . .  41
     9.1.  Hash-based Message Authentication Codes (HMAC)  . . . . .  41
       9.1.1.  Security Considerations . . . . . . . . . . . . . . .  43
     9.2.  AES Message Authentication Code (AES-CBC-MAC) . . . . . .  43
       9.2.1.  Security Considerations . . . . . . . . . . . . . . .  44
   10. Content Encryption Algorithms . . . . . . . . . . . . . . . .  45
     10.1.  AES GCM  . . . . . . . . . . . . . . . . . . . . . . . .  45
       10.1.1.  Security Considerations  . . . . . . . . . . . . . .  46
     10.2.  AES CCM  . . . . . . . . . . . . . . . . . . . . . . . .  47
       10.2.1.  Security Considerations  . . . . . . . . . . . . . .  50
     10.3.  ChaCha20 and Poly1305  . . . . . . . . . . . . . . . . .  50
       10.3.1.  Security Considerations  . . . . . . . . . . . . . .  51
   11. Key Derivation Functions (KDF)  . . . . . . . . . . . . . . .  51
     11.1.  HMAC-based Extract-and-Expand Key Derivation Function
            (HKDF) . . . . . . . . . . . . . . . . . . . . . . . . .  52
     11.2.  Context Information Structure  . . . . . . . . . . . . .  54
   12. Content Key Distribution Methods  . . . . . . . . . . . . . .  59
     12.1.  Direct Encryption  . . . . . . . . . . . . . . . . . . .  59
       12.1.1.  Direct Key . . . . . . . . . . . . . . . . . . . . .  60
       12.1.2.  Direct Key with KDF  . . . . . . . . . . . . . . . .  60
     12.2.  Key Wrapping . . . . . . . . . . . . . . . . . . . . . .  62
       12.2.1.  AES Key Wrapping . . . . . . . . . . . . . . . . . .  63
     12.3.  Key Transport  . . . . . . . . . . . . . . . . . . . . .  64
     12.4.  Direct Key Agreement . . . . . . . . . . . . . . . . . .  64
       12.4.1.  ECDH . . . . . . . . . . . . . . . . . . . . . . . .  65
       12.4.2.  Security Considerations  . . . . . . . . . . . . . .  69
     12.5.  Key Agreement with Key Wrap  . . . . . . . . . . . . . .  69
       12.5.1.  ECDH . . . . . . . . . . . . . . . . . . . . . . . .  69
   13. Key Object Parameters . . . . . . . . . . . . . . . . . . . .  71
     13.1.  Elliptic Curve Keys  . . . . . . . . . . . . . . . . . .  72
       13.1.1.  Double Coordinate Curves . . . . . . . . . . . . . .  72
     13.2.  Octet Key Pair . . . . . . . . . . . . . . . . . . . . .  73
     13.3.  Symmetric Keys . . . . . . . . . . . . . . . . . . . . .  74
   14. CBOR Encoder Restrictions . . . . . . . . . . . . . . . . . .  75
   15. Application Profiling Considerations  . . . . . . . . . . . .  75
   16. IANA Considerations . . . . . . . . . . . . . . . . . . . . .  77
     16.1.  CBOR Tag assignment  . . . . . . . . . . . . . . . . . .  77
     16.2.  COSE Header Parameters Registry  . . . . . . . . . . . .  77
     16.3.  COSE Header Algorithm Parameters Registry  . . . . . . .  78
     16.4.  COSE Algorithms Registry . . . . . . . . . . . . . . . .  78
     16.5.  COSE Key Common Parameters Registry  . . . . . . . . . .  79
     16.6.  COSE Key Type Parameters Registry  . . . . . . . . . . .  80
     16.7.  COSE Key Type Registry . . . . . . . . . . . . . . . . .  81
     16.8.  COSE Elliptic Curve Parameters Registry  . . . . . . . .  81
     16.9.  Media Type Registrations . . . . . . . . . . . . . . . .  82
       16.9.1.  COSE Security Message  . . . . . . . . . . . . . . .  82
       16.9.2.  COSE Key media type  . . . . . . . . . . . . . . . .  83
```

16.10. CoAP Content-Format Registrations 85
16.11. Expert Review Instructions 86
17. Implementation Status 87
17.1. Author's Versions 88
17.2. COSE Testing Library 88
18. Security Considerations 89
19. References 91
19.1. Normative References 91
19.2. Informative References 92
Appendix A. Guidelines for External Data Authentication of Algorithms 95
A.1. Algorithm Identification 95
A.2. Counter Signature Without Headers 98
Appendix B. Two Layers of Recipient Information 99
Appendix C. Examples 101
C.1. Examples of Signed Message 102
C.1.1. Single Signature 102
C.1.2. Multiple Signers 103
C.1.3. Counter Signature 104
C.1.4. Signature w/ Criticality 105
C.2. Single Signer Examples 106
C.2.1. Single ECDSA signature 106
C.3. Examples of Enveloped Messages 107
C.3.1. Direct ECDH 107
C.3.2. Direct plus Key Derivation 108
C.3.3. Counter Signature on Encrypted Content 109
C.3.4. Encrypted Content with External Data 111
C.4. Examples of Encrypted Messages 111
C.4.1. Simple Encrypted Message 111
C.4.2. Encrypted Message w/ a Partial IV 112
C.5. Examples of MACed messages 112
C.5.1. Shared Secret Direct MAC 112
C.5.2. ECDH Direct MAC 113
C.5.3. Wrapped MAC 114
C.5.4. Multi-recipient MACed message 115
C.6. Examples of MAC0 messages 116
C.6.1. Shared Secret Direct MAC 116
C.7. COSE Keys 117
C.7.1. Public Keys 117
C.7.2. Private Keys 118
Acknowledgments 120
Author's Address 121

1. Introduction

There has been an increased focus on 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) [RFC7049]. CBOR extended the data model of the JavaScript Object Notation (JSON) [RFC7159] 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 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 that specified how to process encryption, signatures and Message Authentication Code (MAC) operations, and how to encode keys using JSON. This document defines the CBOR Object Encryption and Signing (COSE) standard which does the same thing for the CBOR encoding format. While there is a strong attempt to keep the flavor of the original JOSE documents, two considerations are taken into account:

- o CBOR has capabilities that are not present in JSON and are appropriate to use. 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.

- o 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 was not always the same.

1.1. Design changes from JOSE

- o Define a single top message structure so that encrypted, signed and MACed messages can easily be identified and still have a consistent view.

- o Signed messages distinguish between the protected and unprotected parameters that relate to the content from those that relate to the signature.

- o MACed messages are separated from signed messages.

- o MACed messages have the ability to use the same set of recipient algorithms as enveloped messages for obtaining the MAC authentication key.

- o Use binary encodings for binary data rather than base64url encodings.

- o Combine the authentication tag for encryption algorithms with the cipher text.

- o The set of cryptographic algorithms has been expanded in some directions, and trimmed in others.

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 is currently no standard CBOR grammar available for use by specifications. The CBOR structures are therefore described in prose.

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 CBOR Data Definition Language (CDDL) [I-D.greevenbosch-appsawg-cbor-cddl]. In this specification, the following primitive types are used:

any - non-specific value that permits all CBOR values to be placed here.

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).

Two syntaxes from CDDL appear in this document as shorthand. These are:

FOO / BAR - indicates that either FOO or BAR can appear here

[+ FOO] - indicates that the type FOO appears one or more times in an array

As well as the prose description, a version of a CBOR grammar is presented in CDDL. Since CDDL has not been published as an RFC, this grammar may not work with the final version of CDDL. The CDDL grammar is informational, the prose description is normative.

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 > as an entity.)

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

CDDL expects the initial non-terminal symbol to be the first symbol in the file. For this reason the first fragment of CDDL is presented here.

```
start = COSE_Messages / COSE_Key / COSE_KeySet / Internal_Types

; This is defined to make the tool quieter:
Internal_Types = Sig_structure / Enc_structure / MAC_structure /
        COSE_KDF_Context
```

The non-terminal Internal_Types is defined for dealing with the automated validation tools used during the writing of this document. It references those non-terminals that are used for security computations, but are not emitted for transport.

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 strings, negative integers and unsigned integers as map keys. The integers are used for compactness of encoding and easy comparison. The inclusion of strings allows for an additional range of short encoded values to be used as well. Since the word "key" is mainly used in its other meaning, as a cryptographic key, we use the term "label" for this usage as a map key.

The presence of a label in a COSE map which is not a string or an integer is an error. Applications can either fail processing or process messages with incorrect labels, however they MUST NOT create messages with incorrect labels.

A CDDL grammar fragment is defined that defines the non-terminals 'label', as in the previous paragraph and 'values', which permits any value to be used.

```
label = int / tstr
values = any
```

1.5. Document Terminology

In this document, we use the following terminology:

Byte is a synonym for octet.

Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use in constrained systems. It is defined in [RFC7252].

Authenticated Encryption (AE) [RFC5116] algorithms are those encryption algorithms which provide an authentication check of the contents algorithm with the encryption service.

Authenticated Encryption with Authenticated Data (AEAD) [RFC5116] algorithms provide the same content authentication service as AE algorithms, but additionally provide for authentication of non-encrypted data as well.

2. Basic COSE Structure

The COSE object structure is designed so that there can be a large amount of common code when parsing and processing the different types of security messages. All of the message structures are built on the CBOR array type. The first three elements of the array always contain the same 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 cipher text as appropriate. The content may be detached, but the location is still used. The content is wrapped in a bstr when present and is a nil value when detached.

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

COSE messages are also built using the concept of layers to separate different types of cryptographic concepts. As an example of how this works, consider the COSE_Encrypt message (Section 5.1). This message type is broken into two layers: the content layer and the recipient layer. In the content layer, the plain text is encrypted and information about the encrypted message are placed. In the recipient layer, the content encryption key (CEK) is encrypted and information about how it is encrypted for each recipient is placed. A single layer version of the encryption message COSE_Encrypt0 (Section 5.2) is provided for cases where the CEK is pre-shared.

Identification of which type of message has been presented is done by the following methods:

1. The specific message type is known from the context. This may be defined by a marker in the containing structure or by restrictions specified by the application protocol.

2. The message type is identified by a CBOR tag. Messages with a CBOR tag are known in this specification as tagged messages, while those without the CBOR tag are known as untagged messages. This document defines a CBOR tag for each of the message structures. These tags can be found in Table 1.

3. When a COSE object is carried in a media type of application/cose, the optional parameter 'cose-type' can be used to identify the embedded object. The parameter is OPTIONAL if the tagged version of the structure is used. The parameter is REQUIRED if the untagged version of the structure is used. The value to use with the parameter for each of the structures can be found in Table 1.

4. When a COSE object is carried as a CoAP payload, the CoAP Content-Format Option can be used to identify the message content. The CoAP Content-Format values can be found in Table 26. The CBOR tag for the message structure is not required as each security message is uniquely identified.

CBOR Tag	cose-type	Data Item	Semantics
98	cose-sign	COSE_Sign	COSE Signed Data Object
18	cose-sign1	COSE_Sign1	COSE Single Signer Data Object
96	cose-encrypt	COSE_Encrypt	COSE Encrypted Data Object
16	cose-encrypt0	COSE_Encrypt0	COSE Single Recipient Encrypted Data Object
97	cose-mac	COSE_Mac	COSE Mac-ed Data Object
17	cose-mac0	COSE_Mac0	COSE Mac w/o Recipients Object

Table 1: COSE Message Identification

The following CDDL fragment identifies all of the top messages defined in this document. Separate non-terminals are defined for the tagged and the untagged versions of the messages.

```
COSE_Messages = COSE_Untagged_Message / COSE_Tagged_Message

COSE_Untagged_Message = COSE_Sign / COSE_Sign1 /
    COSE_Encrypt / COSE_Encrypt0 /
    COSE_Mac / COSE_Mac0

COSE_Tagged_Message = COSE_Sign_Tagged / COSE_Sign1_Tagged /
    COSE_Encrypt_Tagged / COSE_Encrypt0_Tagged /
    COSE_Mac_Tagged / COSE_Mac0_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 except for keys. While these buckets are present, they may not all be usable in all instances. For example, while the protected bucket is defined as part of the recipient structure, some of the algorithms used for recipient structures do not provide for authenticated data. If this is the case, the protected bucket is left empty.

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 have 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 a zero length map as a zero length string rather than as a zero length map (encoded as h'a0'). The zero length binary encoding is preferred because it is both shorter and the version used in the serialization structures for cryptographic computation. After encoding the map, the value is wrapped in the binary object. 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 avoids 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. With the exception of the COSE_Sign structure, the only data that needs 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" registry (Section 16.2). Some common parameters are defined in the next section, but a number of parameters are defined throughout this document.

Labels in each of the maps MUST be unique. When processing messages, if a label appears multiple times, the message MUST be rejected as malformed. Applications SHOULD verify that the same label does not occur in both the protected and unprotected headers. If the message is not rejected as malformed, attributes MUST be obtained from the protected bucket before they are obtained from the unprotected bucket.

The following CDDL fragment represents the two header buckets. A group Headers is defined in CDDL that represents the two buckets in which attributes are placed. This group is used to provide these two fields consistently in all locations. A type is also defined which represents the map of common headers.

```
Headers = (
    protected : empty_or_serialized_map,
    unprotected : header_map
)

header_map = {
    Generic_Headers,
    * label => values
}

empty_or_serialized_map = bstr .cbor header_map / bstr .size 0
```

3.1. Common COSE Headers Parameters

This section defines a set of common header parameters. A summary of these parameters can be found in Table 2. This table should be consulted to determine the value of label, and 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 authenticated where the ability to do so exists. This support is provided by AEAD algorithms or construction (COSE_Sign, COSE_Sign0, COSE_Mac and COSE_Mac0). This authentication can be done either by placing the header in the protected header bucket or as part of the externally supplied data. The value is taken from the "COSE Algorithms" Registry (see Section 16.4).

crit: The parameter is used to indicate which protected header labels an application that is processing a message is required to understand. Parameters defined in this document do not need to be included as they should be understood by all implementations. When present, this parameter MUST be placed in the protected header bucket. The array MUST have at least one value in it. Not all labels need to be included in the 'crit' parameter. The rules for deciding which header labels are placed in the array are:

* Integer labels in the range of 0 to 8 SHOULD be omitted.

* Integer labels in the range -1 to -128 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 common parameters associated with that algorithm. Integer labels in the range -129 to -65536 SHOULD be included as these would be less common parameters that might not be generally supported.

* Labels for parameters required for an application MAY be omitted. Applications should have a statement if the label can be omitted.

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 cipher text fields. Integers are from the "CoAP Content-Formats" IANA registry table [COAP.Formats]. Text values following the syntax of "<type-name>/<subtype-name>" where <type-name> and <subtype-name> are defined in Section 4.2 of [RFC6838]. Leading and trailing whitespace is also omitted. Textual content values along with parameters and subparameters can be located using the IANA "Media Types" registry. Applications SHOULD provide this parameter if the content structure is potentially ambiguous.

kid: This parameter identifies one piece of data that can be used as input to find the needed cryptographic key. The value of this parameter can be matched against the 'kid' member in a COSE_Key structure. Other methods of key distribution can define an equivalent field to be matched. Applications MUST NOT assume that 'kid' values are unique. There may be more than one key with the same 'kid' value, so all of the keys associated with this 'kid' may need to be checked. The internal structure of 'kid' values is not defined and cannot be relied on by applications. Key identifier values are hints about which key to use. This is not a security critical field. For this reason, it can be placed in the unprotected headers bucket.

IV: This parameter holds the Initialization Vector (IV) value. For some symmetric encryption algorithms this may be referred to as a nonce. The IV can be placed in the unprotected header as modifying the IV will cause the decryption to yield plaintext that is readily detectable as garbled.

Partial IV This parameter holds a part of the IV value. When using the COSE_Encrypt0 structure, a portion of the IV can be part of the context associated with the key. This field is used to carry a value that causes the IV to be changed for each message. The IV can be placed in the unprotected header as modifying the IV will cause the decryption to yield plaintext that is readily detectable as garbled. The 'Initialization Vector' and 'Partial Initialization Vector' parameters MUST NOT both be present in the same security layer.
The message IV is generated by the following steps:

1. Left pad the partial IV with zeros to the length of IV.

2. XOR the padded partial IV with the context IV.

counter signature: This parameter holds one or more counter signature values. Counter signatures provide a method of having a second party sign some data. The counter signature parameter can occur as an unprotected attribute in any of the following structures: COSE_Sign1, COSE_Signature, COSE_Encrypt, COSE_recipient, COSE_Encrypt0, COSE_Mac and COSE_Mac0. These structures all have the same beginning elements so that a consistent calculation of the counter signature can be computed. Details on computing counter signatures are found in Section 4.5.

name	label	value type	value registry	description
alg	1	int / tstr	COSE Algorithms registry	Cryptographic algorithm to use
crit	2	[+ label]	COSE Header Labels registry	Critical headers to be understood
content type	3	tstr / uint	CoAP Content-Formats or Media Types registry	Content type of the payload
kid	4	bstr		Key identifier
IV	5	bstr		Full Initialization Vector
Partial IV	6	bstr		Partial Initialization Vector
counter signature	7	COSE_Signature / [+ COSE_Signature]		CBOR encoded signature structure

Table 2: Common Header Parameters

The CDDL fragment that represents the set of headers defined in this section is given below. Each of the headers is tagged as optional because they do not need to be in every map; headers required in specific maps are discussed above.

```
Generic_Headers = (
    ? 1 => int / tstr,  ; algorithm identifier
    ? 2 => [+label],    ; criticality
    ? 3 => tstr / int,  ; content type
    ? 4 => bstr,        ; key identifier
    ? 5 => bstr,        ; IV
    ? 6 => bstr,        ; Partial IV
    ? 7 => COSE_Signature / [+COSE_Signature] ; Counter signature
)
```

4. Signing Objects

COSE supports two different signature structures. COSE_Sign allows for one or more signatures to be applied to the same content. COSE_Sign1 is restricted to a single signer. The structures cannot be converted between each other; as the signature computation includes a parameter identifying which structure is being used, the converted structure will fail signature validation.

4.1. Signing with One or More Signers

The COSE_Sign structure allows for one or more signatures to be applied to a message payload. Parameters relating to the content and parameters relating to the signature are carried along with the signature itself. These parameters may be authenticated by the signature, or just present. An example of a parameter about the content is the content type. Examples of parameters about the signature would be the algorithm and key used to create the signature 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 for 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 Edwards Digital Signature Algorithm (EdDSA) [I-D.irtf-cfrg-eddsa] signature algorithm and with the Elliptic Curve Digital Signature Algorithm (ECDSA) [DSS] 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].

The signature structure can be encoded either as tagged or untagged depending on the context it will be used in. A tagged COSE_Sign structure is identified by the CBOR tag TBD1. The CDDL fragment that represents this is:

```
COSE_Sign_Tagged = #6.98(COSE_Sign)
```

A COSE Signed Message is defined in 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 Signed Messages can be found in Appendix C.1.

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

protected as described in Section 3.

unprotected as 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 ("detached content"), 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.

Note: When a signature with message recovery algorithm is used (Section 8), the maximum number of bytes that can be recovered is the length of the payload. The size of the payload is reduced by the number of bytes that will be recovered. If all of the bytes of the payload are consumed, then the payload is encoded as a zero length binary string rather than as being absent.

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

The CDDL fragment that represents the above text for COSE_Sign follows.

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

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

protected as described in Section 3.

unprotected as described in Section 3.

signature contains the computed signature value. The type of the field is a bstr. Algorithms MUST specify padding if the signature value is not a multiple of 8 bits.

The CDDL fragment that represents the above text for COSE_Signature follows.

```
COSE_Signature =  [
    Headers,
    signature : bstr
]
!
```

4.2. Signing with One Signer

The COSE_Sign1 signature structure is used when only one signature is going to be placed on a message. The parameters dealing with the content and the signature are placed in the same pair of buckets rather than having the separation of COSE_Sign.

The structure can be encoded either tagged or untagged depending on the context it will be used in. A tagged COSE_Sign1 structure is identified by the CBOR tag TBD7. The CDDL fragment that represents this is:

```
COSE_Sign1_Tagged = #6.18(COSE_Sign1)
```

The CBOR object that carries the body, the signature, and the information about the body and signature is called the COSE_Sign1 structure. Examples of COSE_Sign1 messages can be found in Appendix C.2.

The COSE_Sign1 structure is a CBOR array. The fields of the array in order are:

protected as described in Section 3.

unprotected as described in Section 3.

payload as described in Section 4.1.

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

The CDDL fragment that represents the above text for COSE_Sign1 follows.

```
COSE_Sign1 = [
    Headers,
    payload : bstr / nil,
    signature : bstr
]
```

4.3. Externally Supplied Data

One of the features offered in the COSE document is the ability for applications to provide additional data to be authenticated, 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. Examples of data that can be placed in this location would be the CoAP code or CoAP options. If the data is in the header section, then it is available for proxies to help in performing its operations. For example, the Accept Option can be used by a proxy to determine if an appropriate value is in the Proxy's cache. But the sender can prevent a proxy from changing the set of values that it will accept by including that value in the resulting authentication tag. However, it may also be desired to protect these values so that if they are modified in transit, it can be detected.

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, applications need to ensure that the same byte string is not produced if there are different inputs. This could occur by appending the strings 'AB' and 'CDE' or by appending the strings 'ABC' and 'DE'. This is usually addressed by making fields a fixed width and/or encoding the length of the field as part of the output. 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, an 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.

- Applications need to ensure that the byte stream is going to be the same on both sides. Using options from CoAP might give a problem if the same relative numbering is kept. An intermediate node could insert or remove an option, changing how the relative number is done. An application would need to specify that the relative number must be re-encoded to be relative only to the options that are in the external data.

4.4. Signing and Verification Process

In order to create a signature, a well-defined byte stream is needed. The Sig_struture is used to create the canonical form. This signing and verification process takes in the body information (COSE_Sign or COSE_Sign1), the signer information (COSE_Signature), and the application data (external source). A Sig_structure is a CBOR array. The fields of the Sig_struture in order are:

1. A text string identifying the context of the signature. The context string is:

   "Signature" for signatures using the COSE_Signature structure.

   "Signature1" for signatures using the COSE_Sign1 structure.

   "CounterSignature" for signatures used as counter signature attributes.

2. The protected attributes from the body structure encoded in a bstr type. If there are no protected attributes, a bstr of length zero is used.

3. The protected attributes from the signer structure encoded in a bstr type. If there are no protected attributes, a bstr of length zero is used. This field is omitted for the COSE_Sign1 signature structure.

4. 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. (See Section 4.3 for application guidance on constructing this field.)

5. The payload to be signed encoded in a bstr type. The payload is placed here independent of how it is transported.

The CDDL fragment that describes the above text is.

```
Sig_structure = [
    context : "Signature" / "Signature1" / "CounterSignature",
    body_protected : empty_or_serialized_map,
    ? sign_protected : empty_or_serialized_map,
    external_aad : bstr,
    payload : bstr
]
```

How to compute a signature:

1. Create a Sig_structure and populate it with the appropriate fields.

2. Create the value ToBeSigned by encoding the Sig_structure to a byte string, using the encoding described in Section 14.

3. 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).

4. Place the resulting signature value in the 'signature' field of the array.

The steps for verifying a signature are:

1. Create a Sig_structure object and populate it with the appropriate fields.

2. Create the value ToBeSigned by encoding the Sig_structure to a byte string, using the encoding described in Section 14.

3. 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, the application may also perform the appropriate checks to ensure that the key is correctly paired with the signing identity and that the signing identity is authorized before performing actions.

4.5. Computing Counter Signatures

Counter signatures provide a method of associating different signature generated by different signers with 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_Encrypt 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 counter signature on a signature can be found in Appendix C.1.3. An example of a counter signature in an encryption object can be found in Appendix C.3.3.

The creation and validation of counter signatures over the different items relies on the fact that the structure of the 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 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_Encrypt structure, the body_protected and payload items can be mapped into the Sig_structure in the same manner as from the COSE_Sign structure.

It should be noted that only a signature algorithm with appendix (see Section 8) can be used for counter signatures. This is because the body should be able to be processed without having to evaluate the counter signature, and this is not possible for signature schemes with message recovery.

5. Encryption Objects

COSE supports two different encryption structures. COSE_Encrypt0 is used when a recipient structure is not needed because the key to be used is known implicitly. COSE_Encrypt is used the rest of the time. This includes cases where there are multiple recipients or a recipient algorithm other than direct is used.

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 protected parameters associated with the content are authenticated by the content encryption algorithm. The protected parameters associated with the recipient are authenticated by the recipient algorithm (when the algorithm supports it). Examples of parameters about the content are the type of the content and the content encryption algorithm. Examples of parameters about the recipient are the recipient's key identifier and the recipient's encryption algorithm.

The same techniques and structures are used for encrypting both the plain text and the keys. This is different from the approach used by both CMS [RFC5652] and JSON Web Encryption (JWE) [RFC7516] where different structures are used for the content layer and for the recipient layer. Two structures are defined: COSE_Encrypt 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.3.

The COSE_Encrypt structure can be encoded either tagged or untagged depending on the context it will be used in. A tagged COSE_Encrypt structure is identified by the CBOR tag TBD2. The CDDL fragment that represents this is:

```
COSE_Encrypt_Tagged = #6.96(COSE_Encrypt)
```

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

protected as described in Section 3.

unprotected as described in Section 3. '

ciphertext contains the cipher text encoded as a bstr. If the cipher text is to be transported independently of the control information about the encryption process (i.e., detached content) then the 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.

The CDDL fragment that corresponds to the above text is:

```
COSE_Encrypt = [
    Headers,
    ciphertext : bstr / nil,
    recipients : [+COSE_recipient]
]
```

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

protected as described in Section 3.

unprotected as described in Section 3.

ciphertext contains the encrypted key encoded as a bstr. All encoded keys are symetric keys, the binary value of the key is the content. 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. (An example of this can be found in Appendix B.) If there are no recipient information structures, this element is absent.

The CDDL fragment that corresponds to the above text for COSE_recipient is:

```
COSE_recipient = [
    Headers,
    ciphertext : bstr / nil,
    ? recipients : [+COSE_recipient]
]
```

5.1.1. Content Key Distribution Methods

An 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 depend 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 content key distribution methods 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. No key transport algorithms are defined in this document.

passwords: The CEK is encrypted in a KEK that is derived from a password. No password algorithms are defined in this document.

5.2. Single Recipient Encrypted

The COSE_Encrypt0 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.

Examples of encrypted messages can be found in Appendix C.3.

The COSE_Encrypt0 structure can be encoded either tagged or untagged depending on the context it will be used in. A tagged COSE_Encrypt0 structure is identified by the CBOR tag TBD3. The CDDL fragment that represents this is:

```
COSE_Encrypt0_Tagged = #6.16(COSE_Encrypt0)
```

The COSE_Encrypt0 structure is a CBOR array. The fields of the array in order are:

protected as described in Section 3.

unprotected as described in Section 3.

ciphertext as described in Section 5.1.

The CDDL fragment for COSE_Encrypt0 that corresponds to the above text is:

```
COSE_Encrypt0 = [
    Headers,
    ciphertext : bstr / nil,
]
```

5.3. How to encrypt and decrypt for AEAD Algorithms

The encryption algorithm for AEAD algorithms is fairly simple. The first step is to create a consistent byte stream for the authenticated data structure. For this purpose, we use an Enc_structure. The Enc_structure is a CBOR array. The fields of the Enc_structure in order are:

1. A text string identifying the context of the authenticated data structure. The context string is:

   "Encrypt0" for the content encryption of a COSE_Encrypt0 data structure.

   "Encrypt" for the first layer of a COSE_Encrypt data structure (i.e., for content encryption).

   "Enc_Recipient" for a recipient encoding to be placed in an COSE_Encrypt data structure.

   "Mac_Recipient" for a recipient encoding to be placed in a MACed message structure.

   "Rec_Recipient" for a recipient encoding to be placed in a recipient structure.

2. The protected attributes from the body structure encoded in a bstr type. If there are no protected attributes, a bstr of length zero is used.

3. The protected attributes from the application encoded in a bstr type. If this field is not supplied, it defaults to a zero length bstr. (See Section 4.3 for application guidance on constructing this field.)

The CDDL fragment that describes the above text is:

```
Enc_structure = [
    context : "Encrypt" / "Encrypt0" / "Enc_Recipient" /
        "Mac_Recipient" / "Rec_Recipient",
    protected : empty_or_serialized_map,
    external_aad : bstr
]
```

How to encrypt a message:

1. Create an Enc_structure and populate it with the appropriate fields.

2. Encode the Enc_structure to a byte stream (AAD), using the encoding described in Section 14.

3. Determine the encryption key (K). 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 layer. Examples are key transport keys Section 12.3, key wrap keys Section 12.2.1 or pre-shared secrets.

Direct Encryption 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.) Examples of these algorithms are found in Section 12.1.2 and Section 12.4.1.

Other: The key is randomly or pseudo-randomly generated.

4. Call the encryption algorithm with K (the encryption key), P (the plain text) and AAD. 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 K (the encryption key) as the plain text.

How to decrypt a message:

1. Create a Enc_structure and populate it with the appropriate fields.

2. Encode the Enc_structure to a byte stream (AAD), using the encoding described in Section 14.

3. Determine the decryption 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 layer. Examples are key transport keys Section 12.3, key wrap keys Section 12.2.1 or pre-shared secrets.

   Direct Encryption 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.) Examples of these algorithms are found in Section 12.1.2 and Section 12.4.1.

Other: The key is determined by decoding and decrypting one of the recipient structures.

4. Call the decryption algorithm with K (the decryption key to use), C (the cipher text) and AAD.

5.4. How to encrypt and decrypt for AE Algorithms

How to encrypt a message:

1. Verify that the 'protected' field is empty.

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 layer. Examples are key transport keys Section 12.3, key wrap keys Section 12.2.1 or pre-shared secrets.

   Direct Encryption 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.) Examples of these algorithms are found in Section 12.1.2 and Section 12.4.1.

   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 K (the encryption key) as the plain text.

How to decrypt a message:

1. Verify that the 'protected' field is empty.

2. Verify that there was no external additional authenticated data supplied for this operation.

3. Determine the decryption 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 layer. Examples are key transport keys Section 12.3, key wrap keys Section 12.2.1 or pre-shared secrets.

   Direct Encryption 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.) Examples of these algorithms are found in Section 12.1.2 and Section 12.4.1.

   Other: The key is determined by decoding and decrypting one of the recipient structures.

4. Call the decryption algorithm with K (the decryption key to use), and C (the cipher text).

6. MAC Objects

COSE supports two different MAC structures. COSE_MAC0 is used when a recipient structure is not needed because the key to be used is implicitly known. COSE_MAC is used for all other cases. These include a requirement for multiple recipients, the key being unknown, and a recipient algorithm of other than direct.

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 they 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. (This knowledge of sender assumes that there are only two parties involved and you did not send the message to yourself.) The origination property can be obtained with both of the MAC message structures.

6.1. MACed Message with Recipients

The multiple recipient MACed 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 MACed messages can be found in Appendix C.5.

The MAC structure can be encoded either tagged or untagged depending on the context it will be used in. A tagged COSE_Mac structure is identified by the CBOR tag TBD4. The CDDL fragment that represents this is:

```
COSE_Mac_Tagged = #6.97(COSE_Mac)
```

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

protected as described in Section 3.

unprotected as 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 (i.e., detached content), then a nil CBOR value 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 as described in Section 5.1.

The CDDL fragment that represents the above text for COSE_Mac follows.

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

6.2. MACed Messages with Implicit Key

In this section, we describe the structure and methods to be used when doing MAC authentication for those cases where the recipient is implicitly known.

The MACed message uses the COSE_Mac0 structure defined in this section for carrying the body. Examples of MACed messages with an implicit key can be found in Appendix C.6.

The MAC structure can be encoded either tagged or untagged depending on the context it will be used in. A tagged COSE_Mac0 structure is identified by the CBOR tag TBD6. The CDDL fragment that represents this is:

```
COSE_Mac0_Tagged = #6.17(COSE_Mac0)
```

The COSE_Mac0 structure is a CBOR array. The fields of the array in order are:

protected as described in Section 3.

unprotected as described in Section 3.

payload as described in Section 6.1.

tag contains the MAC value.

The CDDL fragment that corresponds to the above text is:

```
COSE_Mac0 = [
   Headers,
   payload : bstr / nil,
   tag : bstr,
]
```

6.3. How to compute and verify 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 fields of the MAC_structure in order are:

1. A text string that identifies the structure that is being encoded. This string is "MAC" for the COSE_Mac structure. This string is "MAC0" for the COSE_Mac0 structure.

2. The protected attributes from the COSE_MAC structure. If there are no protected attributes, a zero length bstr is used.

3. The protected attributes from the application encoded as a bstr type. If this field is not supplied, it defaults to a zero length binary string. (See Section 4.3 for application guidance on constructing this field.)

4. The payload to be MAC-ed encoded in a bstr type. The payload is placed here independent of how it is transported.

The CDDL fragment that corresponds to the above text is:

```
MAC_structure = [
     context : "MAC" / "MAC0",
     protected : empty_or_serialized_map,
     external_aad : bstr,
     payload : bstr
]
```

The steps to compute a MAC are:

1. Create a MAC_structure and populate it with the appropriate fields.

2. Create the value ToBeMaced by encoding the MAC_structure to a byte stream, using the encoding described in Section 14.

3. Call the MAC creation algorithm passing in K (the key to use), alg (the algorithm to MAC with) and ToBeMaced (the value to compute the MAC on).

4. Place the resulting MAC in the 'tag' field of the COSE_Mac or COSE_Mac0 structure.

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

The steps to verify a MAC are:

1. Create a MAC_structure object and populate it with the appropriate fields.

2. Create the value ToBeMaced by encoding the MAC_structure to a byte stream, using the encoding described in Section 14.

3. Obtain the cryptographic key from one of the recipients of the message.

4. Call the MAC creation algorithm passing in K (the key to use), alg (the algorithm to MAC with) and ToBeMaced (the value to compute the MAC on).

5. Compare the MAC value to the 'tag' field of the COSE_Mac or COSE_Mac0 structure.

7. Key Objects

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 "COSE Key Common Parameters" registry (Section 16.5). Additional parameters defined for specific key types can be found in the IANA "COSE Key Type Parameters" registry (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. Examples of Key Sets can be found in Appendix C.7.

Each element in a key set MUST be processed independently. If one element in a key set is either malformed or uses a key that is not understood by an application, that key is ignored and the other keys are processed normally.

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

The CDDL grammar describing COSE_Key and COSE_KeySet is:

```
COSE_Key = {
    1 => tstr / int,          ; kty
    ? 2 => bstr,              ; kid
    ? 3 => tstr / int,        ; alg
    ? 4 => [+ (tstr / int) ], ; key_ops
    ? 5 => bstr,              ; Base IV
    * label => values
}

COSE_KeySet = [+COSE_Key]
```

7.1. COSE Key Common Parameters

This document defines a set of common parameters for a COSE Key object. Table 3 provides a summary of the parameters defined in this section. There are also parameters that are defined for specific key types. Key type specific parameters can be found in Section 13.

name	label	CBOR type	registry	description
kty	1	tstr / int	COSE Key Common Parameters	Identification of the key type
alg	3	tstr / int	COSE Algorithm Values	Key usage restriction to this algorithm
kid	2	bstr		Key Identification value - match to kid in message
key_ops	4	[+ (tstr/int)]		Restrict set of permissible operations
Base IV	5	bstr		Base IV to be xor-ed with Partial IVs

Table 3: 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 21. 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 the trust decision process.

alg: This parameter is used to restrict the algorithm that is used with the 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 4. Algorithms define the values of key ops that are permitted to appear and are required for specific operations. The set of values matches that in [RFC7517] and [W3C.WebCrypto].

Base IV: This parameter is defined to carry the base portion of an IV. It is designed to be used with the partial IV header parameter defined in Section 3.1. This field provides the ability to associate a partial IV with a key that is then modified on a per message basis with the partial IV.

Extreme care needs to be taken when using a Base IV in an application. Many encryption algorithms lose security if the same IV is used twice.

If different keys are derived for each sender, using the same base IV with partial IVs starting at zero is likely to ensure that the IV would not be used twice for a single key. If different keys are derived for each sender, starting at the same base IV is likely to satisfy this condition. If the same key is used for multiple senders, then the application needs to provide for a method of dividing the IV space up between the senders. This could be done by providing a different base point to start from or a different partial IV to start with and restricting the number of messages to be sent before re-keying.

name	value	description
sign	1	The key is used to create signatures. Requires private key fields.
verify	2	The key is used for verification of signatures.
encrypt	3	The key is used for key transport encryption.
decrypt	4	The key is used for key transport decryption. Requires private key fields.
wrap key	5	The key is used for key wrapping.
unwrap key	6	The key is used for key unwrapping. Requires private key fields.
derive key	7	The key is used for deriving keys. Requires private key fields.
derive bits	8	The key is used for deriving bits not to be used as a key. Requires private key fields.
MAC create	9	The key is used for creating MACs.
MAC verify	10	The key is used for validating MACs.

Table 4: Key Operation Values

8. Signature Algorithms

There are two signature algorithm schemes. The first is signature with appendix. In this scheme, the message content is processed and a signature is produced, the signature is called the appendix. This is the scheme used by algorithms such as ECDSA and RSASSA-PSS. (In fact the SSA in RSASSA-PSS stands for Signature Scheme with Appendix.)

The signature functions for this scheme are:

signature = Sign(message content, key)

valid = Verification(message content, key, signature)

The second scheme is signature with message recovery. (An example of such an algorithm is [PVSig].) In this scheme, 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 smaller signatures, the signature size is still potentially large, but the message content has 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 that 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. Finally, 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. This means that mixing of the different schemes in a single message is not supported, and if a recovery signature scheme is used, then the same amount of content needs to be consumed by all of the signatures.

The signature functions for this scheme are:

signature, message sent = Sign(message content, key)

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

Signature algorithms are used with the COSE_Signature and COSE_Sign1 structures. 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. Implementations SHOULD use a deterministic version of ECDSA such as the one defined in [RFC6979]. The use of a deterministic signature algorithms allows for systems to avoid relying on random number generators in order to avoid generating the same value of 'k' (the per-message random value). Biased generation of the value be attacked and collisions will lead to leaked keys. It additionally allows for doing deterministic tests for the signature algorithm. The use of deterministic ECDSA does not lessen the need to to have good random number generation when creating the private key.

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 5.

name	value	hash	description
ES256	-7	SHA-256	ECDSA w/ SHA-256
ES384	-35	SHA-384	ECDSA w/ SHA-384
ES512	-36	SHA-512	ECDSA w/ SHA-512

Table 5: 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' (2 coordinate Elliptic Curve) key type. Implementations need to check that the key type and curve are correct when creating and verifying a signature. Other documents can define 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 the same length 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 [I-D.moriarty-pkcs1] the signature is:
Signature = I2OSP(R, n) | I2OSP(S, n)
where n = ceiling(key_length / 8)

When using a COSE key for this algorithm, the following checks are made:

o The 'kty' field MUST be present and it MUST be 'EC2'.

- o If the 'alg' field is present, it MUST match the ECDSA signature algorithm being used.

- o If the 'key_ops' field is present, it MUST include 'sign' when creating an ECDSA signature.

- o If the 'key_ops' field is present, it MUST include 'verify' when verifying an ECDSA signature.

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.

Note: Use of this technique is 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 and allows for reproducible signature values, which helps testing.

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

- o 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.

- o 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.) As the hash algorithm is part of the signature algorithm identifier, this attack is mitigated by including signature algorithm identifier in the protected header.

8.2. Edwards-curve Digital Signature Algorithms (EdDSA)

[I-D.irtf-cfrg-eddsa] describes the elliptic curve signature scheme Edwards-curve Digital Signature Algorithm (EdDSA). In that document, the signature algorithm is instantiated using parameters for edwards25519 and edwards448 curves. The document additionally describes two variants of the EdDSA algorithm: Pure EdDSA, where no hash function is applied to the content before signing and, HashEdDSA where a hash function is applied to the content before signing and the result of that hash function is signed. For the EdDSA, the content to be signed (either the message or the pre-hash value) is processed twice inside of the signature algorithm. For use with COSE, only the pure EdDSA version is used. This is because it is not expected that extremely large contents are going to be needed and, based on the arrangement of the message structure, the entire message is going to need to be held in memory in order to create or verify a signature. This means that there does not appear to be a need to be able to do block updates of the hash, followed by eliminating the message from memory. Applications can provide the same features by defining the content of the message as a hash value and transporting the COSE object (with the hash value) and the content as separate items.

The algorithms defined in this document can be found in Table 6. A single signature algorithm is defined, which can be used for multiple curves.

name	value	description
EdDSA	-8	EdDSA

Table 6: EdDSA Algorithm Values

[I-D.irtf-cfrg-eddsa] describes the method of encoding the signature value.

When using a COSE key for this algorithm the following checks are made:

- o The 'kty' field MUST be present and it MUST be 'OKP' (Octet Key Pair).
- o The 'crv' field MUST be present, and it MUST be a curve defined for this signature algorithm.
- o If the 'alg' field is present, it MUST match 'EdDSA'.

- o If the 'key_ops' field is present, it MUST include 'sign' when creating an EdDSA signature.

- o If the 'key_ops' field is present, it MUST include 'verify' when verifying an EdDSA signature.

8.2.1. Security Considerations

How public values are computed is not the same when looking at EdDSA and ECDH, for this reason they should not be used with the other algorithm.

If batch signature verification is performed, a well-seeded cryptographic random number generator is REQUIRED. Signing and non-batch signature verification are deterministic operations and do not need random numbers of any kind.

9. Message Authentication (MAC) Algorithms

Message Authentication Codes (MACs) provide data authentication and integrity protection. They provide either no or very limited data origination. A MAC, for example, be used to prove the identity of the sender to a third party.

MACs use the same scheme 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 MAC functions are:

tag = MAC_Create(message content, key)

valid = MAC_Verify(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 using each of these constructions.

MAC algorithms are used in the COSE_Mac and COSE_Mac0 structures.

9.1. Hash-based Message Authentication Codes (HMAC)

The Hash-based Message Authentication Code algorithm (HMAC) [RFC2104][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 shown as solid since, 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 [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 [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 with truncation, 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 algorithms defined in this document can be found in Table 7.

name	value	Hash	Tag Length	description
HMAC 256/64	4	SHA-256	64	HMAC w/ SHA-256 truncated to 64 bits
HMAC 256/256	5	SHA-256	256	HMAC w/ SHA-256
HMAC 384/384	6	SHA-384	384	HMAC w/ SHA-384
HMAC 512/512	7	SHA-512	512	HMAC w/ SHA-512

Table 7: HMAC Algorithm Values

Some recipient algorithms carry the key while others derive a key from secret data. For those algorithms that carry the key (such as 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 (such as ECDH), the derived key MUST be the same size as the underlying hash function.

When using a COSE key for this algorithm, the following checks are made:

- The ’kty’ field MUST be present and it MUST be ’Symmetric’.

- If the 'alg' field is present, it MUST match the HMAC algorithm being used.

- If the 'key_ops' field is present, it MUST include 'MAC create' when creating an HMAC authentication tag.

- If the 'key_ops' field is present, it MUST include 'MAC verify' when verifying an HMAC authentication tag.

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 weakened hash algorithms. The current best known attack appears is to brute force 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]. (Note this is not the same algorithm as AES-CMAC [RFC4493]).

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 8.

name	value	key length	tag length	description
AES-MAC 128/64	14	128	64	AES-MAC 128 bit key, 64-bit tag
AES-MAC 256/64	15	256	64	AES-MAC 256 bit key, 64-bit tag
AES-MAC 128/128	25	128	128	AES-MAC 128 bit key, 128-bit tag
AES-MAC 256/128	26	256	128	AES-MAC 256 bit key, 128-bit tag

Table 8: AES-MAC Algorithm Values

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

When using a COSE key for this algorithm, the following checks are made:

o The 'kty' field MUST be present and it MUST be 'Symmetric'.

o If the 'alg' field is present, it MUST match the AES-MAC algorithm being used.

o If the 'key_ops' field is present, it MUST include 'MAC create' when creating an AES-MAC authentication tag.

o If the 'key_ops' field is present, it MUST include 'MAC verify' when verifying an AES-MAC authentication tag.

9.2.1. Security Considerations

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

o 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 and tag pairs. This can be addressed by using different keys for different length messages. The current structure mitigates this problem, as a specific encoding structure that includes lengths is built and signed. (CMAC also addresses this issue.)

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

- o 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 CEK is obtained.

COSE restricts 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 encryption functions are:

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.

These algorithms are used in COSE_Encrypt and COSE_Encrypt0.

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 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 9.

name	value	description
A128GCM	1	AES-GCM mode w/ 128-bit key, 128-bit tag
A192GCM	2	AES-GCM mode w/ 192-bit key, 128-bit tag
A256GCM	3	AES-GCM mode w/ 256-bit key, 128-bit tag

Table 9: Algorithm Value for AES-GCM

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

When using a COSE key for this algorithm, the following checks are made:

- o The ’kty’ field MUST be present and it MUST be ’Symmetric’.
- o If the ’alg’ field is present, it MUST match the AES-GCM algorithm being used.
- o If the ’key_ops’ field is present, it MUST include ’encrypt’ or ’wrap key’ when encrypting.
- o If the ’key_ops’ field is present, it MUST include ’decrypt’ or ’unwrap key’ when decrypting.

10.1.1. Security Considerations

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

- o The key and nonce pair MUST be unique for every message encrypted.
- o The total amount of data encrypted for a single key 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 so 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 growth (from the tag) 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 convention 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 both L and M. Less constrained devices 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 L and M.

The following values are used for L:

16 bits (2) limits messages to 2^16 bytes (64 KiB) in length. This is sufficiently long for messages in the constrained world. The nonce length is 13 bytes allowing for 2^(13*8) 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.

name	value	L	M	k	description
AES-CCM-16-64-128	10	16	64	128	AES-CCM mode 128-bit key, 64-bit tag, 13-byte nonce
AES-CCM-16-64-256	11	16	64	256	AES-CCM mode 256-bit key, 64-bit tag, 13-byte nonce
AES-CCM-64-64-128	12	64	64	128	AES-CCM mode 128-bit key, 64-bit tag, 7-byte nonce
AES-CCM-64-64-256	13	64	64	256	AES-CCM mode 256-bit key, 64-bit tag, 7-byte nonce
AES-CCM-16-128-128	30	16	128	128	AES-CCM mode 128-bit key, 128-bit tag, 13-byte nonce
AES-CCM-16-128-256	31	16	128	256	AES-CCM mode 256-bit key, 128-bit tag, 13-byte nonce
AES-CCM-64-128-128	32	64	128	128	AES-CCM mode 128-bit key, 128-bit tag, 7-byte nonce
AES-CCM-64-128-256	33	64	128	256	AES-CCM mode 256-bit key, 128-bit tag, 7-byte nonce

Table 10: Algorithm Values for AES-CCM

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

When using a COSE key for this algorithm, the following checks are made:

- o The 'kty' field MUST be present and it MUST be 'Symmetric'.
- o If the 'alg' field is present, it MUST match the AES-CCM algorithm being used.
- o If the 'key_ops' field is present, it MUST include 'encrypt' or 'wrap key' when encrypting.
- o If the 'key_ops' field is present, it MUST include 'decrypt' or 'unwrap key' when decrypting.

10.2.1. Security Considerations

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

- o The key and nonce pair MUST be unique for every message encrypted. Note that the value of L influences the number of unique nonces.
- o 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 portions of the plaintext are 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 an AEAD mode that is defined in [RFC7539]. This is an 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 11.

name	value	description
ChaCha20/Poly1305	24	ChaCha20/Poly1305 w/ 256-bit key, 128-bit tag

Table 11: Algorithm Value for AES-GCM

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

When using a COSE key for this algorithm, the following checks are made:

- o The 'kty' field MUST be present and it MUST be 'Symmetric'.
- o If the 'alg' field is present, it MUST match the ChaCha20/Poly1305 algorithm being used.
- o If the 'key_ops' field is present, it MUST include 'encrypt' or 'wrap key' when encrypting.
- o If the 'key_ops' field is present, it MUST include 'decrypt' or 'unwrap key' when decrypting.

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 secret value comes in three flavors:

- o Secrets that are uniformly random: This is the type of secret that is created by a good random number generator.

- o Secrets that are not uniformly random: This is type of secret that is created by operations like key agreement.

- o 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 reasonably 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 PBES2 [I-D.moriarty-pkcs5-v2dot1] need to be used for that type of secret.

The same KDF function can be setup to deal with the first two types of secrets in a different way. The KDF function defined in Section 11.1 is such a function. This is reflected in the set of algorithms defined for HKDF.

When using KDF functions, one component that is 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. These elements are used for all of the recipient algorithms defined in this document that require a KDF process. These algorithms are defined in Section 12.1.2, Section 12.4.1, and Section 12.5.1.

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 value that is used to change the generation process. The salt value can be either public or private. If the salt is public and carried in the message, then the 'salt' algorithm header parameter defined in Table 13 is used. 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. This parameter is protected by being included in the key computation and does not need to be separately authenticated. The salt value does not need to be unique for every message sent.

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 in which the material is going to be used ensures that the resulting material will always be tied to that usage. The context structure defined in Section 11.2 is used by the KDF functions in this document.

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 is 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. For the AES algorithm versions, the extract step is always skipped.

The extract step cannot be skipped if the secret is not uniformly random, for example, if it is the result of an ECDH key agreement step. (This implies that the AES HKDF version cannot be used with ECDH.) If the extract step is skipped, the 'salt' value is not used as part of the HKDF functionality.

The algorithms defined in this document are found in Table 12.

name	PRF	description
HKDF SHA-256	HMAC with SHA-256	HKDF using HMAC SHA-256 as the PRF
HKDF SHA-512	HMAC with SHA-512	HKDF using HMAC SHA-512 as the PRF
HKDF AES-MAC-128	AES-CBC-MAC-128	HKDF using AES-MAC as the PRF w/ 128-bit key
HKDF AES-MAC-256	AES-CBC-MAC-256	HKDF using AES-MAC as the PRF w/ 256-bit key

Table 12: HKDF algorithms

name	label	type	algorithm	description
salt	-20	bstr	direct+HKDF-SHA-256, direct +HKDF-SHA-512, direct+HKDF-AES-128, direct+HKDF-AES-256, ECDH-ES+HKDF-256, ECDH-ES+HKDF-512, ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-ES+A128KW, ECDH-ES+A192KW, ECDH-ES+A256KW, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	Random salt

Table 13: 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]).

The context information structure refers to PartyU and PartyV as the two parties that 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.

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

o Fields can be defined by the application. This is commonly used to assign fixed names to parties, but can be used for other items such as nonces.

o Fields can be defined by usage of the output. Examples of this are the algorithm and key size that are being generated.

o Fields can be defined by parameters from the message. We define a set of parameters in Table 14 that 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.)

name	label	type	algorithm	description
PartyU identity	-21	bstr	direct+HKDF-SHA-256, direct+HKDF-SHA-512, direct+HKDF-AES-128, direct+HKDF-AES-256, ECDH-ES+HKDF-256, ECDH-ES+HKDF-512, ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-ES+A128KW, ECDH-ES+A192KW, ECDH-ES+A256KW, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	Party U identity Information
PartyU nonce	-22	bstr / int	direct+HKDF-SHA-256, direct+HKDF-SHA-512, direct+HKDF-AES-128, direct+HKDF-AES-256, ECDH-ES+HKDF-256, ECDH-ES+HKDF-512, ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-ES+A128KW, ECDH-ES+A192KW, ECDH-ES+A256KW, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	Party U provided nonce
PartyU other	-23	bstr	direct+HKDF-SHA-256, direct+HKDF-SHA-512, direct+HKDF-AES-128, direct+HKDF-AES-256, ECDH-ES+HKDF-256, ECDH-ES+HKDF-512, ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-	Party U other provided information

			ES+A128KW, ECDH-ES+A192KW, ECDH-ES+A256KW, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	
PartyV identity	-24	bstr	direct+HKDF-SHA-256, direct+HKDF-SHA-512, direct+HKDF-AES-128, direct+HKDF-AES-256, ECDH-ES+HKDF-256, ECDH-ES+HKDF-512, ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-ES+A128KW, ECDH-ES+A192KW, ECDH-ES+A256KW, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	Party V identity Information
PartyV nonce	-25	bstr / int	direct+HKDF-SHA-256, direct+HKDF-SHA-512, direct+HKDF-AES-128, direct+HKDF-AES-256, ECDH-ES+HKDF-256, ECDH-ES+HKDF-512, ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-ES+A128KW, ECDH-ES+A192KW, ECDH-ES+A256KW, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	Party V provided nonce
PartyV other	-26	bstr	direct+HKDF-SHA-256, direct+HKDF-SHA-512, direct+HKDF-AES-128, direct+HKDF-AES-256, ECDH-ES+HKDF-256, ECDH-ES+HKDF-512, ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-ES+A128KW, ECDH-ES+A192KW, ECDH-ES+A256KW, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	Party V other provided information

Table 14: Context Algorithm Parameters

We define a CBOR object to hold the context information. This object is referred to as COSE_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 normally is either a Key Wrap algorithm identifier or a Content Encryption algorithm identifier. The values are from the "COSE Algorithm Value" registry. This field is required to be present. 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 for each of those algorithms. (This practice means if algorithm A is broken and thus is easier to find, the key derived for algorithm B will not be the same as the key derived for algorithm A.)

PartyUInfo This field holds information about party U. The PartyUInfo is encoded as a CBOR array. The elements of PartyUInfo are encoded in the order presented, however if the element does not exist no element is placed in the array. The elements of the PartyUInfo 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 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 is set to nil in that case.

nonce This contains a 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 option does not need to be specified and is set to nil in that case

other This contains other information that is defined by the protocol.
This option does not need to be specified and is set to nil in that case

PartyVInfo This field holds information about party V. The content of the structure are the same as for the PartyUInfo but for party V.

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. If there are no elements in the protected field, then use a zero length bstr.

other This field 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 private information. An example of this information would be a pre-existing shared secret. (This could, for example, be used in combination with an ECDH key agreement to provide a secondary proof of identity.) 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.

The following CDDL fragment corresponds to the text above.

```
   PartyInfo = (
       identity : bstr / nil,
       nonce : bstr / int / nil,
       other : bstr / nil,
   )

   COSE_KDF_Context = [
       AlgorithmID : int / tstr,
       PartyUInfo : [ PartyInfo ],
       PartyVInfo : [ PartyInfo ],
       SuppPubInfo : [
           keyDataLength : uint,
           protected : empty_or_serialized_map,
           ? other : bstr
       ],
       ? SuppPrivInfo : bstr
   ]
```

12. Content Key Distribution Methods

Content key distribution methods (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 the 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 CEK. When direct encryption mode is used, it MUST be the only mode used on the message.

The COSE_Encrypt structure for the recipient is organized as follows:

- o The ’protected’ field MUST be a zero length item unless it is used in the computation of the content key.
- o The ’alg’ parameter MUST be present.
- o A parameter identifying the shared secret SHOULD be present.
- o The ’ciphertext’ field MUST be a zero length item.
- o 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 15.

When this algorithm is used, the protected field MUST be zero length. The key type MUST be 'Symmetric'.

name	value	description
direct	-6	Direct use of CEK

Table 15: Direct Key

12.1.1.1. Security Considerations

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

- o 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.

- o 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.

- o 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 CEK. The 'protected' field can be of non-zero length. 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 shared secret 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 for each message if the same key is used. The IV can be modified in a predictable manner, a random manner or an unpredictable manner (i.e., encrypting a counter).

The IV used for a key can also be generated from the same HKDF functionality as the key is generated. 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 16.

name	value	KDF	description
direct+HKDF-SHA-256	-10	HKDF SHA-256	Shared secret w/ HKDF and SHA-256
direct+HKDF-SHA-512	-11	HKDF SHA-512	Shared secret w/ HKDF and SHA-512
direct+HKDF-AES-128	-12	HKDF AES-MAC-128	Shared secret w/ AES-MAC 128-bit key
direct+HKDF-AES-256	-13	HKDF AES-MAC-256	Shared secret w/ AES-MAC 256-bit key

Table 16: Direct Key with KDF

When using a COSE key for this algorithm, the following checks are made:

o The 'kty' field MUST be present and it MUST be 'Symmetric'.

- o If the 'alg' field is present, it MUST match the algorithm being used.

- o If the 'key_ops' field is present, it MUST include 'deriveKey' or 'deriveBits'.

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.

While these methods do not provide for perfect forward secrecy, as the same shared secret is used for all of the keys generated, if the key for any single message is discovered only the message (or series of messages) using that derived key are compromised. A new key derivation step will generate a new key which requires the same amount of work to get the key.

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 that 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_Encrypt structure for the recipient is organized as follows:

- o The 'protected' field MUST be absent if the key wrap algorithm is an AE algorithm.

- o The 'recipients' field is normally absent, but can be used. Applications MUST deal with a recipient field being 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.

- o 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. The protected header field MUST be empty.

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

When using a COSE key for this algorithm, the following checks are made:

- The 'kty' field MUST be present and it MUST be 'Symmetric'.
- If the 'alg' field is present, it MUST match the AES Key Wrap algorithm being used.
- If the 'key_ops' field is present, it MUST include 'encrypt' or 'wrap key' when encrypting.
- If the 'key_ops' field is present, it MUST include 'decrypt' or 'unwrap key' when decrypting.

name	value	key size	description
A128KW	-3	128	AES Key Wrap w/ 128-bit key
A192KW	-4	192	AES Key Wrap w/ 192-bit key
A256KW	-5	256	AES Key Wrap w/ 256-bit key

Table 17: AES Key Wrap Algorithm Values

12.2.1.1. Security Considerations for AES-KW

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

12.3. Key Transport

Key transport mode is also called key encryption mode in some standards. Key transport 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 does not define any key transport mode algorithms.

When using a key transport algorithm, the COSE_Encrypt structure for the recipient is organized as follows:

- o The 'protected' field MUST be absent.
- o The plain text to be encrypted is the key from next layer down (usually the content layer).
- o 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 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 dynamic 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 object.

Two variants of DH that are 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 for the KDF is required to ensure that a different key is created for each message.

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_Encrypt structure for the recipient is organized as follows:

- At a minimum, headers MUST contain the 'alg' parameter and SHOULD contain a parameter identifying the recipient's asymmetric key.

- The headers SHOULD identify the sender's key for the static-static versions and MUST contain the sender's ephemeral key for the ephemeral-static versions.

12.4.1. ECDH

The mathematics for Elliptic Curve Diffie-Hellman can be found in [RFC6090]. In this document, the algorithm is extended to be used with the two curves defined in [RFC7748].

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 22. The math used to obtain the computed secret is based on the curve selected and not on the ECDH algorithm. For this reason, a new algorithm does not need to be defined for each of the curves.

- Computed Secret to Shared Secret: Once the computed secret is known, the resulting value needs to be converted to a byte string to run the KDF function. The X coordinate is used for all of the curves defined in this document. For curves X25519 and X448, the resulting value is used directly as it is a byte string of a known length. For the P-256, P-384 and P-521 curves, the X coordinate is run through the I2OSP function defined in [I-D.moriarty-pkcs1], using the same computation for n as is defined in Section 8.1.

o 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. For the KDF functions used, this means either in the 'salt' parameter for HKDF (Table 13) or in the 'PartyU nonce' parameter for the context structure (Table 14) MUST be present. (Both may be present if desired.) The value in the parameter MUST be unique for the pair of keys 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 should 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 19.

o 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 context material: how the key is going to be used, and one-time material for static-static key agreement. All of the algorithms defined in this document use one of the HKDF algorithms defined in Section 11.1 with the context structure defined in Section 11.2.

o Key Wrap algorithm: No key wrap algorithm is used. This is represented in Table 18 as 'none'. The key size for the context structure is the content layer encryption algorithm size.

The set of direct ECDH algorithms defined in this document are found in Table 18.

name	value	KDF	Ephemeral-Static	Key Wrap	description
ECDH-ES + HKDF-256	-25	HKDF - SHA-256	yes	none	ECDH ES w/ HKDF - generate key directly
ECDH-ES + HKDF-512	-26	HKDF - SHA-512	yes	none	ECDH ES w/ HKDF - generate key directly
ECDH-SS + HKDF-256	-27	HKDF - SHA-256	no	none	ECDH SS w/ HKDF - generate key directly
ECDH-SS + HKDF-512	-28	HKDF - SHA-512	no	none	ECDH SS w/ HKDF - generate key directly

Table 18: ECDH Algorithm Values

name	label	type	algorithm	description
ephemeral key	-1	COSE_Key	ECDH-ES+HKDF-256, ECDH-ES+HKDF-512, ECDH-ES+A128KW, ECDH-ES+A192KW, ECDH-ES+A256KW	Ephemeral Public key for the sender
static key	-2	COSE_Key	ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	Static Public key for the sender
static key id	-3	bstr	ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	Static Public key identifier for the sender

Table 19: ECDH Algorithm Parameters

This document defines these algorithms to be used with the curves P-256, P-384, P-521, X25519, and X448. 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.

When using a COSE key for this algorithm, the following checks are made:

o The 'kty' field MUST be present and it MUST be 'EC2' or 'OKP'.

o If the 'alg' field is present, it MUST match the Key Agreement algorithm being used.

o If the 'key_ops' field is present, it MUST include 'derive key' or 'derive bits' for the private key.

o If the 'key_ops' field is present, it MUST be empty for the public key.

12.4.2. Security Considerations

Some method of checking that points provided from external entities are valid. For the 'EC2' key format, this can be done by checking that the x and y values form a point on the curve. For the 'OKP' format, there is no simple way to do point validation.

Consideration was given to requiring that the public keys of both entities be provided as part of the key derivation process. (As recommended in section 6.1 of [RFC7748].) This was not done as COSE is used in a store and forward format rather than in on line key exchange. In order for this to be a problem, either the receiver public key has to be chosen maliciously or the sender has to be malicious. In either case, all security evaporates anyway.

A proof of possession of the private key associated with the public key is recommended when a key is moved from untrusted to trusted. (Either by the end user or by the entity that is responsible for making trust statements on keys.)

12.5. Key Agreement with Key Wrap

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 function for this would be:

```
encryptedKey = KeyWrap(KDF(DH-Shared, context), CEK)
```

The COSE_Encrypt 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 20.

ECDH with Key Agreement is parameterized by the same parameters as for ECDH Section 12.4.1 with the following modifications:

o Key Wrap Algorithm: Any of the key wrap algorithms defined in Section 12.2.1 are supported. The size of the key used for the key wrap algorithm is fed into the KDF function. The set of identifiers are found in Table 20.

name	value	KDF	Ephemeral-Static	Key Wrap	description
ECDH-ES + A128KW	-29	HKDF - SHA-256	yes	A128KW	ECDH ES w/ Concat KDF and AES Key wrap w/ 128 bit key
ECDH-ES + A192KW	-30	HKDF - SHA-256	yes	A192KW	ECDH ES w/ Concat KDF and AES Key wrap w/ 192 bit key
ECDH-ES + A256KW	-31	HKDF - SHA-256	yes	A256KW	ECDH ES w/ Concat KDF and AES Key wrap w/ 256 bit key
ECDH-SS + A128KW	-32	HKDF - SHA-256	no	A128KW	ECDH SS w/ Concat KDF and AES Key wrap w/ 128 bit key
ECDH-SS + A192KW	-33	HKDF - SHA-256	no	A192KW	ECDH SS w/ Concat KDF and AES Key wrap w/ 192 bit key
ECDH-SS + A256KW	-34	HKDF - SHA-256	no	A256KW	ECDH SS w/ Concat KDF and AES Key wrap w/ 256 bit key

Table 20: ECDH Algorithm Values with Key Wrap

When using a COSE key for this algorithm, the following checks are made:

- o The 'kty' field MUST be present and it MUST be 'EC2' or 'OKP'.

- o If the 'alg' field is present, it MUST match the Key Agreement algorithm being used.

- o If the 'key_ops' field is present, it MUST include 'derive key' or 'derive bits' for the private key.

- o If the 'key_ops' field is present, it MUST be empty for the public key.

13. Key Object Parameters

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. Private members allow for the archival of keys by individuals. However, there are some circumstances in which private keys may be distributed to 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:

name	value	description
OKP	1	Octet Key Pair
EC2	2	Elliptic Curve Keys w/ X,Y Coordinate pair
Symmetric	4	Symmetric Keys
Reserved	0	This value is reserved

Table 21: 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 ('EC2'). 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 ('OKP').

Applications MUST check that the curve and the key type are consistent and reject a key if they are not.

name	key type	value	description
P-256	EC2	1	NIST P-256 also known as secp256r1
P-384	EC2	2	NIST P-384 also known as secp384r1
P-521	EC2	3	NIST P-521 also known as secp521r1
X25519	OKP	4	X25519 for use w/ ECDH only
X448	OKP	5	X448 for use w/ ECDH only
Ed25519	OKP	6	Ed25519 for use w/ EdDSA only
Ed448	OKP	7	Ed448 for use w/ EdDSA only

Table 22: 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 23. 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 22. 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]. Leading zero octets MUST be preserved.

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. The compressed point encoding is also supported. Compute the sign bit as laid out in the Elliptic-Curve-Point-to-Octet-String Conversion function of [SEC1]. If the sign bit is zero, then encode y as a CBOR false value, otherwise encode y as a CBOR true value. The encoding of the infinity point is not supported.

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.

name	key type	label	type	description
crv	2	-1	int / tstr	EC Curve identifier - Taken from the COSE Curves registry
x	2	-2	bstr	X Coordinate
y	2	-3	bstr / bool	Y Coordinate
d	2	-4	bstr	Private key

Table 23: EC Key Parameters

13.2. Octet Key Pair

A new key type is defined for Octet Key Pairs (OKP). Do not assume that keys using this type are elliptic curves. This key type could be used for other curve types (for example, mathematics based on hyper-elliptic surfaces).

The key parameters defined in this section are summarized in Table 24. 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 22. 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 octet string represents a little-endian encoding of x.

d contains the private key.

For public keys, it is REQUIRED that 'crv' and 'x' 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' also be present, but it can be recomputed from the required elements and omitting it saves on space.

name	key type	label	type	description
crv	1	-1	int / tstr	EC Curve identifier - Taken from the COSE Key Common Parameters registry
x	1	-2	bstr	X Coordinate
d	1	-4	bstr	Private key

Table 24: Octet Key Pair Parameters

13.3. 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 does not have a form that contains only public members. As it is expected that this key structure is going to be transmitted, care must be taking that it is never transmitted accidentally or insecurely. For symmetric keys, it is REQUIRED that 'k' be present in the structure.

name	key type	label	type	description
k	4	-1	bstr	Key Value

Table 25: Symmetric Key Parameters

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

This document is designed to provide a set of security services, but not to provide implementation requirements for specific usage. The interoperability requirements are provided for how each of the individual services are used and how the algorithms are to be used for interoperability. The requirements about which algorithms and which services are needed are deferred to each application.

An example of a profile can be found in [I-D.selander-ace-object-security] where two profiles are being developed. One is for carrying content by itself, and the other is for carrying content in combination with CoAP headers.

It is intended that a profile of this document be created that defines the interoperability requirements for that specific application. This section provides a set of guidelines and topics that need to be considered when profiling this document.

- Applications need to determine the set of messages defined in this document that they will be using. The set of messages corresponds fairly directly to the set of security services that are needed and to the security levels needed.

- Applications may define new header parameters for a specific purpose. Applications will often times select specific header parameters to use or not to use. For example, an application would normally state a preference for using either the IV or the partial IV parameter. If the partial IV parameter is specified, then the application would also need to define how the fixed portion of the IV would be determined.

- When applications use externally defined authenticated data, they need to define how that data is encoded. This document assumes that the data will be provided as a byte stream. More information can be found in Section 4.3.

- Applications need to determine the set of security algorithms that are to be used. When selecting the algorithms to be used as the mandatory to implement set, consideration should be given to choosing different types of algorithms when two are chosen for a specific purpose. An example of this would be choosing HMAC-SHA512 and AES-CMAC as different MAC algorithms; the construction is vastly different between these two algorithms. This means that a weakening of one algorithm would be unlikely to lead to a weakening of the other algorithms. Of course, these algorithms do not provide the same level of security and thus may not be comparable for the desired security functionality.

- Applications may need to provide some type of negotiation or discovery method if multiple algorithms or message structures are permitted. The method can 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 that applications could follow:

  * 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 "CBOR Tags" registry. It is requested that the tags for COSE_Sign1, COSE_Encrypt0, and COSE_Mac0 be assigned in the 1 to 23 value range (one byte long when encoded). It is requested that the tags for COSE_Sign, COSE_Encrypt and COSE_MAC be assigned in the 24 to 255 value range (two bytes long when encoded).

The tags to be assigned are in Table 1.

16.2. COSE Header Parameters Registry

It is requested that IANA create a new registry entitled "COSE Header Parameters". The registry should be created as Expert Review Required. Guidelines for the experts is provided Section 16.11. It should be noted that in additional to the expert review, some portions of the registry require a specification, potentially on standards track, be supplied as well.

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 expert review. Integer values in the range -1 to -65536 are delegated to the "COSE Header Algorithm Parameters" registry. Integer values less than -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 2 and Table 27. 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 Parameters Registry

It is requested that IANA create a new registry entitled "COSE Header Algorithm Parameters". The registry is to be created as Expert Review Required. Expert review guidelines are provided in Section 16.11.

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 Values" 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.

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 13, Table 14, and Table 19. The specification column for all rows in that table should be this document.

16.4. COSE Algorithms Registry

It is requested that IANA create a new registry entitled "COSE Algorithms Registry". The registry is to be created as Expert Review Required. Guidelines for the experts is provided Section 16.11. It should be noted that in additional to the expert review, some portions of the registry require a specification, potentially on standards track, be supplied as well.

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 -256 and 255 and strings of length 1 are designated as Standards Track Document required. Integer values from -65536 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 expert review. Integer values less than -65536 are marked as private use.

description: A short description of the algorithm.

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

recommended: Does the IETF have a concensus recommendation to use the algorithm. The legal values are 'yes', 'no' and 'deprecated'.

The initial contents of the registry can be found in Table 10, Table 9, Table 11, Table 5, Table 7, Table 8, Table 15, Table 16, Table 17, Table 6, Table 20 and Table 18. The specification column for all rows in the table should be this document. The recommneded column for all rows in the table are set to 'yes'.

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

NOTE: The assignment of algorithm identifiers in this document was done so that positive numbers were used for the first layer objects (COSE_Sign, COSE_Sign1, COSE_Encrypt, COSE_Encrypt0, COSE_Mac, and COSE_Mac0). Negative numbers were used for second layer objects (COSE_Signature and COSE_recipient). Expert reviewers should consider this practice, but are not expected to be restricted by this precedent.

16.5. COSE Key Common Parameters Registry

It is requested that IANA create a new registry entitled "COSE Key Common Parameters" registry. The registry is to be created as Expert Review Required. Guidelines for the experts is provided Section 16.11. It should be noted that in additional to the expert review, some portions of the registry require a specification, potentially on standards track, be supplied as well.

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 expert review. Integer values in the range -1 to -65536 are used for key parameters specific to a single algorithm delegated to the "COSE Key Type Parameter Labels" registry. Integer values less than -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 Table 3. The specification column for all of these entries will be this document.

16.6. COSE Key Type Parameters Registry

It is requested that IANA create a new registry "COSE Key Type Parameters". The registry is to be created as Expert Review Required. Expert review guidelines are provided in Section 16.11.

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 Key Common Parameters 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 23, Table 24, and Table 25. The specification column for all of these entries will be this document.

16.7. COSE Key Type Registry

It is requested that IANA create a new registry "COSE Key Type Registry". The registry is to be created as Expert Review Required. Expert review guidelines are provided in Section 16.11.

The columns of this table are:

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

value This is the value used to identify the curve. These values MUST be unique. The value can be a positive integer, a negative integer or a string.

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 21. The specification column for all of these entries will be this document.

16.8. COSE Elliptic Curve Parameters Registry

It is requested that IANA create a new registry "COSE Elliptic Curve Parameters". The registry is to be created as Expert Review Required. Guidelines for the experts is provided Section 16.11. It should be noted that in additional to the expert review, some portions of the registry require a specification, potentially on standards track, be supplied as well.

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 expert review. 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.

recommended: Does the IETF have a concensus recommendation to use the algorithm. The legal values are 'yes', 'no' and 'deprecated'.

This registry will be initially populated by the values in Table 22. The specification column for all of these entries will be this document. The recommended column for all of the inital entries will be 'yes'.

16.9. Media Type Registrations

16.9.1. COSE Security Message

This section registers the "application/cose" media type in the "Media Types" registry. These media types are used to indicate that the content is a COSE message.

Type name: application

Subtype name: cose

Required parameters: N/A

Optional parameters: cose-type

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: IoT applications sending security content over HTTP(S) transports.

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

16.9.2. COSE Key media type

This section registers the "application/cose-key" and "application/cose-key-set" 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.

The template for registering "application/cose-key" is:

Type name: application

Subtype name: cose-key

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: Distribution of COSE based keys for IoT applications.

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

The template for registering "application/cose-key-set" is:

Type name: application

Subtype name: cose-key-set

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: Distribution of COSE based keys for IoT applications.

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

16.10. CoAP Content-Format Registrations

IANA is requested to add the following entries to the "CoAP Content-Format" registry. ID assignment in the 24-255 range is requested.

Media Type	Encoding	ID	Reference
application/cose; cose-type ="cose-sign"		TBD10	[This Document]
application/cose; cose-type ="cose-sign1"		TBD11	[This Document]
application/cose; cose-type ="cose-encrypt"		TBD12	[This Document]
application/cose; cose-type ="cose-encrypt0"		TBD13	[This Document]
application/cose; cose-type ="cose-mac"		TBD14	[This Document]
application/cose; cose-type ="cose-mac0"		TBD15	[This Document]
application/cose-key		TBD16	[This Document]
application/cose-key-set		TBD17	[This Document

Table 26

16.11. Expert Review Instructions

All of the IANA registries established in this document are defined as expert review. This section gives some general guidelines for what the experts should be looking for, but they are being designated as experts for a reason so they should be given substantial latitude.

Expert reviewers should take into consideration the following points:

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

- Specifications are required for the standards track range of point assignment. Specifications should exist for specification required ranges, but early assignment before a specification is available is considered to be permissible. Specifications are needed for the first-come, first-serve range if they are expected to be used outside of closed environments in an interoperable way. When specifications are not provided, the description provided needs to have sufficient information to identify what the point is being used for.

- Experts should take into account the expected usage of fields when approving point assignment. The fact that there is a range for standards track documents does not mean that a standards track document cannot have points assigned outside of that range. The length of the encoded value should be weighed against how many code points of that length are left, the size of device it will be used on, and the number of code points left that encode to that size.

- When algorithms are registered, vanity registrations should be discouraged. One way to do this is to require registrations to provide additional documentation on security analysis of the algorithm. Another thing that should be considered is to request for an opinion on the algorithm from the Crypto Forum Research Group (CFRG). Algorithms that do not meet the security requirements of the community and the messages structures should not be registered.

17. Implementation Status

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations may exist.

According to [RFC7942], "this will allow reviewers and working groups to assign due consideration to documents that have the benefit of running code, which may serve as evidence of valuable experimentation and feedback that have made the implemented protocols more mature.

It is up to the individual working groups to use this information as they see fit".

17.1. Author’s Versions

There are three different implementations that have been created by the author of the document both to create the examples that are included in the document and to validate the structures and methodology used in the design of COSE.

Implementation Location: https://github.com/cose-wg

Primary Maintainer: Jim Schaad

Languages: There are three different languages that are currently supported: Java, C# and C.

Cryptography: The Java and C# libraries use Bouncy Castle to provide the required cryptography. The C version uses OPENSSL Version 1.0 for the cryptography.

Coverage: The libraries currently do not have full support for counter signatures of either variety. They do have support to allow for implicit algorithm support as they allow for the application to set attributes that are not to be sent in the message.

Testing: All of the examples in the example library are generated by the C# library and then validated using the Java and C libraries. All three libraries have tests to allow for the creating of the same messages that are in the example library followed by validating them. These are not compared against the example library. The Java and C# libraries have unit testing included. Not all of the MUST statements in the document have been implemented as part of the libraries. One such statement is the requirement that unique labels be present.

Licensing: Revised BSD License

17.2. COSE Testing Library

Implementation Location: https://github.com/cose-wg/Examples

Primary Maintainer: Jim Schaad

Description: A set of tests for the COSE library is provided as part of the implementation effort. Both success and fail tests have been provided. All of the examples in this document are part of this example set.

Coverage: An attempt has been made to have test cases for every message type and algorithm in the document. Currently examples dealing with counter signatures, EdDSA, and ECDH with Curve24459 and Goldilocks are missing.

Licensing: Public Domain

18. Security Considerations

There are a number of security considerations that need to be taken into account by implementers of this specification. The security considerations that are specific to an individual algorithm are placed next to the description of the algorithm. While some considerations have been highlighted here, additional considerations may be found in the documents listed in the references.

Implementations need to protect the private key material for any individuals. There are some cases in this document that need to be highlighted on this issue.

- o Using the same key for two different algorithms can leak information about the key. It is therefore recommended that keys be restricted to a single algorithm.

- o Use of 'direct' as a recipient algorithm combined with a second recipient algorithm, exposes the direct key to the second recipient.

- o Several of the algorithms in this document have limits on the number of times that a key can be used without leaking information about the key.

The use of ECDH and direct plus KDF (with no key wrap) will not directly lead to the private key being leaked; the one way function of the KDF will prevent that. There is however, a different issue that needs to be addressed. Having two recipients requires that the CEK be shared between two recipients. The second recipient therefore has a CEK that was derived from material that can be used for the weak proof of origin. The second recipient could create a message using the same CEK and send it to the first recipient, the first recipient would, for either static-static ECDH or direct plus KDF, make an assumption that the CEK could be used for proof of origin even though it is from the wrong entity. If the key wrap step is added, then no proof of origin is implied and this is not an issue.

Although it has been mentioned before, the use of a single key for multiple algorithms has been demonstrated in some cases to leak information about a key, provide for attackers to forge integrity tags, or gain information about encrypted content. Binding a key to a single algorithm prevents these problems. Key creators and key consumers are strongly encouraged not only to create new keys for each different algorithm, but to include that selection of algorithm in any distribution of key material and strictly enforce the matching of algorithms in the key structure to algorithms in the message structure. In addition to checking that algorithms are correct, the key form needs to be checked as well. Do not use an 'EC2' key where an 'OKP' key is expected.

Before using a key for transmission, or before acting on information received, a trust decision on a key needs to be made. Is the data or action something that the entity associated with the key has a right to see or a right to request? A number of factors are associated with this trust decision. Some of the ones that are highlighted here are:

- What are the permissions associated with the key owner?

- Is the cryptographic algorithm acceptable in the current context?

- Have the restrictions associated with the key, such as algorithm or freshness, been checked and are correct?

- Is the request something that is reasonable, given the current state of the application?

- Have any security considerations that are part of the message been enforced (as specified by the application or 'crit' parameter)?

There are a large number of algorithms presented in this document that use nonce values. For all of the nonces defined in this document, there is some type of restriction on the nonce being a unique value either for a key or for some other conditions. In all of these cases, there is no known requirement on the nonce being both unique and unpredictable, under these circumstances it reasonable to use a counter for creation of the nonce. In cases where one wants the pattern of the nonce to be unpredictable as well as unique, one can use a key created for that purpose and encrypt the counter to produce the nonce value.

One area that has been starting to get exposure is doing traffic analysis of encrypted messages based on the length of the message. This specification does not provide for a uniform method of providing padding as part of the message structure. An observer can distinguish between two different strings (for example, 'YES' and 'NO') based on length for all of the content encryption algorithms that are defined in this document. This means that it is up to applications to document how content padding is to be done in order to prevent or discourage such analysis. (For example, the strings could be defined as 'YES' and 'NO '.)

19. References

19.1. Normative References

[AES-GCM] Dworkin, M., "NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC.", Nov 2007.

[COAP.Formats] IANA, , "CoAP Content-Formats".

[DSS] U.S. National Institute of Standards and Technology, "Digital Signature Standard (DSS)", July 2013.

[I-D.irtf-cfrg-eddsa] Josefsson, S. and I. Liusvaara, "Edwards-curve Digital Signature Algorithm (EdDSA)", draft-irtf-cfrg-eddsa-08 (work in progress), August 2016.

[MAC] NiST, N., "FIPS PUB 113: Computer Data Authentication", May 1985.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <http://www.rfc-editor.org/info/rfc2104>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

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

[RFC3610] Whiting, D., Housley, R., and N. Ferguson, "Counter with CBC-MAC (CCM)", RFC 3610, DOI 10.17487/RFC3610, September 2003, <http://www.rfc-editor.org/info/rfc3610>.

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

[RFC6090] McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms", RFC 6090, DOI 10.17487/RFC6090, February 2011, <http://www.rfc-editor.org/info/rfc6090>.

[RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <http://www.rfc-editor.org/info/rfc6979>.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <http://www.rfc-editor.org/info/rfc7049>.

[RFC7539] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 7539, DOI 10.17487/RFC7539, May 2015, <http://www.rfc-editor.org/info/rfc7539>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <http://www.rfc-editor.org/info/rfc7748>.

[SEC1] Standards for Efficient Cryptography Group, "SEC 1: Elliptic Curve Cryptography", May 2009.

19.2. Informative References

[I-D.greevenbosch-appsawg-cbor-cddl]
Vigano, C. and H. Birkholz, "CBOR data definition language (CDDL): a notational convention to express CBOR data structures", draft-greevenbosch-appsawg-cbor-cddl-09 (work in progress), September 2016.

[I-D.moriarty-pkcs1]
Moriarty, K., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1 Version 2.2: RSA Cryptography Specifications", draft-moriarty-pkcs1-03 (work in progress), September 2016.

[I-D.moriarty-pkcs5-v2dot1]
Moriarty, K., Kaliski, B., and A. Rusch, "PKCS #5: Password-Based Cryptography Specification Version 2.1", draft-moriarty-pkcs5-v2dot1-04 (work in progress), September 2016.

[I-D.selander-ace-object-security]
Selander, G., Mattsson, J., Palombini, F., and L. Seitz, "Object Security of CoAP (OSCOAP)", draft-selander-ace-object-security-06 (work in progress), October 2016.

[PVSig] Brown, D. and D. Johnson, "Formal Security Proofs for a Signature Scheme with Partial Message Recover", February 2000.

[RFC2633] Ramsdell, B., Ed., "S/MIME Version 3 Message Specification", RFC 2633, DOI 10.17487/RFC2633, June 1999, <http://www.rfc-editor.org/info/rfc2633>.

[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, <http://www.rfc-editor.org/info/rfc4231>.

[RFC4262] Santesson, S., "X.509 Certificate Extension for Secure/Multipurpose Internet Mail Extensions (S/MIME) Capabilities", RFC 4262, DOI 10.17487/RFC4262, December 2005, <http://www.rfc-editor.org/info/rfc4262>.

[RFC4493] Song, JH., Poovendran, R., Lee, J., and T. Iwata, "The AES-CMAC Algorithm", RFC 4493, DOI 10.17487/RFC4493, June 2006, <http://www.rfc-editor.org/info/rfc4493>.

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007, <http://www.rfc-editor.org/info/rfc4949>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <http://www.rfc-editor.org/info/rfc5116>.

[RFC5480] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Elliptic Curve Cryptography Subject Public Key Information", RFC 5480, DOI 10.17487/RFC5480, March 2009, <http://www.rfc-editor.org/info/rfc5480>.

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

[RFC5751] Ramsdell, B. and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification", RFC 5751, DOI 10.17487/RFC5751, January 2010, <http://www.rfc-editor.org/info/rfc5751>.

[RFC5752] Turner, S. and J. Schaad, "Multiple Signatures in Cryptographic Message Syntax (CMS)", RFC 5752, DOI 10.17487/RFC5752, January 2010, <http://www.rfc-editor.org/info/rfc5752>.

[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, <http://www.rfc-editor.org/info/rfc5990>.

[RFC6151] Turner, S. and L. Chen, "Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms", RFC 6151, DOI 10.17487/RFC6151, March 2011, <http://www.rfc-editor.org/info/rfc6151>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <http://www.rfc-editor.org/info/rfc6838>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <http://www.rfc-editor.org/info/rfc7159>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <http://www.rfc-editor.org/info/rfc7252>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <http://www.rfc-editor.org/info/rfc7515>.

[RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <http://www.rfc-editor.org/info/rfc7516>.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <http://www.rfc-editor.org/info/rfc7517>.

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <http://www.rfc-editor.org/info/rfc7518>.

[RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <http://www.rfc-editor.org/info/rfc7942>.

[SP800-56A] Barker, E., Chen, L., Roginsky, A., and M. Smid, "NIST Special Publication 800-56A: Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", May 2013.

[W3C.WebCrypto] Watson, M., "Web Cryptography API", July 2016.

Appendix A. Guidelines for External Data Authentication of Algorithms

There has been a portion of the working group who have expressed a strong desire to relax the rule that the algorithm identifier be required to appear in each level of a COSE object. There are two basic reasons that have been advanced to support this position. First, the resulting message will be smaller if the algorithm identifier is omitted from the most common messages in a CoAP environment. Second, there is a potential bug that will arise if full checking is not done correctly between the different places that an algorithm identifier could be placed (the message itself, an application statement, the key structure that the sender possesses and the key structure the recipient possesses).

This appendix lays out how such a change can be made and the details that an application needs to specify in order to use this option. Two different sets of details are specified: Those needed to omit an algorithm identifier and those needed to use a variant on the counter signature attribute that contains no attributes about itself.

A.1. Algorithm Identification

In this section are laid out three sets of recommendations. The first set of recommendations apply to having an implicit algorithm identified for a single layer of a COSE object. The second set of recommendations apply to having multiple implicit algorithms identified for multiple layers of a COSE object. The third set of recommendations apply to having implicit algorithms for multiple COSE object constructs.

RFC 2119 language is deliberately not used here. This specification can provide recommendations, but it cannot enforce them.

This set of recommendations applies to the case where an application is distributing a fixed algorithm along with the key information for use in a single COSE object. This normally applies to the smallest of the COSE objects, specifically COSE_Sign1, COSE_Mac0, and COSE_Encrypt0, but could apply to the other structures as well.

The following items should be taken into account:

- o Applications need to list the set of COSE structures that implicit algorithms are to be used in. Applications need to require that the receipt of an explicit algorithm identifier in one of these structures will lead to the message being rejected. This requirement is stated so that there will never be a case where there is any ambiguity about the question of which algorithm should be used, the implicit or the explicit one. This applies even if the transported algorithm identifier is a protected attribute. This applies even if the transported algorithm is the same as the implicit algorithm.

- o Applications need to define the set of information that is to be considered to be part of a context when omitting algorithm identifiers. At a minimum, this would be the key identifier (if needed), the key, the algorithm, and the COSE structure it is used with. Applications should restrict the use of a single key to a single algorithm. As noted for some of the algorithms in this document, the use of the same key in different related algorithms can lead to leakage of information about the key, leakage about the data or the ability to perform forgeries.

- o In many cases, applications that make the algorithm identifier implicit will also want to make the context identifier implicit for the same reason. That is, omitting the context identifier will decrease the message size (potentially significantly depending on the length of the identifier). Applications that do this will need to describe the circumstances where the context identifier is to be omitted and how the context identifier is to be inferred in these cases. (Exhaustive search over all of the keys would normally not be considered to be acceptable.) An example of how this can be done is to tie the context to a transaction identifier. Both would be sent on the original message, but only the transaction identifier would need to be sent after that point as the context is tied into the transaction identifier. Another way would be to associate a context with a network address. All messages coming from a single network address can be assumed to be associated with a specific context. (In this case the address would normally be distributed as part of the context.)

- Applications cannot rely on key identifiers being unique unless they take significant efforts to ensure that they are computed in such a way as to create this guarantee. Even when an application does this, the uniqueness might be violated if the application is run in different contexts (i.e., with a different context provider) or if the system combines the security contexts from different applications together into a single store.

- Applications should continue the practice of protecting the algorithm identifier. Since this is not done by placing it in the protected attributes field, applications should define an application specific external data structure that includes this value. This external data field can be used as such for content encryption, MAC, and signature algorithms. It can be used in the SuppPrivInfo field for those algorithms which use a KDF function to derive a key value. Applications may also want to protect other information that is part of the context structure as well. It should be noted that those fields, such as the key or a base IV, are protected by virtue of being used in the cryptographic computation and do not need to be included in the external data field.

The second case is having multiple implicit algorithm identifiers specified for a multiple layer COSE object. An example of how this would work is the encryption context that an application specifies contains a content encryption algorithm, a key wrap algorithm, a key identifier, and a shared secret. The sender omits sending the algorithm identifier for both the content layer and the recipient layer leaving only the key identifier. The receiver then uses the key identifier to get the implicit algorithm identifiers.

The following additional items need to be taken into consideration:

- Applications that want to support this will need to define a structure that allows for, and clearly identifies, both the COSE structure to be used with a given key and the structure and algorithm to be used for the secondary layer. The key for the secondary layer is computed per normal from the recipient layer.

The third case is having multiple implicit algorithm identifiers, but targeted at potentially unrelated layers or different COSE objects.

There are a number of different scenarios where this might be applicable. Some of these scenarios are:

- o Two contexts are distributed as a pair. Each of the contexts is for use with a COSE_Encrypt message. Each context will consist of distinct secret keys and IVs and potentially even different algorithms. One context is for sending messages from party A to party B, the second context is for sending messages from party B to party A. This means that there is no chance for a reflection attack to occur as each party uses different secret keys to send its messages, a message that is reflected back to it would fail to decrypt.

- o Two contexts are distributed as a pair. The first context is used for encryption of the message; the second context is used to place a counter signature on the message. The intention is that the second context can be distributed to other entities independently of the first context. This allows these entities to validate that the message came from an individual without being able to decrypt the message and see the content.

- o Two contexts are distributed as a pair. The first context contains a key for dealing with MACed messages, the second context contains a key for dealing with encrypted messages. This allows for a unified distribution of keys to participants for different types of messages that have different keys, but where the keys may be used in coordinated manner.

For these cases, the following additional items need to be considered:

- o Applications need to ensure that the multiple contexts stay associated. If one of the contexts is invalidated for any reason, all of the contexts associated with it should also be invalidated.

A.2. Counter Signature Without Headers

There is a group of people who want to have a counter signature parameter that is directly tied to the value being signed and thus the authenticated and unauthenticated buckets can be removed from the message being sent. The focus on this is an even smaller size, as all of the information on the process of creating the counter signature is implicit rather than being explicitly carried in the message. This includes not only the algorithm identifier as presented above, but also items such as the key identification is always external to the signature structure. This means that the entities that are doing the validation of the counter signature are required to infer which key is to be used from context rather than being explicit. One way of doing this would be to presume that all data coming from a specific port (or to a specific URL) is to be validated by a specific key. (Note that this does not require that the key identifier be part of the value signed as it does not serve a cryptographic purpose. If the key validates the counter signature, then it should be presumed that the entity associated with that key produced the signature.)

When computing the signature for the bare counter signature header, the same Sig_structure defined in Section 4.4 is used. The sign_protected field is omitted, as there is no protected header field in in this counter signature header. The value of "CounterSignature0" is placed in the context field of the Sig_stucture.

name	label	value type	value	description
CounterSignature0	9	bstr		Counter signature with implied signer and headers

Table 27

Appendix B. Two Layers of Recipient Information

All of the currently defined recipient algorithms classes only use two layers of the COSE_Encrypt structure. The first layer is the message content and the second layer 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 makes sense to have three layers of the COSE_Encrypt structure.

These layers would be:

o Layer 0: The content encryption layer. This layer contains the payload of the message.

o Layer 1: The encryption of the CEK by a KEK.

o Layer 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:

- Layer 0: Has a content encrypted with AES-GCM using a 128-bit key.
- Layer 1: Uses the AES Key wrap algorithm with a 128-bit key.
- Layer 2: Uses ECDH Ephemeral-Static direct to generate the layer 1 key.

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

Size of binary file is 183 bytes

```
   96(
     [
       / protected / h'a10101' / {
           \ alg \ 1:1 \ AES-GCM 128 \
         } / ,
       / unprotected / {
         / iv / 5:h'02d1f7e6f26c43d4868d87ce'
       },
       / ciphertext / h'64f84d913ba60a76070a9a48f26e97e863e2852948658f0
   811139868826e89218a75715b',
       / recipients / [
         [
           / protected / h'',
           / unprotected / {
             / alg / 1:-3 / A128KW /
           },
           / ciphertext / h'dbd43c4e9d719c27c6275c67d628d493f090593db82
   18f11',
           / recipients / [
             [
               / protected / h'a1013818' / {
                   \ alg \ 1:-25 \ ECDH-ES + HKDF-256 \
                 } / ,
               / unprotected / {
                 / ephemeral / -1:{
                   / kty / 1:2,
                   / crv / -1:1,
                   / x / -2:h'b2add44368ea6d641f9ca9af308b4079aeb519f11
   e9b8a55a600b21233e86e68',
                   / y / -3:false
                 },
                 / kid / 4:'meriadoc.brandybuck@buckland.example'
               },
               / ciphertext / h''
             ]
           ]
         ]
       ]
     ]
   )
```

Appendix C. Examples

This appendix includes a set of examples that show the different features and message types that have been defined in this document. To make the examples easier to read, they are presented using the extended CBOR diagnostic notation (defined in [I-D.greevenbosch-appsawg-cbor-cddl]) rather than as a binary dump.

A GitHub project has been created at https://github.com/cose-wg/Examples that contains not only the examples presented in this document, but a more complete set of testing examples as well. Each example is found in a JSON file that contains the inputs used to create the example, some of the intermediate values that can be used in debugging the example and the output of the example presented in both a hex and a CBOR diagnostic notation format. Some of the examples at the site are designed failure testing cases; these are clearly marked as such in the JSON file. If errors in the examples in this document are found, the examples on github will be updated and a note to that effect will be placed in the JSON file.

As noted, the examples are presented using the CBOR's diagnostic notation. A Ruby based tool exists that can convert between the diagnostic notation and binary. 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.rb < 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'. (Depending on the XPath evaluator one is using, it may be necessary to deal with > as an entity.)

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

C.1. Examples of Signed Message

C.1.1. Single Signature

This example uses the following:

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

Size of binary file is 103 bytes

```
98(
  [
    / protected / h'',
    / unprotected / {},
    / payload / 'This is the content.',
    / signatures / [
      [
        / protected / h'a10126' / {
            \ alg \ 1:-7 \ ECDSA 256 \
          } / ,
        / unprotected / {
          / kid / 4:'11'
        },
        / signature / h'e2aeafd40d69d19dfe6e52077c5d7ff4e408282cbefb
5d06cbf414af2e19d982ac45ac98b8544c908b4507de1e90b717c3d34816fe926a2b
98f53afd2fa0f30a'
      ]
    ]
  ]
)
```

C.1.2. Multiple Signers

This example uses the following:

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

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

Size of binary file is 277 bytes

```
   98(
     [
       / protected / h'',
       / unprotected / {},
       / payload / 'This is the content.',
       / signatures / [
         [
           / protected / h'a10126' / {
               \ alg \ 1:-7 \ ECDSA 256 \
             } / ,
           / unprotected / {
             / kid / 4:'11'
           },
           / signature / h'e2aeafd40d69d19dfe6e52077c5d7ff4e408282cbefb
   5d06cbf414af2e19d982ac45ac98b8544c908b4507de1e90b717c3d34816fe926a2b
   98f53afd2fa0f30a'
         ],
         [
           / protected / h'a1013823' / {
               \ alg \ 1:-36
             } / ,
           / unprotected / {
             / kid / 4:'bilbo.baggins@hobbiton.example'
           },
           / signature / h'00a2d28a7c2bdb1587877420f65adf7d0b9a06635dd1
   de64bb62974c863f0b160dd2163734034e6ac003b01e8705524c5c4ca479a952f024
   7ee8cb0b4fb7397ba08d009e0c8bf482270cc5771aa143966e5a469a09f613488030
   c5b07ec6d722e3835adb5b2d8c44e95ffb13877dd2582866883535de3bb03d01753f
   83ab87bb4f7a0297'
         ]
       ]
     ]
   )
```

C.1.3. Counter Signature

This example uses the following:

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

- o The same parameters are used for both the signature and the counter signature.

Size of binary file is 180 bytes

```
   98(
     [
       / protected / h'',
       / unprotected / {
         / countersign / 7:[
           / protected / h'a10126' / {
               \ alg \ 1:-7 \ ECDSA 256 \
             } / ,
           / unprotected / {
             / kid / 4:'11'
           },
           / signature / h'5ac05e289d5d0e1b0a7f048a5d2b643813ded50bc9e4
   9220f4f7278f85f19d4a77d655c9d3b51e805a74b099e1e085aacd97fc29d72f887e
   8802bb6650cceb2c'
         ]
       },
       / payload / 'This is the content.',
       / signatures / [
         [
           / protected / h'a10126' / {
               \ alg \ 1:-7 \ ECDSA 256 \
             } / ,
           / unprotected / {
             / kid / 4:'11'
           },
           / signature / h'e2aeafd40d69d19dfe6e52077c5d7ff4e408282cbefb
   5d06cbf414af2e19d982ac45ac98b8544c908b4507de1e90b717c3d34816fe926a2b
   98f53afd2fa0f30a'
         ]
       ]
     ]
   )
```

C.1.4. Signature w/ Criticality

This example uses the following:

- o Signature Algorithm: ECDSA w/ SHA-256, Curve P-256
- o There is a criticality marker on the "reserved" header parameter

Size of binary file is 125 bytes

```
   98(
     [
       / protected / h'a2687265736572766564f40281687265736572766564' /
   {
           "reserved":false,
           \ crit \ 2:[
             "reserved"
           ]
         } / ,
       / unprotected / {},
       / payload / 'This is the content.',
       / signatures / [
         [
           / protected / h'a10126' / {
               \ alg \ 1:-7 \ ECDSA 256 \
             } / ,
           / unprotected / {
             / kid / 4:'11'
           },
           / signature / h'3fc54702aa56e1b2cb20284294c9106a63f91bac658d
   69351210a031d8fc7c5ff3e4be39445b1a3e83e1510d1aca2f2e8a7c081c7645042b
   18aba9d1fad1bd9c'
         ]
       ]
     ]
   )
```

C.2. Single Signer Examples

C.2.1. Single ECDSA signature

This example uses the following:

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

Size of binary file is 98 bytes

```
18(
  [
    / protected / h'a10126' / {
        \ alg \ 1:-7 \ ECDSA 256 \
      } / ,
    / unprotected / {
      / kid / 4:'11'
    },
    / payload / 'This is the content.',
    / signature / h'eae868ecc176883766c5dc5ba5b8dca25dab3c2e56a551ce
5705b793914348e19f43d6c6ba654472da301b645b293c9ba939295b97c4bdb84778
2bff384c5794'
  ]
)
```

C.3. Examples of Enveloped Messages

C.3.1. 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 151 bytes

```
   96(
     [
       / protected / h'a10101' / {
           \ alg \ 1:1 \ AES-GCM 128 \
         } / ,
       / unprotected / {
         / iv / 5:h'c9cf4df2fe6c632bf7886413'
       },
       / ciphertext / h'7adbe2709ca818fb415f1e5df66f4e1a51053ba6d65a1a0
   c52a357da7a644b8070a151b0',
       / recipients / [
         [
           / protected / h'a1013818' / {
               \ alg \ 1:-25 \ ECDH-ES + HKDF-256 \
             } / ,
           / unprotected / {
             / ephemeral / -1:{
               / kty / 1:2,
               / crv / -1:1,
               / x / -2:h'98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbf
   bf054e1c7b4d91d6280',
               / y / -3:true
             },
             / kid / 4:'meriadoc.brandybuck@buckland.example'
           },
           / ciphertext / h''
         ]
       ]
     ]
   )
```

C.3.2. Direct plus Key Derivation

This example uses the following:

- CEK: AES-CCM w/128-bit key, truncate the tag to 64 bits
- Recipient class: Use HKDF on a shared secret with the following implicit fields as part of the context.
    - salt: "aabbccddeeffgghh"
    - APU identity: "lighting-client"
    - APV identity: "lighting-server"
    - Supplementary Public Other: "Encryption Example 02"

Size of binary file is 91 bytes

```
96(
  [
    / protected / h'a1010a' / {
        \ alg \ 1:10 \ AES-CCM-16-64-128 \
      } / ,
    / unprotected / {
      / iv / 5:h'89f52f65a1c580933b5261a76c'
    },
    / ciphertext / h'753548a19b1307084ca7b2056924ed95f2e3b17006dfe93
1b687b847',
    / recipients / [
      [
        / protected / h'a10129' / {
            \ alg \ 1:-10
          } / ,
        / unprotected / {
          / salt / -20:'aabbccddeeffgghh',
          / kid / 4:'our-secret'
        },
        / ciphertext / h''
      ]
    ]
  ]
)
```

C.3.3. Counter Signature on Encrypted Content

This example uses the following:

- o CEK: AES-GCM w/ 128-bit key

- o Recipient class: ECDH Ephemeral-Static, Curve P-256

Size of binary file is 326 bytes

```
   96(
     [
       / protected / h'a10101' / {
           \ alg \ 1:1 \ AES-GCM 128 \
         } / ,
       / unprotected / {
         / iv / 5:h'c9cf4df2fe6c632bf7886413',
         / countersign / 7:[
           / protected / h'a1013823' / {
               \ alg \ 1:-36
             } / ,
           / unprotected / {
             / kid / 4:'bilbo.baggins@hobbiton.example'
           },
           / signature / h'00929663c8789bb28177ae28467e66377da12302d7f9
   594d2999afa5dfa531294f8896f2b6cdf1740014f4c7f1a358e3a6cf57f4ed6fb02f
   cf8f7aa989f5dfd07f0700a3a7d8f3c604ba70fa9411bd10c2591b483e1d2c31de00
   3183e434d8fba18f17a4c7e3dfa003ac1cf3d30d44d2533c4989d3ac38c38b71481c
   c3430c9d65e7ddff'
         ]
       },
       / ciphertext / h'7adbe2709ca818fb415f1e5df66f4e1a51053ba6d65a1a0
   c52a357da7a644b8070a151b0',
       / recipients / [
         [
           / protected / h'a1013818' / {
               \ alg \ 1:-25 \ ECDH-ES + HKDF-256 \
             } / ,
           / unprotected / {
             / ephemeral / -1:{
               / kty / 1:2,
               / crv / -1:1,
               / x / -2:h'98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbf
   bf054e1c7b4d91d6280',
               / y / -3:true
             },
             / kid / 4:'meriadoc.brandybuck@buckland.example'
           },
           / ciphertext / h''
         ]
       ]
     ]
   )
```

C.3.4. Encrypted Content with External Data

This example uses the following:

- o CEK: AES-GCM w/ 128-bit key
- o Recipient class: ECDH static-Static, Curve P-256 with AES Key Wrap
- o Externally Supplied AAD: h'0011bbcc22dd44ee55ff660077'

Size of binary file is 173 bytes

```
96(
  [
    / protected / h'a10101' / {
        \ alg \ 1:1 \ AES-GCM 128 \
      } / ,
    / unprotected / {
      / iv / 5:h'02d1f7e6f26c43d4868d87ce'
    },
    / ciphertext / h'64f84d913ba60a76070a9a48f26e97e863e28529d8f5335
e5f0165eee976b4a5f6c6f09d',
    / recipients / [
      [
        / protected / h'a101381f' / {
            \ alg \ 1:-32 \ ECHD-SS+A128KW \
          } / ,
        / unprotected / {
          / static kid / -3:'peregrin.took@tuckborough.example',
          / kid / 4:'meriadoc.brandybuck@buckland.example',
          / U nonce / -22:h'0101'
        },
        / ciphertext / h'41e0d76f579dbd0d936a662d54d8582037de2e366fd
e1c62'
      ]
    ]
  ]
)
```

C.4. Examples of Encrypted Messages

C.4.1. Simple Encrypted Message

This example uses the following:

- o CEK: AES-CCM w/ 128-bit key and a 64-bit tag

Size of binary file is 52 bytes

```
16(
  [
    / protected / h'a1010a' / {
        \ alg \ 1:10 \ AES-CCM-16-64-128 \
      } / ,
    / unprotected / {
      / iv / 5:h'89f52f65a1c580933b5261a78c'
    },
    / ciphertext / h'5974e1b99a3a4cc09a659aa2e9e7fff161d38ce7edd5617
388e77baf'
  ]
)
```

C.4.2. Encrypted Message w/ a Partial IV

This example uses the following:

- CEK: AES-CCM w/ 128-bit key and a 64-bit tag

- Prefix for IV is 89F52F65A1C580933B52

Size of binary file is 41 bytes

```
16(
  [
    / protected / h'a1010a' / {
        \ alg \ 1:10 \ AES-CCM-16-64-128 \
      } / ,
    / unprotected / {
      / partial iv / 6:h'61a7'
    },
    / ciphertext / h'252a8911d465c125b6764739700f0141ed09192da5c69e5
33abf852b'
  ]
)
```

C.5. Examples of MACed messages

C.5.1. Shared Secret Direct MAC

This example uses the following:

- MAC: AES-CMAC, 256-bit key, truncated to 64 bits

- Recipient class: direct shared secret

Size of binary file is 57 bytes

```
97(
  [
    / protected / h'a1010f' / {
        \ alg \ 1:15 \ AES-CBC-MAC-256//64 \
      } / ,
    / unprotected / {},
    / payload / 'This is the content.',
    / tag / h'9e1226ba1f81b848',
    / recipients / [
      [
        / protected / h'',
        / unprotected / {
          / alg / 1:-6 / direct /,
          / kid / 4:'our-secret'
        },
        / ciphertext / h''
      ]
    ]
  ]
)
```

C.5.2. 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 214 bytes

```
97(
  [
    / protected / h'a10105' / {
        \ alg \ 1:5 \ HMAC 256//256 \
      } / ,
    / unprotected / {},
    / payload / 'This is the content.',
    / tag / h'81a03448acd3d305376eaa11fb3fe416a955be2cbe7ec96f012c99
4bc3f16a41',
    / recipients / [
      [
        / protected / h'a101381a' / {
            \ alg \ 1:-27 \ ECDH-SS + HKDF-256 \
          } / ,
        / unprotected / {
          / static kid / -3:'peregrin.took@tuckborough.example',
          / kid / 4:'meriadoc.brandybuck@buckland.example',
          / U nonce / -22:h'4d8553e7e74f3c6a3a9dd3ef286a8195cbf8a23d
19558ccfec7d34b824f42d92bd06bd2c7f0271f0214e141fb779ae2856abf585a583
68b017e7f2a9e5ce4db5'
        },
        / ciphertext / h''
      ]
    ]
  ]
)
```

C.5.3. Wrapped MAC

This example uses the following:

o MAC: AES-MAC, 128-bit key, truncated to 64 bits

o Recipient class: AES keywrap w/ a pre-shared 256-bit key

Size of binary file is 109 bytes

```
97(
  [
    / protected / h'a1010e' / {
        \ alg \ 1:14 \ AES-CBC-MAC-128//64 \
      } / ,
    / unprotected / {},
    / payload / 'This is the content.',
    / tag / h'36f5afaf0bab5d43',
    / recipients / [
      [
        / protected / h'',
        / unprotected / {
          / alg / 1:-5 / A256KW /,
          / kid / 4:'018c0ae5-4d9b-471b-bfd6-eef314bc7037'
        },
        / ciphertext / h'711ab0dc2fc4585dce27effa6781c8093eba906f227
b6eb0'
      ]
    ]
  ]
)
```

C.5.4. Multi-recipient MACed 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 309 bytes

```
   97(
     [
       / protected / h'a10105' / {
           \ alg \ 1:5 \ HMAC 256//256 \
         } / ,
       / unprotected / {},
       / payload / 'This is the content.',
       / tag / h'bf48235e809b5c42e995f2b7d5fa13620e7ed834e337f6aa43df16
   1e49e9323e',
       / recipients / [
         [
           / protected / h'a101381c' / {
               \ alg \ 1:-29 \ ECHD-ES+A128KW \
             } / ,
           / unprotected / {
             / ephemeral / -1:{
               / kty / 1:2,
               / crv / -1:3,
               / x / -2:h'0043b12669acac3fd27898ffba0bcd2e6c366d53bc4db
   71f909a759304acfb5e18cdc7ba0b13ff8c7636271a6924b1ac63c02688075b55ef2
   d613574e7dc242f79c3',
               / y / -3:true
             },
             / kid / 4:'bilbo.baggins@hobbiton.example'
           },
           / ciphertext / h'339bc4f79984cdc6b3e6ce5f315a4c7d2b0ac466fce
   a69e8c07dfbca5bb1f661bc5f8e0df9e3eff5'
         ],
         [
           / protected / h'',
           / unprotected / {
             / alg / 1:-5 / A256KW /,
             / kid / 4:'018c0ae5-4d9b-471b-bfd6-eef314bc7037'
           },
           / ciphertext / h'0b2c7cfce04e98276342d6476a7723c090dfdd15f9a
   518e7736549e998370695e6d6a83b4ae507bb'
         ]
       ]
     ]
   )
```

C.6. Examples of MAC0 messages

C.6.1. Shared Secret Direct MAC

This example uses the following:

- MAC: AES-CMAC, 256-bit key, truncated to 64 bits o Recipient class: direct shared secret

Size of binary file is 37 bytes

```
17(
  [
    / protected / h'a1010f' / {
        \ alg \ 1:15 \ AES-CBC-MAC-256//64 \
      } / ,
    / unprotected / {},
    / payload / 'This is the content.',
    / tag / h'726043745027214f'
  ]
)
```

Note that this example uses the same inputs as Appendix C.5.1.

C.7. COSE Keys

C.7.1. 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

```
[
  {
    -1:1,
    -2:h'65eda5a12577c2bae829437fe338701a10aaa375e1bb5b5de108de439c0
8551d',
    -3:h'1e52ed75701163f7f9e40ddf9f341b3dc9ba860af7e0ca7ca7e9eecd008
4d19c',
    1:2,
    2:'meriadoc.brandybuck@buckland.example'
  },
  {
    -1:1,
    -2:h'bac5b11cad8f99f9c72b05cf4b9e26d244dc189f745228255a219a86d6a
09eff',
    -3:h'20138bf82dc1b6d562be0fa54ab7804a3a64b6d72ccfed6b6fb6ed28bbf
c117e',
    1:2,
    2:'11'
  },
  {
    -1:3,
    -2:h'0072992cb3ac08ecf3e5c63dedec0d51a8c1f79ef2f82f94f3c737bf5de
7986671eac625fe8257bbd0394644caaa3aaf8f27a4585fbbcad0f2457620085e5c8
f42ad',
    -3:h'01dca6947bce88bc5790485ac97427342bc35f887d86d65a089377e247e
60baa55e4e8501e2ada5724ac51d6909008033ebc10ac999b9d7f5cc2519f3fe1ea1
d9475',
    1:2,
    2:'bilbo.baggins@hobbiton.example'
  },
  {
    -1:1,
    -2:h'98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbfbf054e1c7b4d91
d6280',
    -3:h'f01400b089867804b8e9fc96c3932161f1934f4223069170d924b7e03bf
822bb',
    1:2,
    2:'peregrin.took@tuckborough.example'
  }
]
```

C.7.2. 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:

- 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 816 bytes


```
[
  {
    1:2,
    2:'meriadoc.brandybuck@buckland.example',
    -1:1,
    -2:h'65eda5a12577c2bae829437fe338701a10aaa375e1bb5b5de108de439c0
8551d',
    -3:h'1e52ed75701163f7f9e40ddf9f341b3dc9ba860af7e0ca7ca7e9eecd008
4d19c',
    -4:h'aff907c99f9ad3aae6c4cdf21122bce2bd68b5283e6907154ad911840fa
208cf'
  },
  {
    1:2,
    2:'11',
    -1:1,
    -2:h'bac5b11cad8f99f9c72b05cf4b9e26d244dc189f745228255a219a86d6a
09eff',
    -3:h'20138bf82dc1b6d562be0fa54ab7804a3a64b6d72ccfed6b6fb6ed28bbf
c117e',
    -4:h'57c92077664146e876760c9520d054aa93c3afb04e306705db609030850
7b4d3'
  },
  {
    1:2,
    2:'bilbo.baggins@hobbiton.example',
    -1:3,
    -2:h'0072992cb3ac08ecf3e5c63dedec0d51a8c1f79ef2f82f94f3c737bf5de
7986671eac625fe8257bbd0394644caaa3aaf8f27a4585fbbcad0f2457620085e5c8
f42ad',
    -3:h'01dca6947bce88bc5790485ac97427342bc35f887d86d65a089377e247e
60baa55e4e8501e2ada5724ac51d6909008033ebc10ac999b9d7f5cc2519f3fe1ea1
d9475',
```

```
        -4:h'00085138ddabf5ca975f5860f91a08e91d6d5f9a76ad4018766a476680b
   55cd339e8ab6c72b5facdb2a2a50ac25bd086647dd3e2e6e99e84ca2c3609fdf177f
   eb26d'
     },
     {
       1:4,
       2:'our-secret',
       -1:h'849b57219dae48de646d07dbb533566e976686457c1491be3a76dcea6c4
   27188'
     },
     {
       1:2,
       -1:1,
       2:'peregrin.took@tuckborough.example',
       -2:h'98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbfbf054e1c7b4d91
   d6280',
       -3:h'f01400b089867804b8e9fc96c3932161f1934f4223069170d924b7e03bf
   822bb',
       -4:h'02d1f7e6f26c43d4868d87ceb2353161740aacf1f7163647984b522a848
   df1c3'
     },
     {
       1:4,
       2:'our-secret2',
       -1:h'849b5786457c1491be3a76dcea6c4271'
     },
     {
       1:4,
       2:'018c0ae5-4d9b-471b-bfd6-eef314bc7037',
       -1:h'849b57219dae48de646d07dbb533566e976686457c1491be3a76dcea6c4
   27188'
     }
   ]
```

Acknowledgments

This document is a product of the COSE working group of the IETF.

The following individuals are to blame for getting me started on this project in the first place: Richard Barnes, Matt Miller, and Martin Thomson.

The initial version of the draft was based to some degree on the outputs of the JOSE and S/MIME working groups.

The following individuals provided input into the final form of the document: Carsten Bormann, John Bradley, Brain Campbell, Michael B.

Jones, Ilari Liusvaara, Francesca Palombini, Goran Selander, and Ludwig Seitz.

Author's Address

Jim Schaad
August Cellars

Email: ietf@augustcellars.com

ACE Working Group   G. Selander
Internet-Draft   J. Mattsson
Intended status: Standards Track   F. Palombini
Expires: April 21, 2016   Ericsson
                                                                L. Seitz
                                                        SICS Swedish ICT
                                                        October 19, 2015

Object Security of CoAP (OSCOAP)
draft-selander-ace-object-security-03

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

Copyright Notice

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

B.1. MAC Only 26
B.2. Signature Only 27
B.3. Authenticated Encryption with Additional Data (AEAD) . . 29
B.4. Symmetric Encryption with Asymmetric Signature (SEAS) . . 30
Appendix C. Object Security of Content (OSCON) 32
C.1. Security Considerations of OSCON 33
Appendix D. Examples 34
D.1. CoAP Message Protection 34
D.1.1. Integrity Protection of CoAP Message Exchange 34
D.1.2. Additional Encryption of CoAP Message 36
D.2. Payload Protection 37
D.2.1. Proxy Caching 37
D.2.2. Publish-Subscribe 38
D.2.3. Transporting Authorization Information 40
Authors' Addresses 41

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:

o the Object-Security option, indicating that OSCOAP is being used;

o a compact cryptographic message format called "Secure Message", based on the COSE message format ([I-D.ietf-cose-msg]); and

o 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.

No.	C	U	N	R	Name	Format	Length
TBD	x		x		Object-Security	opaque	0, TBD

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 has 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:

- o 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.

- o 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 a 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:

- o The CoAP header SHALL be the same as the unprotected CoAP message.

- o The CoAP options SHALL consist of the same options as the unprotected CoAP message, and the Object-Security option.

- o 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.

- o 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).

- o 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:

- o the SM Header;

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

- o 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

- o 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:

- o 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

- o 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.

No.	C	U	N	R	Name	Format	Length	E	IP
1	x			x	If-Match	opaque	0-8	x	x
3	x	x	-		Uri-Host	string	1-255		a
4				x	ETag	opaque	1-8	x	x
5	x				If-None-Match	empty	0	x	x
6		x	-		Observe	uint	0-3		
7	x	x	-		Uri-Port	uint	0-2		a
8				x	Location-Path	string	0-255	x	x
11	x	x	-	x	Uri-Path	string	0-255	x	b
12					Content-Format	uint	0-2	x	x
14		x	-		Max-Age	uint	0-4		
15	x	x	-	x	Uri-Query	string	0-255	x	b
17	x				Accept	uint	0-2	x	x
20				x	Location-Query	string	0-255	x	x
23	x	x	-		Block2	uint	0-3		
27	x	x	-		Block1	uint	0-3		
28			x		Size2	uint	0-4	x	x
35	x	x	-		Proxy-Uri	string	1-1034		i
39	x	x	-		Proxy-Scheme	string	1-255		i
60			x		Size1	uint	0-4	x	x

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.

- o 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:

- o 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

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

- o 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.)

- o 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):

- o 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.

- o 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:

- o All Uri-* options removed

- o A Proxy-Uri option with the full URI included

- o 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:

- o Max-Age, Observe, Block1, Block2, Proxy-Uri, Proxy-Scheme: This information is intended to be read by a proxy.

- o Uri-Host, Uri-Port: This information can be inferred from destination IP address and port.

- o 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 malicous 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:

Number	Name	Reference
TBD	Object-Security	[[this document]]

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

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <http://www.rfc-editor.org/info/rfc6347>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <http://www.rfc-editor.org/info/rfc7252>.

[RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an Attack", BCP 188, RFC 7258, DOI 10.17487/RFC7258, May 2014, <http://www.rfc-editor.org/info/rfc7258>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <http://www.rfc-editor.org/info/rfc7515>.

[RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <http://www.rfc-editor.org/info/rfc7516>.

12.2. Informative References

[I-D.ietf-ace-actors]
Gerdes, S., Seitz, L., Selander, G., and C. Bormann, "An architecture for authorization in constrained environments", draft-ietf-ace-actors-02 (work in progress), September 2015.

[I-D.ietf-ace-usecases]
Seitz, L., Gerdes, S., Selander, G., Mani, M., and S. Kumar, "ACE use cases", draft-ietf-ace-usecases-09 (work in progress), October 2015.

[I-D.ietf-core-block]
Bormann, C. and Z. Shelby, "Block-wise transfers in CoAP", draft-ietf-core-block-18 (work in progress), September 2015.

[I-D.ietf-cose-msg]
Schaad, J. and B. Campbell, "CBOR Encoded Message Syntax", draft-ietf-cose-msg-05 (work in progress), September 2015.

[I-D.seitz-ace-core-authz]
Seitz, L., Selander, G., and M. Vucinic, "Authorization for Constrained RESTful Environments", draft-seitz-ace-core-authz-00 (work in progress), June 2015.

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007, <http://www.rfc-editor.org/info/rfc4949>.

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

[RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <http://www.rfc-editor.org/info/rfc7228>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <http://www.rfc-editor.org/info/rfc7231>.

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.

- o The "COSE_MSG" top level object as defined in COSE corresponds to the Secure Message object.
- o 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.
- o 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:

o The "protected" field includes:

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

o 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:

o The "protected" field includes:

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

o 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.

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

- o 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).

- o The "payload" field (as described in Appendix A.1.1 and Appendix A.1.2).

- o 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.

- o The "msg_type" parameter takes a new value, msg_type_encryption=6.

- o 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).

- o The "ciphertext" field (as described in Appendix A.2).

- o 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:

- o cid: 0xa1534e3c5fdc09bd

- o 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:

Overhead = Total Size - (MAC + 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'a20104637365714311223',                 # protected:
                                              {1: 4,
                                              "seq": h'112233'}
  {},                                       # unprotected
  h'',                                      # payload
  MAC,                                      # truncated 16-byte MAC
  [                                         # recipients
    [                                       # recipient structure
      h'',                                  # protected
      {1:-6, "cid":h'a1534e3c5fdc09bd'},    # 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:

```
[
  5,                                         # msg_type
  h'a20b48a1534e3c5fdc09bd0c43112233',       # protected:
                                               {11:h'a1534e3c5fdc09bd',
                                                12:h'112233'}
  h'',                                       # payload
  MAC                                        # truncated 16-byte MAC
]
```

This modified COSE object encodes to a total size of 37 bytes.

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

```
[
  h'112233',                                 # seq
  h'a1534e3c5fdc09bd',                       # 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:

```
[
  1,                                        # msg_type
  h'a16373657143112233',                    # protected:
                                              {"seq": h'112233'}
  {},                                       # unprotected
  h'',                                      # payload
  [                                         # signatures
    [                                       # signature structure
      h'a201266363696448a1534e3c5fdc09bd',  # protected:
                                              {1: -7,
                                            "cid":h'a1534e3c5fdc09bd'}
      {},                                   # unprotected
      SIG                                   # 64-byte signature
    ]
  ]
]
```

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
]
```

This object encodes to a total size of 81 bytes.

Figure 4 summarizes these results.

Scheme	seq+cid	SIG	Total Size	Overhead
COSE	11 B	64 B	100 bytes	25 bytes
mod-COSE	11 B	64 B	86 bytes	11 bytes
bound	11 B	64 B	81 bytes	6 bytes

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'a2010463736571431122233',             # protected:
                                            {1: 4,
                                            "seq": h'112233'}
  {},                                     # unprotected
  TAG,                                    # 8byte authentication tag
  [                                       # recipients
    [                                     # recipient structure
      h'',                                # protected
      {1:-6, "cid":h'a1534e3c5fdc09bd'},  # 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:

```
[
  6,                                        # msg_type
  h'a20b48a1534e3c5fdc09bd0c43112233',      # protected:
                                              {11:h'a1534e3c5fdc09bd',
                                               12:h'112233'}
  h'',                                      # ciphertext
  TAG                                       # 8byte authentication tag
]
```

The modified COSE object encodes to a total size of 29 bytes.

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

```
[
  h'112233',                                # seq
  h'a1534e3c5fdc09bd',                      # 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:

```
[
  1,                                       # msg_type
  h'a16373657143112233',                   # protected:
                                             {"seq": h'112233'}
  {},                                      # unprotected
  h'85024ba2010a6373657143112233a04081834
  0a201256363696448a1534e3c5fdc09bd40',    # payload:
                                        [2,
                                   h'a2010a6373657143112233',
                                   {}, h', [[h'',
                                   {1: -6,
                                        "cid": h'a1534e3c5fdc09bd'
                                        }, h'']]]
    [                                      # signatures
      [                                    # signature structure
        h'a201266363696448a1534e3c5fdc09bd', # protected:
                                             {1: -7,
                                           "cid":h'a1534e3c5fdc09bd'}
        {},                                # unprotected
        SIG                                # 64-byte signature
      ]
    ]
]
```

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'a20b48a1534e3c5fdc09bd0c43112233',     # protected:
                                             {11:h'a1534e3c5fdc09bd',
                                              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'a1534e3c5fdc09bd',                     # cid
  h'',                                     # ciphertext
  SIG                                      # 64-byte signature
]
```

This object encodes to a total size of 81 bytes.

Figure 6 summarizes these results.

Scheme	seq+cid	SIG	Total Size	Overhead
COSE	11 B	64 B	134 bytes	59 bytes
mod-COSE	11 B	64 B	86 bytes	11 bytes
bound	11 B	64 B	81 bytes	6 bytes

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).

```
   Client  Proxy  Server
      |      |      |
      |      |      |
      |      |      |
      +----->|      |      Code: 0.03 (PUT)
      | PUT  |      |     Token: 0x8c
      |      |      |  Uri-Path: lock
      |      |      |  Object-Security:
      |      |      |   Payload: ["seq":"142",
      |      |      |            "cid":"a1534e3c5fdc09bd", 1, <Tag>]
      |      |      |
      |      +----->|      Code: 0.03 (PUT)
      |      | PUT  |    Token: 0x7b
      |      |      |  Uri-Path: lock
      |      |      |  Object-Security:
      |      |      |   Payload: ["seq":"142",
      |      |      |            "cid":"a1534e3c5fdc09bd", 1, <Tag>]
      |      |      |
      |      |<-----+      Code: 2.04 (Changed)
      |      | 2.04 |     Token: 0x7b
      |      |      |   Object-Security: ["seq":"a6",
      |      |      |            "cid":"5fdc09bda1534e3c", , <Tag>]
      |      |      |
      |<-----+      |      Code: 2.04 (Changed)
      | 2.04 |      |     Token: 0x8c
      |      |      |   Object-Security: ["seq":"a6",
      |      |      |            "cid":"5fdc09bda1534e3c", , <Tag>]
      |      |      |
```

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 supports payload the Object-Security option carries the OSCOAP object.

The Header contains Sequence Number ("seq":"a6") and Context Identifier ("cid":"5fdc09bda1534e3c"), 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
      |      |      |
      |      |      |
      |      |      |
      +----->|      |      Code: 0.01 (GET)
      | GET  |      |     Token: 0x83
      |      |      |   Object-Security: ["seq":"15b7",
      |      |      |            "cid":"34e3c5fdca1509bd",
      |      |      |            {"glucose"}, <Tag>]
      |      |      |
      |      +----->|      Code: 0.01 (GET)
      |      | GET  |     Token: 0xbe
      |      |      |   Object-Security: ["seq":"15b7",
      |      |      |            "cid":"34e3c5fdca1509bd",
      |      |      |            {"glucose"}, <Tag>]
      |      |      |
      |      |      |
      |      |<-----+      Code: 2.05 (Content)
      |      | 2.05 |     Token: 0xbe
      |      |      |   Object-Security:
      |      |      |   Payload: ["seq":"32c9",
      |      |      |            "cid":"c09bda155fd34e3c",
      |      |      |            {220}, <Tag>]
      |      |      |
      |<-----+      |      Code: 2.05 (Content)
      | 2.05 |      |     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.

```
   Client1 Proxy  Server

      |      |      |
      |      |      |
      +----->|      |      Code: 0.01 (GET)
      | GET  |      |     Token: 0x83
      |      |      | Proxy-Uri: example.com/temp
      |      |      |
      |      |      |
      |      +----->|      Code: 0.01 (GET)
      |      | GET  |     Token: 0xbe
      |      |      |  Uri-Host: example.com
      |      |      |  Uri-Path: temp
      |      |      |
      |      |      |
      |      |<-----+      Code: 2.05 (Content)
      |      | 2.05 |     Token: 0xbe
      |      |      |   Payload: ["seq":"15b7",
      |      |      |            "cid":"c09bda155fd34e3c",
      |      |      |            "471 F", <Tag>]
      |      |      |
      |<-----+      |      Code: 2.05 (Content)
      | 2.05 |      |     Token: 0x83
      |      |      |   Payload: ["seq":"15b7",
      |      |      |            "cid":"c09bda155fd34e3c",
             |      |            "471 F", <Tag>]
   Client2   |      |
             |      |
      |      |      |
      |      |      |
      +----->|      |      Code: 0.01 (GET)
      | GET  |      |     Token: 0xa1
      |      |      | Proxy-Uri: example.com/temp
      |      |      |
      |<-----+      |      Code: 2.05 (Content)
      | 2.05 |      |     Token: 0xa1
      |      |      |   Payload: ["seq":"15b7",
      |      |      |            "cid":"c09bda155fd34e3c",
      |      |      |            "471 F", <Tag>]
```

Figure 9: Proxy caching protected with Object Security of Content (OSCON)

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.

```
Sub-    PubSub- Pub-
scriber Broker  lisher

   |      |      |
   +----->|      |      Code: 0.01 (GET)
   | GET  |      |     Token: 0x72
   |      |      |  Uri-Path: ps
   |      |      |  Uri-Path: birch-pollen
   |      |      |   Observe: 0 (register)
   |      |      |
   |      |      |
   |<-----+      |      Code: 2.05 (Content)
   | 2.05 |      |     Token: 0x72
   |      |      |   Observe: 1
   |      |      |   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.

```
Authz           Resource
Server  Client  Server
   |       |       |
   |       |       |     Client accesses Access Token:
   +- - ->|       |     ["seq":"142",
   |       |       |     "cid":"c09bda1534e3c5fdc09bd",
   |       |       |            <AuthzInfo>, <Tag>]
   |       |       |
   |       +------>|      Code: 0.02 (POST)
   |       | POST  |     Token: 0xac
   |       |       |  Uri-Path: authz-info
   |       |       |   Payload: ["seq":"142",
   |       |       |            "cid":"c09bda1534e3c5fdc09bd",
   |       |       |            <AuthzInfo>, <Tag>]

a)
   |       |       |
   |       |<------+      Code: 2.04 (Changed)
   |       | 2.04  |     Token: 0xac
   |       |       |

b)
   |       |       |
   |       |<------+      Code: 4.01 (Unauthorized)
   |       | 4.01  |     Token: 0xac
   |       |       |
```

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
Scheelevagen 17
Lund 22370
Sweden

Email: ludwig@sics.se