

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

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 26, 2017.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	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:

- o 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.

- o 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.
- o 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_structure 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_structure 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 symmetric 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]  
]
```

5.1.1.1. 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.

8.1.1.1. 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:

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

- o If the 'alg' field is present, it MUST match the HMAC algorithm being used.
- o If the 'key_ops' field is present, it MUST include 'MAC create' when creating an HMAC authentication tag.
- o 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 probability 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.

12.1.1.1. 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

12.1.1.1.1. 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.

12.1.1.2. 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).

- o 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:

- o The 'kty' field MUST be present and it MUST be 'Symmetric'.
- o If the 'alg' field is present, it MUST match the AES Key Wrap 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.

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:

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

- o 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.
- o 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:

- o The 'protected' field is fed into the KDF context structure.
- o The plain text to be encrypted is the key from next layer down (usually the content layer).
- o The 'alg' parameter MUST be present in the layer.
- o 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
<code>crv</code>	1	-1	int / tstr	EC Curve identifier - Taken from the COSE Key Common Parameters registry
<code>x</code>	1	-2	bstr	X Coordinate
<code>d</code>	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:

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

- o 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.
- o 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.
- o 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.
- o 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.
- o 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 addition 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 addition 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 consensus 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 recommended 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 addition 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 addition 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 consensus 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 initial 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.

- o 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.
- o 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.
- o 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:

- o What are the permissions associated with the key owner?
- o Is the cryptographic algorithm acceptable in the current context?
- o Have the restrictions associated with the key, such as algorithm or freshness, been checked and are correct?
- o Is the request something that is reasonable, given the current state of the application?
- o 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.)

- o 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.
- o 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:

- o 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_structure`.

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:

- o Layer 0: Has a content encrypted with AES-GCM using a 128-bit key.
- o Layer 1: Uses the AES Key wrap algorithm with a 128-bit key.
- o 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:

- o Signature Algorithm: ECDSA w/ SHA-256, Curve P-256
- o 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
69351210a031d8fc7c5ff3e4be39445b1a3e83e1510dlaca2f2e8a7c081c7645042b
18aba9d1fad1bd9c'
      ]
    ]
  ]
)

```

C.2. Single Signer Examples

C.2.1. Single ECDSA signature

This example uses the following:

- o 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:

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

- o CEK: AES-CCM w/128-bit key, truncate the tag to 64 bits
- o 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'89f52f65alc580933b5261a76c'
    },
    / 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
elc62'
      ]
    ]
  ]
)

```

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'89f52f65alc580933b5261a78c'
    },
    / ciphertext / h'5974e1b99a3a4cc09a659aa2e9e7fff161d38ce7edd5617
388e77baf'
  ]
)
```

C.4.2. Encrypted Message w/ a Partial IV

This example uses the following:

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

- o MAC: AES-CMAC, 256-bit key, truncated to 64 bits
- o 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'9e1226balf81b848',
    / 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:

- o MAC: HMAC w/SHA-256, 256-bit key
- o 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:

- o MAC: HMAC w/ SHA-256, 128-bit key
- o 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
71f909a759304acfb5e18cdc7ba0b13ff8c7636271a6924blac63c02688075b55ef2
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:

- o 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:

- o An EC key with a kid of "meriadoc.brandybuck@buckland.example"
- o An EC key with a kid of "peregrin.took@tuckborough.example"
- o An EC key with a kid of "bilbo.baggins@hobbiton.example"
- o 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:

- o An EC key with a kid of "meriadoc.brandybuck@buckland.example"
- o A shared-secret key with a kid of "our-secret"
- o An EC key with a kid of "peregrin.took@tuckborough.example"
- o A shared-secret key with a kid of "018c0ae5-4d9b-471b-bfd6-eef314bc7037"
- o An EC key with a kid of "bilbo.baggins@hobbiton.example"
- o 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

COSE Working Group
Internet-Draft
Intended status: Informational
Expires: September 22, 2016

J. Schaad
August Cellars
March 21, 2016

CBOR Encoded Message Syntax: Additional Algorithms
draft-schaad-cose-alg-01

Abstract

This document defines the identifiers and usage for a set of additional cryptographic algorithms in the CBOR Encoded Message (COSE) Syntax.

The algorithms setup in this document are: RSA-PSS, RSA-OAEP,
!!TBD!!

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-algs>>. Instructions are on that page as well. Editorial changes can be managed in GitHub, but any substantial issues need to be discussed on the COSE mailing list.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 22, 2016.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Requirements Terminology	3
1.2. Document Terminology	3
2. Signature Algorithms	3
2.1. RSASSA-PSS	3
2.1.1. Security Considerations	4
2.2. Edwards-curve Digital Signature Algorithms (EdDSA)	4
3. Message Authentication (MAC) Algorithms	6
4. Content Encryption Algorithms	6
5. Key Derivation Functions (KDF)	6
6. Recipient Algorithms	6
6.1. RSAES-OAEP	6
6.1.1. Security Considerations for RSAES-OAEP	6
6.2. ECDH	7
7. Keys	7
7.1. Octet Key Pair	8
7.2. RSA Keys	9
8. IANA Considerations	10
8.1. COSE Header Parameter Registry	10
8.2. COSE Header Algorithm Label Table	11
8.3. COSE Algorithm Registry	11
8.4. COSE Key Common Parameter Registry	11
8.5. COSE Key Type Parameter Registry	11
8.6. COSE Elliptic Curve Registry	11
9. Security Considerations	12
10. References	13
10.1. Normative References	13
10.2. Informative References	13
Appendix A. Document Updates	16
A.1. Version -00	16
Author's Address	17

1. Introduction

In the process of writing RFCXXXX [I-D.ietf-cose-msg] several algorithms were removed from that document to be addressed at a later date. This document deals with a large set of the cryptographic algorithms which were removed at that time.

This document provides the necessary conventions needed to use the algorithms defined in this document. This document additionally provides the necessary registration in the appropriate IANA registry tables.

1.1. 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.

1.2. Document Terminology

In this document we use the following terminology: [CREF1]

2. Signature Algorithms

This document defines two new signature algorithms: RSA-PSS and Edwards Curve Digital Signature Algorithm (EdDSA). Both of these signature algorithms are Signature Scheme with Appendix algorithms. (For a discussion on the difference between signature scheme with appendix and signature scheme with message recovery algorithms, see [I-D.ietf-cose-msg].)

2.1. RSASSA-PSS

The RSASSA-PSS signature algorithm is defined in [RFC3447].

The RSASSA-PSS signature algorithm is parametrized with a hash function (h), a mask generation function (mgf) and a salt length ($sLen$). For this specification, the mask generation function is fixed to be MGF1 as defined in [RFC3447]. It has been recommended that the same hash function be used for hashing the data as well as in the mask generation function, for this specification we follow this recommendation. The salt length is the same length as the hash function output.

Implementations need to check that the key type is 'RSA' when creating or verifying a signature.

The algorithms defined in this document can be found in Table 1.

name	value	hash	salt length	description
PS256	TBD1	SHA-256	32	RSASSA-PSS w/ SHA-256
PS384	TBD2	SHA-384	48	RSASSA-PSS w/ SHA-384
PS512	TBD3	SHA-512	64	RSASSA-PSS w/ SHA-512

Table 1: RSASSA-PSS Algorithm Values

2.1.1. Security Considerations

In addition to needing to worry about keys that are too small to provide the required security, there are issues with keys that are too large. Denial of service attacks have been mounted with overly large keys. This has the potential to consume resources with potentially bad keys. There are two reasonable ways to address this attack. First, a key should not be used for a cryptographic operation until it has been matched back to an authorized user. This approach means that no cryptography would be done except for authorized users. Second, applications can impose maximum as well as minimum length requirements on keys. This limits the resources consumed even if the matching is not performed until the cryptography has been done.

There is a theoretical hash substitution attack that can be mounted against RSASSA-PSS. However, the requirement that the same hash function be used consistently for all operations is an effective mitigation against it. Unlike ECDSA, hash functions are not truncated so that the full hash value is always signed. The internal padding structure of RSASSA-PSS means that one needs to have multiple collisions between the two hash functions in order to be successful in producing a forgery based on changing the hash function. This is highly unlikely.

2.2. 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 use with COSE, on 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. Thus, the use of an incremental update process would not be useful. Applications can provide the same features by defining the content of the message as a hash value and transporting the COSE message and the content as separate items.

The algorithms defined in this document can be found in Table 2. A single signature algorithm is defined which can be used for multiple curves.

name	value	description
EdDSA	*	EdDSA

Table 2: 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'.
- 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.

3. Message Authentication (MAC) Algorithms

This document defines no new Message Authentication Code algorithms.

4. Content Encryption Algorithms

This document defines no new content inception algorithms.

5. Key Derivation Functions (KDF)

This document defines new new key derivation functions.

6. Recipient Algorithms

6.1. RSAES-OAEP

RSAES-OAEP is an asymmetric key encryption algorithm. The definition of RSAEA-OAEP can be find in Section 7.1 of [RFC3447]. The algorithm is parameterized using a masking generation function (mgf), a hash function (h) and encoding parameters (P). For the algorithm identifiers defined in this section:

- o mgf is always set to MGF1 from [RFC3447] and uses the same hash function as h.
- o P is always set to the empty octet string.

Table 3 summarizes the rest of the values.

name	value	hash	description
RSAES-OAEP w/SHA-256	-25	SHA-256	RSAES OAEP w/ SHA-256
RSAES-OAEP w/SHA-512	-26	SHA-512	RSAES OAEP w/ SHA-512

Table 3: RSAES-OAEP Algorithm Values

The key type MUST be 'RSA'.

6.1.1. Security Considerations for RSAES-OAEP

A key size of 2048 bits or larger MUST be used with these algorithms. This key size corresponds roughly to the same strength as provided by a 128-bit symmetric encryption algorithm.

It is highly recommended that checks on the key length be done before starting a decryption operation. One potential denial of service operation is to provide encrypted objects using either abnormally long or oddly sized RSA modulus values. Implementations SHOULD be able to encrypt and decrypt with modulus between 2048 and 16K bits in length. Applications can impose additional restrictions on the length of the modulus.

6.2. ECDH

The algorithm ECDH is defined for use in COSE in [I-D.ietf-cose-msg]. In this document the algorithm is extended to be used with the two curves defined in [I-D.irtf-cfrg-curves].

The following updates [I-D.ietf-cose-msg] sections 12.4.1 and 12.5.1.

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

All the rest of the checks remain the same.

7. Keys

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

For each of the key types, we define both public and private members. The public members are what is transmitted to others for their usage. We define private members mainly for the purpose of archival of keys by individuals. However, there are some circumstances where private keys may be distributed by various entities in a protocol. Examples include: Entities which have poor random number generation. Centralized key creation for multi-cast type operations. Protocols where 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
OPK	TBDXX	Octet Key Pair
RSA	TBDXX1	RSA Keys

Table 4: Key Type Values

7.1. 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 5. The members that are defined for this key type are:

`crv` contains an identifier of the curve to be used with the key.

[CREF2] The curves defined in this document for this key type can be found in Table 6. 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	value	type	description
crv	1	-1	int / tstr	EC Curve identifier - Taken from the COSE General Registry
x	1	-2	bstr	X Coordinate
d	1	-4	bstr	Private key

Table 5: EC Key Parameters

name	key type	value	description
Curve25519	EC1	TBDYY1	Curve 25519
Curve448	EC1	TBDYY2	Curve 448

Table 6: EC Curves

7.2. RSA Keys

This document defines a key structure for both the public and private halves of RSA keys. Together, an RSA public key and an RSA private key form an RSA key pair. [CREF3]

The document also provides support for the so-called "multi-prime" RSA where the modulus may have more than two prime factors. The benefit of multi-prime RSA is lower computational cost for the decryption and signature primitives. For a discussion on how multi-prime affects the security of RSA crypto-systems, the reader is referred to [MultiPrimeRSA].

This document follows the naming convention of [RFC3447] for the naming of the fields of an RSA public or private key. The table Table 7 provides a summary of the label values and the types associated with each of those labels. The requirements for fields for RSA keys are as follows:

- o For all keys, 'kty' MUST be present and MUST have a value of 3.
- o For public keys, the fields 'n' and 'e' MUST be present. All other fields defined in Table 7 MUST be absent.

- o For private keys with two primes, the fields 'other', 'r_i', 'd_i' and 't_i' MUST be absent, all other fields MUST be present.
- o For private keys with more than two primes, all fields MUST be present. For the third to nth primes, each of the primes is represented as a map containing the fields 'r_i', 'd_i' and 't_i'. The field 'other' is an array of those maps.

name	key type	value	type	description
n	3	-1	bstr	Modulus Parameter
e	3	-2	int	Exponent Parameter
d	3	-3	bstr	Private Exponent Parameter
p	3	-4	bstr	First Prime Factor
q	3	-5	bstr	Second Prime Factor
dP	3	-6	bstr	First Factor CRT Exponent
dQ	3	-7	bstr	Second Factor CRT Exponent
qInv	3	-8	bstr	First CRT Coefficient
other	3	-9	array	Other Primes Info
r_i	3	-10	bstr	i-th factor, Prime Factor
d_i	3	-11	bstr	i-th factor, Factor CRT Exponent
t_i	3	-12	bstr	i-th factor, Factor CRT Coefficient

Table 7: RSA Key Parameters

8. IANA Considerations

8.1. COSE Header Parameter Registry

There are currently no registration requests here

8.2. COSE Header Algorithm Label Table

TBD

8.3. COSE Algorithm Registry

TBD

8.4. COSE Key Common Parameter Registry

There are currently no registration tasks in this section.

8.5. COSE Key Type Parameter Registry

It is requested that IANA create a new registry "COSE Key Type Parameters".

The columns of the table are:

key type This field contains a descriptive string of a key type. This should be a value that is in the COSE General Values table and is placed in the 'kty' field of a COSE Key structure.

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

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

CBOR type This field contains the CBOR type for the field

description This field contains a brief description for the field

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

This registry will be initially populated by the values in Table 5, and Table 7. The specification column for all of these entries will be this document.

8.6. COSE Elliptic Curve Registry

It is requested that IANA create a new registry "COSE Elliptic Curve Parameters".

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 the integer values from 256 to 65535 and -65536 to -257 are designated as Specification Required. Integer values over 65535 are designated as first come first serve. Integer values less than -65536 are marked as private use.

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

description This field contains a brief description of the curve.

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

This registry will be initially populated by the values in Table 4. The specification column for all of these entries will be this document.

9. Security Considerations

There are security considerations:

1. Protect private keys
2. MAC messages with more than one recipient means one cannot figure out who sent the message
3. Use of direct key with other recipient structures hands the key to other recipients.
4. Use of direct ECDH direct encryption is easy for people to leak information on if there are other recipients in the message.
5. Considerations about protected vs unprotected header fields.
6. Need to verify that: 1) the kty field of the key matches the key and algorithm being used. 2) that the kty field needs to be included in the trust decision as well as the other key fields. 3) that the algorithm be included in the trust decision.

10. References

10.1. 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>>.
- [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>>.

10.2. Informative 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.
- [DSS] U.S. National Institute of Standards and Technology, "Digital Signature Standard (DSS)", July 2013.
- [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-07 (work in progress), October 2015.
- [I-D.ietf-cose-msg]
Schaad, J., "CBOR Encoded Message Syntax", draft-ietf-cose-msg-10 (work in progress), February 2016.
- [I-D.irtf-cfrg-curves]
Langley, A. and M. Hamburg, "Elliptic Curves for Security", draft-irtf-cfrg-curves-11 (work in progress), October 2015.
- [I-D.irtf-cfrg-eddsa]
Josefsson, S. and I. Liusvaara, "Edwards-curve Digital Signature Algorithm (EdDSA)", draft-irtf-cfrg-eddsa-05 (work in progress), March 2016.
- [MAC] NIST, N., "FIPS PUB 113: Computer Data Authentication", May 1985.
- [MultiPrimeRSA]
Hinek, M. and D. Cheriton, "On the Security of Multi-prime RSA", June 2006.

- [PVSig] Brown, D. and D. Johnson, "Formal Security Proofs for a Signature Scheme with Partial Message Recover", February 2000.
- [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>>.
- [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>>.
- [RFC2898] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, DOI 10.17487/RFC2898, September 2000, <<http://www.rfc-editor.org/info/rfc2898>>.
- [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>>.
- [RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, DOI 10.17487/RFC3447, February 2003, <<http://www.rfc-editor.org/info/rfc3447>>.
- [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>>.
- [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>>.
- [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>>.
- [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>>.
- [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>>.
- [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>>.
- [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>>.
- [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>>.
- [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>>.
- [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>>.
- [SEC1] Standards for Efficient Cryptography Group, "SEC 1: Elliptic Curve Cryptography", May 2009.
- [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.

Appendix A. Document Updates

A.1. Version -00

- o TBD

Editorial Comments

- [CREF1] JLS: I have not gone through the document to determine what needs to be here yet. We mostly want to grab terms which are used in unusual ways or are not generally understood.
- [CREF2] JLS: Is is the same registry for both OKP and EC2?
- [CREF3] JLS: Looking at the CBOR specification, the bstr that we are looking in our table below should most likely be specified as big numbers rather than as binary strings. This means that we would use the tag 6.2 instead. From my reading of the specification, there is no difference in the encoded size of the

resulting output. The specification of bignum does explicitly allow for integers encoded with leading zeros.

Author's Address

Jim Schaad
August Cellars

Email: ietf@augustcellars.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: March 14, 2020

G. Selander
J. Mattsson
F. Palombini
Ericsson AB
September 11, 2019

Ephemeral Diffie-Hellman Over COSE (EDHOC)
draft-selander-ace-cose-ecdhe-14

Abstract

This document specifies Ephemeral Diffie-Hellman Over COSE (EDHOC), a very compact, and lightweight authenticated Diffie-Hellman key exchange with ephemeral keys. EDHOC provides mutual authentication, perfect forward secrecy, and identity protection. EDHOC is intended for usage in constrained scenarios and a main use case is to establish an OSCORE security context. By reusing COSE for cryptography, CBOR for encoding, and CoAP for transport, the additional code footprint can be kept very low.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on March 14, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect

to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Rationale for EDHOC	4
1.2. Terminology and Requirements Language	5
2. Background	6
3. EDHOC Overview	7
3.1. Cipher Suites	9
3.2. Ephemeral Public Keys	9
3.3. Key Derivation	9
4. EDHOC Authenticated with Asymmetric Keys	12
4.1. Overview	12
4.2. EDHOC Message 1	14
4.3. EDHOC Message 2	16
4.4. EDHOC Message 3	19
5. EDHOC Authenticated with Symmetric Keys	21
5.1. Overview	21
5.2. EDHOC Message 1	22
5.3. EDHOC Message 2	23
5.4. EDHOC Message 3	23
6. Error Handling	24
6.1. EDHOC Error Message	24
7. Transferring EDHOC and Deriving Application Keys	25
7.1. Transferring EDHOC in CoAP	25
7.2. Transferring EDHOC over Other Protocols	28
8. Security Considerations	28
8.1. Security Properties	28
8.2. Cryptographic Considerations	29
8.3. Cipher Suites	30
8.4. Unprotected Data	30
8.5. Denial-of-Service	30
8.6. Implementation Considerations	31
8.7. Other Documents Referencing EDHOC	32
9. IANA Considerations	32
9.1. EDHOC Cipher Suites Registry	32
9.2. EDHOC Method Type Registry	32
9.3. The Well-Known URI Registry	33
9.4. Media Types Registry	33
9.5. CoAP Content-Formats Registry	34
9.6. Expert Review Instructions	34
10. References	35
10.1. Normative References	35
10.2. Informative References	37

Appendix A. Use of CBOR, CDDL and COSE in EDHOC	39
A.1. CBOR and CDDL	39
A.2. COSE	40
Appendix B. EDHOC Authenticated with Diffie-Hellman Keys	40
Appendix C. Test Vectors	41
C.1. Test Vectors for EDHOC Authenticated with Asymmetric Keys (RPK)	41
C.2. Test Vectors for EDHOC Authenticated with Symmetric Keys (PSK)	57
Acknowledgments	70
Authors' Addresses	70

1. Introduction

Security at the application layer provides an attractive option for protecting Internet of Things (IoT) deployments, for example where transport layer security is not sufficient [I-D.hartke-core-e2e-security-reqs] or where the protection needs to work over a variety of underlying protocols. IoT devices may be constrained in various ways, including memory, storage, processing capacity, and energy [RFC7228]. A method for protecting individual messages at the application layer suitable for constrained devices, is provided by CBOR Object Signing and Encryption (COSE) [RFC8152], which builds on the Concise Binary Object Representation (CBOR) [I-D.ietf-cbor-7049bis]. Object Security for Constrained RESTful Environments (OSCORE) [RFC8613] is a method for application-layer protection of the Constrained Application Protocol (CoAP), using COSE.

In order for a communication session to provide forward secrecy, the communicating parties can run an Elliptic Curve Diffie-Hellman (ECDH) key exchange protocol with ephemeral keys, from which shared key material can be derived. This document specifies Ephemeral Diffie-Hellman Over COSE (EDHOC), a lightweight key exchange protocol providing perfect forward secrecy and identity protection. Authentication is based on credentials established out of band, e.g. from a trusted third party, such as an Authorization Server as specified by [I-D.ietf-ace-oauth-authz]. EDHOC supports authentication using pre-shared keys (PSK), raw public keys (RPK), and public key certificates. After successful completion of the EDHOC protocol, application keys and other application specific data can be derived using the EDHOC-Exporter interface. A main use case for EDHOC is to establish an OSCORE security context. EDHOC uses COSE for cryptography, CBOR for encoding, and CoAP for transport. By reusing existing libraries, the additional code footprint can be kept very low. Note that this document focuses on authentication and key establishment: for integration with authorization of resource access, refer to [I-D.ietf-ace-oscore-profile].

EDHOC is designed to work in highly constrained scenarios making it especially suitable for network technologies such as Cellular IoT, 6TiSCH [I-D.ietf-6tisch-dtsecurity-zerotouch-join], and LoRaWAN [LoRa1][LoRa2]. These network technologies are characterized by their low throughput, low power consumption, and small frame sizes. Compared to the DTLS 1.3 handshake [I-D.ietf-tls-dtls13] with ECDH and connection ID, the number of bytes in EDHOC is less than 1/4 when PSK authentication is used and less than 1/3 when RPK authentication is used, see [I-D.ietf-lwig-security-protocol-comparison]. Typical message sizes for EDHOC with pre-shared keys, raw public keys, and X.509 certificates are shown in Figure 1.

	PSK	RPK	x5t	x5chain
message_1	40	38	38	38
message_2	45	114	126	116 + Certificate chain
message_3	11	80	91	81 + Certificate chain
Total	96	232	255	235 + Certificate chains

Figure 1: Typical message sizes in bytes

The ECDH exchange and the key derivation follow [SIGMA], NIST SP-800-56A [SP-800-56A], and HKDF [RFC5869]. CBOR [I-D.ietf-cbor-7049bis] and COSE [RFC8152] are used to implement these standards. The use of COSE provides crypto agility and enables use of future algorithms and headers designed for constrained IoT.

This document is organized as follows: Section 2 describes how EDHOC builds on SIGMA-I, Section 3 specifies general properties of EDHOC, including message flow, formatting of the ephemeral public keys, and key derivation, Section 4 specifies EDHOC with asymmetric key authentication, Section 5 specifies EDHOC with symmetric key authentication, Section 6 specifies the EDHOC error message, and Section 7 describes how EDHOC can be transferred in CoAP and used to establish an OSCORE security context.

1.1. Rationale for EDHOC

Many constrained IoT systems today do not use any security at all, and when they do, they often do not follow best practices. One reason is that many current security protocols are not designed with constrained IoT in mind. Constrained IoT systems often deal with personal information, valuable business data, and actuators interacting with the physical world. Not only do such systems need security and privacy, they often need end-to-end protection with

source authentication and perfect forward secrecy. EDHOC and OSCORE [RFC8613] enables security following current best practices to devices and systems where current security protocols are impractical.

EDHOC is optimized for small message sizes and can therefore be sent over a small number of radio frames. The message size of a key exchange protocol may have a large impact on the performance of an IoT deployment, especially in noisy environments. For example, in a network bootstrapping setting a large number of devices turned on in a short period of time may result in large latencies caused by parallel key exchanges. Requirements on network formation time in constrained environments can be translated into key exchange overhead. In networks technologies with transmission back-off time, each additional frame significantly increases the latency even if no other devices are transmitting.

Power consumption for wireless devices is highly dependent on message transmission, listening, and reception. For devices that only send a few bytes occasionally, the battery lifetime may be significantly reduced by a heavy key exchange protocol. Moreover, a key exchange may need to be executed more than once, e.g. due to a device losing power or rebooting for other reasons.

EDHOC is adapted to primitives and protocols designed for the Internet of Things: EDHOC is built on CBOR and COSE which enables small message overhead and efficient parsing in constrained devices. EDHOC is not bound to a particular transport layer, but it is recommended to transport the EDHOC message in CoAP payloads. EDHOC is not bound to a particular communication security protocol but works off-the-shelf with OSCORE [RFC8613] providing the necessary input parameters with required properties. Maximum code complexity (ROM/Flash) is often a constraint in many devices and by reusing already existing libraries, the additional code footprint for EDHOC + OSCORE can be kept very low.

1.2. Terminology and Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The word "encryption" without qualification always refers to authenticated encryption, in practice implemented with an Authenticated Encryption with Additional Data (AEAD) algorithm, see [RFC5116].

Readers are expected to be familiar with the terms and concepts described in CBOR [I-D.ietf-cbor-7049bis], COSE [RFC8152], and CDDL [RFC8610]. The Concise Data Definition Language (CDDL) is used to express CBOR data structures [I-D.ietf-cbor-7049bis]. Examples of CBOR and CDDL are provided in Appendix A.1.

2. Background

SIGMA (SIGn-and-Mac) is a family of theoretical protocols with a large number of variants [SIGMA]. Like IKEv2 and (D)TLS 1.3 [RFC8446], EDHOC is built on a variant of the SIGMA protocol which provide identity protection of the initiator (SIGMA-I), and like (D)TLS 1.3, EDHOC implements the SIGMA-I variant as Sign-then-MAC. The SIGMA-I protocol using an authenticated encryption algorithm is shown in Figure 2.

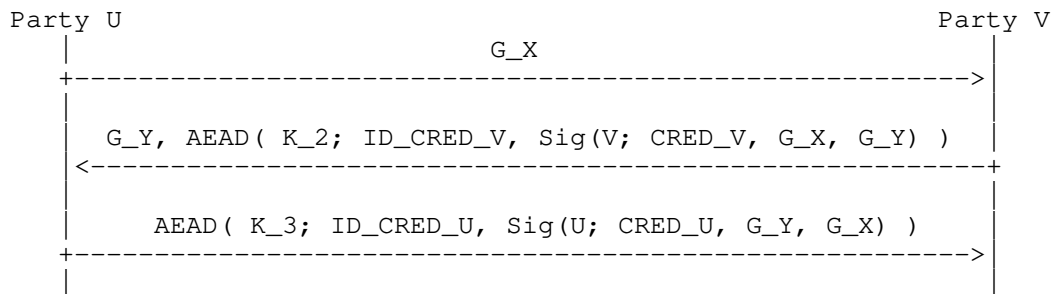


Figure 2: Authenticated encryption variant of the SIGMA-I protocol.

The parties exchanging messages are called "U" and "V". They exchange identities and ephemeral public keys, compute the shared secret, and derive symmetric application keys.

- o G_X and G_Y are the ECDH ephemeral public keys of U and V, respectively.
- o CRED_U and CRED_V are the credentials containing the public authentication keys of U and V, respectively.
- o ID_CRED_U and ID_CRED_V are data enabling the recipient party to retrieve the credential of U and V, respectively.
- o $\text{Sig}(U; \cdot)$ and $\text{Sig}(V; \cdot)$ denote signatures made with the private authentication key of U and V, respectively.
- o $\text{AEAD}(K; \cdot)$ denotes authenticated encryption with additional data using the key K derived from the shared secret. The authenticated

encryption MUST NOT be replaced by plain encryption, see Section 8.

In order to create a "full-fledged" protocol some additional protocol elements are needed. EDHOC adds:

- o Explicit connection identifiers C_U, C_V chosen by U and V, respectively, enabling the recipient to find the protocol state.
- o Transcript hashes TH_2, TH_3, TH_4 used for key derivation and as additional authenticated data.
- o Computationally independent keys derived from the ECDH shared secret and used for encryption of different messages.
- o Verification of a common preferred cipher suite (AEAD algorithm, ECDH algorithm, ECDH curve, signature algorithm):
 - * U lists supported cipher suites in order of preference
 - * V verifies that the selected cipher suite is the first supported cipher suite
- o Method types and error handling.
- o Transport of opaque application defined data.

EDHOC is designed to encrypt and integrity protect as much information as possible, and all symmetric keys are derived using as much previous information as possible. EDHOC is furthermore designed to be as compact and lightweight as possible, in terms of message sizes, processing, and the ability to reuse already existing CBOR, COSE, and CoAP libraries.

To simplify for implementors, the use of CBOR in EDHOC is summarized in Appendix A and test vectors including CBOR diagnostic notation are given in Appendix C.

3. EDHOC Overview

EDHOC consists of three flights (message_1, message_2, message_3) that maps directly to the three messages in SIGMA-I, plus an EDHOC error message. EDHOC messages are CBOR Sequences [I-D.ietf-cbor-sequence], where the first data item of message_1 is an int (TYPE) specifying the method (asymmetric, symmetric) and the correlation properties of the transport used.

While EDHOC uses the COSE_Key, COSE_Sign1, and COSE_Encrypt0 structures, only a subset of the parameters is included in the EDHOC messages. After creating EDHOC message_3, Party U can derive symmetric application keys, and application protected data can therefore be sent in parallel with EDHOC message_3. The application may protect data using the algorithms (AEAD, HMAC, etc.) in the selected cipher suite and the connection identifiers (C_U, C_V). EDHOC may be used with the media type application/edhoc defined in Section 9.

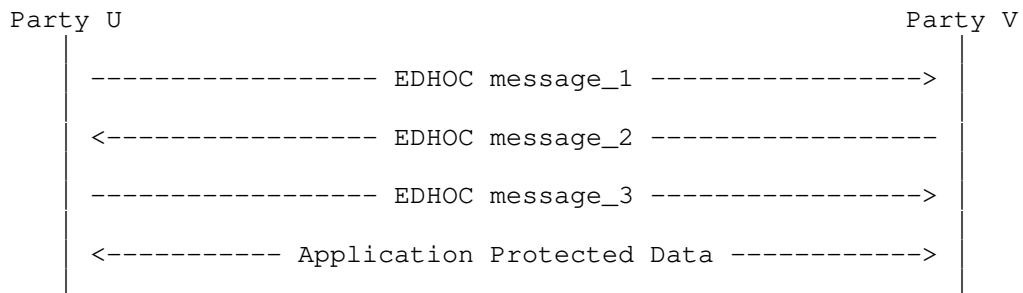


Figure 3: EDHOC message flow

The EDHOC message exchange may be authenticated using pre-shared keys (PSK), raw public keys (RPK), or public key certificates. EDHOC assumes the existence of mechanisms (certification authority, manual distribution, etc.) for binding identities with authentication keys (public or pre-shared). When a public key infrastructure is used, the identity is included in the certificate and bound to the authentication key by trust in the certification authority. When the credential is manually distributed (PSK, RPK, self-signed certificate), the identity and authentication key is distributed out-of-band and bound together by trust in the distribution method. EDHOC with symmetric key authentication is very similar to EDHOC with asymmetric key authentication, the difference being that information is only MACed, not signed, and that session keys are derived from the ECDH shared secret and the PSK.

EDHOC allows opaque application data (UAD and PAD) to be sent in the EDHOC messages. Unprotected Application Data (UAD_1, UAD_2) may be sent in message_1 and message_2 and can be e.g. be used to transfer access tokens that are protected outside of EDHOC. Protected application data (PAD_3) may be used to transfer any application data in message_3.

Cryptographically, EDHOC does not put requirements on the lower layers. EDHOC is not bound to a particular transport layer, and can be used in environments without IP. It is recommended to transport

the EDHOC message in CoAP payloads, see Section 7. An implementation may support only Party U or only Party V.

3.1. Cipher Suites

EDHOC cipher suites consist of an ordered set of COSE algorithms: an AEAD algorithm, an HMAC algorithm, an ECDH curve, a signature algorithm, and signature algorithm parameters. The signature algorithm is not used when EDHOC is authenticated with symmetric keys. Each cipher suite is either identified with a pre-defined int label or with an array of labels and values from the COSE Algorithms and Elliptic Curves registries.

```
suite = int / [ 4*4 algs: int / tstr, ? para: any ]
```

This document specifies two pre-defined cipher suites.

- 0. [10, 5, 4, -8, 6]
(AES-CCM-16-64-128, HMAC 256/256, X25519, EdDSA, Ed25519)
- 1. [10, 5, 1, -7, 1]
(AES-CCM-16-64-128, HMAC 256/256, P-256, ES256, P-256)

3.2. Ephemeral Public Keys

The ECDH ephemeral public keys are formatted as a COSE_Key of type EC2 or OKP according to Sections 13.1 and 13.2 of [RFC8152], but only the x-coordinate is included in the EDHOC messages. For Elliptic Curve Keys of type EC2, compact representation as per [RFC6090] MAY be used also in the COSE_Key. If the COSE implementation requires an y-coordinate, any of the possible values of the y-coordinate can be used, see Appendix C of [RFC6090]. COSE [RFC8152] always use compact output for Elliptic Curve Keys of type EC2.

3.3. Key Derivation

Key and IV derivation SHALL be performed with HKDF [RFC5869] following the specification in Section 11 of [RFC8152] using the HMAC algorithm in the selected cipher suite. The pseudorandom key (PRK) is derived using HKDF-Extract [RFC5869]

```
PRK = HKDF-Extract( salt, IKM )
```

with the following input:

- o The salt SHALL be the PSK when EDHOC is authenticated with symmetric keys, and the empty byte string when EDHOC is authenticated with asymmetric keys. The PSK is used as 'salt' to

simplify implementation. Note that [RFC5869] specifies that if the salt is not provided, it is set to a string of zeros (see Section 2.2 of [RFC5869]). For implementation purposes, not providing the salt is the same as setting the salt to the empty byte string.

- o The input keying material (IKM) SHALL be the ECDH shared secret G_XY as defined in Section 12.4.1 of [RFC8152]. When using the curve25519, the ECDH shared secret is the output of the X25519 function [RFC7748].

Example: Assuming use of HMAC 256/256 the extract phase of HKDF produces a PRK as follows:

```
PRK = HMAC-SHA-256( salt, G_XY )
```

where salt = 0x (the empty byte string) in the asymmetric case and salt = PSK in the symmetric case.

The keys and IVs used in EDHOC are derived from PRK using HKDF-Expand [RFC5869]

```
OKM = HKDF-Expand( PRK, info, L )
```

where L is the length of output keying material (OKM) in bytes and info is the CBOR encoding of a COSE_KDF_Context

```
info = [  
  AlgorithmID,  
  [ null, null, null ],  
  [ null, null, null ],  
  [ keyDataLength, h'', other ]  
]
```

where

- o AlgorithmID is an int or tstr, see below
- o keyDataLength is a uint set to the length of output keying material in bits, see below
- o other is a bstr set to one of the transcript hashes TH_2, TH_3, or TH_4 as defined in Sections 4.3.1, 4.4.1, and 3.3.1.

For message_2 and message_3, the keys K_2 and K_3 SHALL be derived using transcript hashes TH_2 and TH_3 respectively. The key SHALL be derived using AlgorithmID set to the integer value of the AEAD in the

selected cipher suite, and keyDataLength equal to the key length of the AEAD.

If the AEAD algorithm uses an IV, then IV_2 and IV_3 for message_2 and message_3 SHALL be derived using the transcript hashes TH_2 and TH_3 respectively. The IV SHALL be derived using AlgorithmID = "IV-GENERATION" as specified in Section 12.1.2. of [RFC8152], and keyDataLength equal to the IV length of the AEAD.

Assuming the output OKM length L is smaller than the hash function output size, the expand phase of HKDF consists of a single HMAC invocation

$$\text{OKM} = \text{first } L \text{ bytes of } \text{HMAC}(\text{PRK}, \text{info} \parallel 0x01)$$

where \parallel means byte string concatenation.

Example: Assuming use of the algorithm AES-CCM-16-64-128 and HMAC 256/256, K_i and IV_i are therefore the first 16 and 13 bytes, respectively, of

$$\text{HMAC-SHA-256}(\text{PRK}, \text{info} \parallel 0x01)$$

calculated with (AlgorithmID, keyDataLength) = (10, 128) and (AlgorithmID, keyDataLength) = ("IV-GENERATION", 104), respectively.

3.3.1. EDHOC-Exporter Interface

Application keys and other application specific data can be derived using the EDHOC-Exporter interface defined as:

$$\text{EDHOC-Exporter}(\text{label}, \text{length}) = \text{HKDF-Expand}(\text{PRK}, \text{info}, \text{length})$$

The output of the EDHOC-Exporter function SHALL be derived using AlgorithmID = label, keyDataLength = 8 * length, and other = TH_4 where label is a tstr defined by the application and length is a uint defined by the application. The label SHALL be different for each different exporter value. The transcript hash TH_4 is a CBOR encoded bstr and the input to the hash function is a CBOR Sequence.

$$\text{TH}_4 = \text{H}(\text{TH}_3, \text{CIPHERTEXT}_3)$$

where H() is the hash function in the HMAC algorithm. Example use of the EDHOC-Exporter is given in Sections 3.3.2 and 7.1.1.

3.3.2. EDHOC PSK Chaining

An application using EDHOC may want to derive new PSKs to use for authentication in future EDHOC exchanges. In this case, the new PSK and the ID_PSK 'kid_value' parameter SHOULD be derived as follows where length is the key length (in bytes) of the AEAD Algorithm.

```
PSK      = EDHOC-Exporter( "EDHOC Chaining PSK", length )
ID_PSK   = EDHOC-Exporter( "EDHOC Chaining ID_PSK", 4 )
```

4. EDHOC Authenticated with Asymmetric Keys

4.1. Overview

EDHOC supports authentication with raw public keys (RPK) and public key certificates with the requirements that:

- o Only Party V SHALL have access to the private authentication key of Party V,
- o Only Party U SHALL have access to the private authentication key of Party U,
- o Party U is able to retrieve Party V's public authentication key using ID_CRED_V,
- o Party V is able to retrieve Party U's public authentication key using ID_CRED_U,

where the identifiers ID_CRED_U and ID_CRED_V are COSE header_maps, i.e. a CBOR map containing COSE Common Header Parameters, see [RFC8152]). ID_CRED_U and ID_CRED_V need to contain parameters that can identify a public authentication key, see Appendix A.2. In the following we give some examples of possible COSE header parameters.

Raw public keys are most optimally stored as COSE_Key objects and identified with a 'kid' parameter (see [RFC8152]):

- o ID_CRED_x = { 4 : kid_value }, where kid_value : bstr, for x = U or V.

Public key certificates can be identified in different ways. Several header parameters for identifying X.509 certificates are defined in [I-D.ietf-cose-x509] (the exact labels are TBD):

- o by a hash value with the 'x5t' parameter;

* ID_CRED_x = { TBD1 : COSE_CertHash }, for x = U or V,

- o by a URL with the 'x5u' parameter;
 - * ID_CRED_x = { TBD2 : uri }, for x = U or V,
- o or by a bag of certificates with the 'x5bag' parameter;
 - * ID_CRED_x = { TBD3 : COSE_X509 }, for x = U or V.
- o by a certificate chain with the 'x5chain' parameter;
 - * ID_CRED_x = { TBD4 : COSE_X509 }, for x = U or V,

In the latter two examples, ID_CRED_U and ID_CRED_V contain the actual credential used for authentication. The purpose of ID_CRED_U and ID_CRED_V is to facilitate retrieval of a public authentication key and when they do not contain the actual credential, they may be very short. It is RECOMMENDED that they uniquely identify the public authentication key as the recipient may otherwise have to try several keys. ID_CRED_U and ID_CRED_V are transported in the ciphertext, see Section 4.3.2 and Section 4.4.2.

The actual credentials CRED_U and CRED_V (e.g. a COSE_Key or a single X.509 certificate) are signed by party U and V, respectively to prevent duplicate-signature key selection (DSKS) attacks, see Section 4.4.1 and Section 4.3.1. Party U and Party V MAY use different types of credentials, e.g. one uses RPK and the other uses certificate. When included in the signature payload, COSE_Keys of type OKP SHALL only include the parameters 1 (kty), -1 (crv), and -2 (x-coordinate). COSE_Keys of type EC2 SHALL only include the parameters 1 (kty), -1 (crv), -2 (x-coordinate), and -3 (y-coordinate). The parameters SHALL be encoded in decreasing order.

The connection identifiers C_U and C_V do not have any cryptographic purpose in EDHOC. They contain information facilitating retrieval of the protocol state and may therefore be very short. The connection identifier MAY be used with an application protocol (e.g. OSCORE) for which EDHOC establishes keys, in which case the connection identifiers SHALL adhere to the requirements for that protocol. Each party chooses a connection identifier it desires the other party to use in outgoing messages.

The first data item of message_1 is an int TYPE = 4 * method + corr specifying the method and the correlation properties of the transport used. corr = 0 is used when there is no external correlation mechanism. corr = 1 is used when there is an external correlation mechanism (e.g. the Token in CoAP) that enables Party U to correlate message_1 and message_2. corr = 2 is used when there is an external correlation mechanism that enables Party V to correlate message_2 and

message_3. corr = 3 is used when there is an external correlation mechanism that enables the parties to correlate all the messages. The use of the correlation parameter is exemplified in Section 7.1.

1 byte connection and credential identifiers are realistic in many scenarios as most constrained devices only have a few keys and connections. In cases where a node only has one connection or key, the identifiers may even be the empty byte string.

EDHOC with asymmetric key authentication is illustrated in Figure 4.

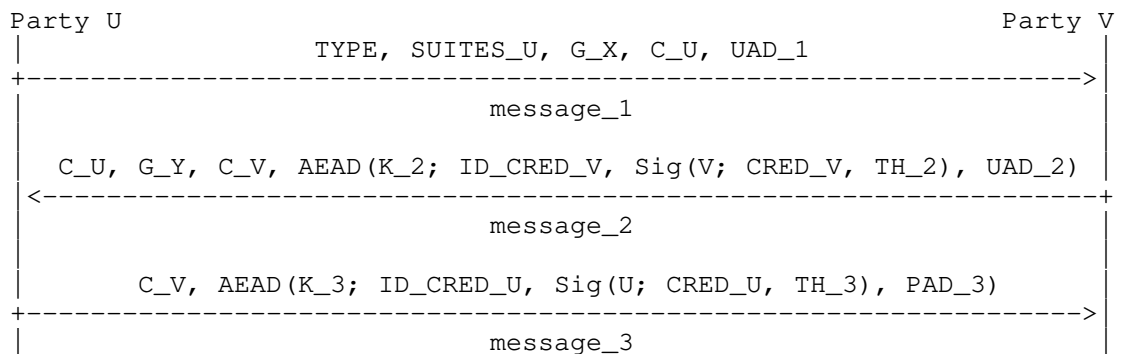


Figure 4: Overview of EDHOC with asymmetric key authentication.

4.2. EDHOC Message 1

4.2.1. Formatting of Message 1

message_1 SHALL be a CBOR Sequence (see Appendix A.1) as defined below

```

message_1 = (
  TYPE : int,
  SUITES_U : suite / [ index : uint, 2* suite ],
  G_X : bstr,
  C_U : bstr,
  ? UAD_1 : bstr,
)
    
```

where:

- o TYPE = 4 * method + corr, where the method = 0 and the correlation parameter corr is chosen based on the transport and determines which connection identifiers that are omitted (see Section 4.1).

- o SUITES_U - cipher suites which Party U supports in order of decreasing preference. One cipher suite is selected. If a single cipher suite is conveyed then that cipher suite is selected. If multiple cipher suites are conveyed then zero-based index (i.e. 0 for the first suite, 1 for the second suite, etc.) identifies the selected cipher suite out of the array elements listing the cipher suites (see Section 6).
- o G_X - the x-coordinate of the ephemeral public key of Party U
- o C_U - variable length connection identifier
- o UAD_1 - bstr containing unprotected opaque application data

4.2.2. Party U Processing of Message 1

Party U SHALL compose message_1 as follows:

- o The supported cipher suites and the order of preference MUST NOT be changed based on previous error messages. However, the list SUITES_U sent to Party V MAY be truncated such that cipher suites which are the least preferred are omitted. The amount of truncation MAY be changed between sessions, e.g. based on previous error messages (see next bullet), but all cipher suites which are more preferred than the least preferred cipher suite in the list MUST be included in the list.
- o Determine the cipher suite to use with Party V in message_1. If Party U previously received from Party V an error message to message_1 with diagnostic payload identifying a cipher suite that U supports, then U SHALL use that cipher suite. Otherwise the first cipher suite in SUITES_U MUST be used.
- o Generate an ephemeral ECDH key pair as specified in Section 5 of [SP-800-56A] using the curve in the selected cipher suite. Let G_X be the x-coordinate of the ephemeral public key.
- o Choose a connection identifier C_U and store it for the length of the protocol.
- o Encode message_1 as a sequence of CBOR encoded data items as specified in Section 4.2.1

4.2.3. Party V Processing of Message 1

Party V SHALL process message_1 as follows:

- o Decode message_1 (see Appendix A.1).

- o Verify that the selected cipher suite is supported and that no prior cipher suites in SUITES_U are supported.
- o Validate that there is a solution to the curve definition for the given x-coordinate G_X.
- o Pass UAD_1 to the application.

If any verification step fails, Party V MUST send an EDHOC error message back, formatted as defined in Section 6, and the protocol MUST be discontinued. If V does not support the selected cipher suite, then SUITES_V MUST include one or more supported cipher suites. If V does not support the selected cipher suite, but supports another cipher suite in SUITES_U, then SUITES_V MUST include the first supported cipher suite in SUITES_U.

4.3. EDHOC Message 2

4.3.1. Formatting of Message 2

message_2 and data_2 SHALL be CBOR Sequences (see Appendix A.1) as defined below

```
message_2 = (  
  data_2,  
  CIPHERTEXT_2 : bstr,  
)
```

```
data_2 = (  
  ? C_U : bstr,  
  G_Y : bstr,  
  C_V : bstr,  
)
```

where:

- o G_Y - the x-coordinate of the ephemeral public key of Party V
- o C_V - variable length connection identifier

4.3.2. Party V Processing of Message 2

Party V SHALL compose message_2 as follows:

- o If TYPE mod 4 equals 1 or 3, C_U is omitted, otherwise C_U is not omitted.

- o Generate an ephemeral ECDH key pair as specified in Section 5 of [SP-800-56A] using the curve in the selected cipher suite. Let `G_Y` be the x-coordinate of the ephemeral public key.
- o Choose a connection identifier `C_V` and store it for the length of the protocol.
- o Compute the transcript hash `TH_2 = H(message_1, data_2)` where `H()` is the hash function in the HMAC algorithm. The transcript hash `TH_2` is a CBOR encoded bstr and the input to the hash function is a CBOR Sequence.

- o Compute `COSE_Sign1` as defined in Section 4.4 of [RFC8152], using the signature algorithm in the selected cipher suite, the private authentication key of Party V, and the parameters below. Note that only 'signature' of the `COSE_Sign1` object is used to create `message_2`, see next bullet. The unprotected header (not included in the EDHOC message) MAY contain parameters (e.g. 'alg').

- * `protected = bstr .cbor ID_CRED_V`

- * `payload = CRED_V`

- * `external_aad = TH_2`

- * `ID_CRED_V` - identifier to facilitate retrieval of `CRED_V`, see Section 4.1

- * `CRED_V` - bstr credential containing the credential of Party V, e.g. its public authentication key or X.509 certificate see Section 4.1. The public key must be a signature key. Note that if objects that are not bstr are used, such as `COSE_Key` for public authentication keys, these objects must be wrapped in a CBOR bstr.

COSE constructs the input to the Signature Algorithm as follows:

- * The key is the private authentication key of V.

- * The message M to be signed is the CBOR encoding of:

- ["Signature1", << `ID_CRED_V` >>, `TH_2`, `CRED_V`]

- o Compute `COSE_Encrypt0` as defined in Section 5.3 of [RFC8152], with the AEAD algorithm in the selected cipher suite, `K_2`, `IV_2`, and the parameters below. Note that only 'ciphertext' of the `COSE_Encrypt0` object is used to create `message_2`, see next bullet. The protected header SHALL be empty. The unprotected header (not

included in the EDHOC message) MAY contain parameters (e.g. 'alg').

- * plaintext = (ID_CRED_V / kid_value, signature, ? UAD_2)
- * external_aad = TH_2
- * UAD_2 = bstr containing opaque unprotected application data

where signature is taken from the COSE_Sign1 object, ID_CRED_V is a COSE header_map (i.e. a CBOR map containing COSE Common Header Parameters, see [RFC8152]), and kid_value is a bstr. If ID_CRED_V contains a single 'kid' parameter, i.e., ID_CRED_V = { 4 : kid_value }, only kid_value is conveyed in the plaintext.

COSE constructs the input to the AEAD [RFC5116] as follows:

- * Key K = K_2
 - * Nonce N = IV_2
 - * Plaintext P = (ID_CRED_V / kid_value, signature, ? UAD_2)
 - * Associated data A = ["Encrypt0", h'', TH_2]
- o Encode message_2 as a sequence of CBOR encoded data items as specified in Section 4.3.1. CIPHERTEXT_2 is the COSE_Encrypt0 ciphertext.

4.3.3. Party U Processing of Message 2

Party U SHALL process message_2 as follows:

- o Decode message_2 (see Appendix A.1).
- o Retrieve the protocol state using the connection identifier C_U and/or other external information such as the CoAP Token and the 5-tuple.
- o Validate that there is a solution to the curve definition for the given x-coordinate G_Y.
- o Decrypt and verify COSE_Encrypt0 as defined in Section 5.3 of [RFC8152], with the AEAD algorithm in the selected cipher suite, K_2, and IV_2.

- o Verify COSE_Sign1 as defined in Section 4.4 of [RFC8152], using the signature algorithm in the selected cipher suite and the public authentication key of Party V.

If any verification step fails, Party U MUST send an EDHOC error message back, formatted as defined in Section 6, and the protocol MUST be discontinued.

4.4. EDHOC Message 3

4.4.1. Formatting of Message 3

message_3 and data_3 SHALL be CBOR Sequences (see Appendix A.1) as defined below

```
message_3 = (  
  data_3,  
  CIPHERTEXT_3 : bstr,  
)
```

```
data_3 = (  
  ? C_V : bstr,  
)
```

4.4.2. Party U Processing of Message 3

Party U SHALL compose message_3 as follows:

- o If TYPE mod 4 equals 2 or 3, C_V is omitted, otherwise C_V is not omitted.
- o Compute the transcript hash TH_3 = H(TH_2 , CIPHERTEXT_2, data_3) where H() is the hash function in the HMAC algorithm. The transcript hash TH_3 is a CBOR encoded bstr and the input to the hash function is a CBOR Sequence.
- o Compute COSE_Sign1 as defined in Section 4.4 of [RFC8152], using the signature algorithm in the selected cipher suite, the private authentication key of Party U, and the parameters below. Note that only 'signature' of the COSE_Sign1 object is used to create message_3, see next bullet. The unprotected header (not included in the EDHOC message) MAY contain parameters (e.g. 'alg').

* protected = bstr .cbor ID_CRED_U

* payload = CRED_U

* external_aad = TH_3

- * ID_CRED_U - identifier to facilitate retrieval of CRED_U, see Section 4.1
- * CRED_U - bstr credential containing the credential of Party U, e.g. its public authentication key or X.509 certificate see Section 4.1. The public key must be a signature key. Note that if objects that are not bstr are used, such as COSE_Key for public authentication keys, these objects must be wrapped in a CBOR bstr.

COSE constructs the input to the Signature Algorithm as follows:

- * The key is the private authentication key of U.
- * The message M to be signed is the CBOR encoding of:

["Signature1", << ID_CRED_U >>, TH_3, CRED_U]

- o Compute COSE_Encrypt0 as defined in Section 5.3 of [RFC8152], with the AEAD algorithm in the selected ciphersuite, K_3, and IV_3 and the parameters below. Note that only 'ciphertext' of the COSE_Encrypt0 object is used to create message_3, see next bullet. The protected header SHALL be empty. The unprotected header (not included in the EDHOC message) MAY contain parameters (e.g. 'alg').

- * plaintext = (ID_CRED_U / kid_value, signature, ? PAD_3)
- * external_aad = TH_3
- * PAD_3 = bstr containing opaque protected application data

where signature is taken from the COSE_Sign1 object, ID_CRED_U is a COSE header_map (i.e. a CBOR map containing COSE Common Header Parameters, see [RFC8152]), and kid_value is a bstr. If ID_CRED_U contains a single 'kid' parameter, i.e., ID_CRED_U = { 4 : kid_value }, only kid_value is conveyed in the plaintext.

COSE constructs the input to the AEAD [RFC5116] as follows:

- * Key K = K_3
- * Nonce N = IV_2
- * Plaintext P = (ID_CRED_U / kid_value, signature, ? PAD_3)
- * Associated data A = ["Encrypt0", h'', TH_3]

- o Encode message_3 as a sequence of CBOR encoded data items as specified in Section 4.4.1. CIPHERTEXT_3 is the COSE_Encrypt0 ciphertext.
- o Pass the connection identifiers (C_U, C_V) and the selected cipher suite to the application. The application can now derive application keys using the EDHOC-Exporter interface.

4.4.3. Party V Processing of Message 3

Party V SHALL process message_3 as follows:

- o Decode message_3 (see Appendix A.1).
- o Retrieve the protocol state using the connection identifier C_V and/or other external information such as the CoAP Token and the 5-tuple.
- o Decrypt and verify COSE_Encrypt0 as defined in Section 5.3 of [RFC8152], with the AEAD algorithm in the selected cipher suite, K_3, and IV_3.
- o Verify COSE_Sign1 as defined in Section 4.4 of [RFC8152], using the signature algorithm in the selected cipher suite and the public authentication key of Party U.

If any verification step fails, Party V MUST send an EDHOC error message back, formatted as defined in Section 6, and the protocol MUST be discontinued.

- o Pass PAD_3, the connection identifiers (C_U, C_V), and the selected cipher suite to the application. The application can now derive application keys using the EDHOC-Exporter interface.

5. EDHOC Authenticated with Symmetric Keys

5.1. Overview

EDHOC supports authentication with pre-shared keys. Party U and V are assumed to have a pre-shared key (PSK) with a good amount of randomness and the requirement that:

- o Only Party U and Party V SHALL have access to the PSK,
- o Party V is able to retrieve the PSK using ID_PSK.

where the identifier ID_PSK is a COSE header_map (i.e. a CBOR map containing COSE Common Header Parameters, see [RFC8152]) containing

COSE header parameter that can identify a pre-shared key. Pre-shared keys are typically stored as COSE_Key objects and identified with a 'kid' parameter (see [RFC8152]):

o ID_PSK = { 4 : kid_value } , where kid_value : bstr

The purpose of ID_PSK is to facilitate retrieval of the PSK and in the case a 'kid' parameter is used it may be very short. It is RECOMMENDED that it uniquely identify the PSK as the recipient may otherwise have to try several keys.

EDHOC with symmetric key authentication is illustrated in Figure 5.

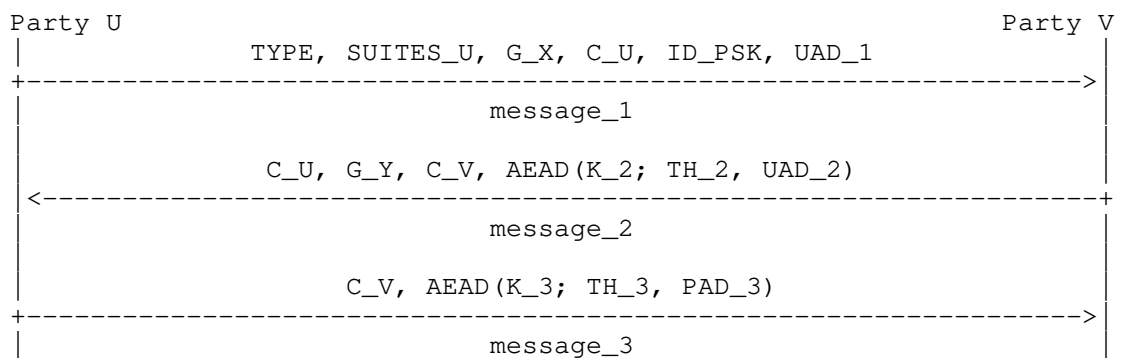


Figure 5: Overview of EDHOC with symmetric key authentication.

EDHOC with symmetric key authentication is very similar to EDHOC with asymmetric key authentication. In the following subsections the differences compared to EDHOC with asymmetric key authentication are described.

5.2. EDHOC Message 1

5.2.1. Formatting of Message 1

message_1 SHALL be a CBOR Sequence (see Appendix A.1) as defined below

```

message_1 = (
  TYPE : int,
  SUITES_U : suite / [ index : uint, 2* suite ],
  G_X : bstr,
  C_U : bstr,
  ID_PSK : header_map // kid_value : bstr,
  ? UAD_1 : bstr,
)
  
```


where:

- o `TYPE = 4 * method + corr`, where the `method = 1` and the connection parameter `corr` is chosen based on the transport and determines which connection identifiers that are omitted (see Section 4.1).
- o `ID_PSK` - identifier to facilitate retrieval of the pre-shared key. If `ID_PSK` contains a single 'kid' parameter, i.e., `ID_PSK = { 4 : kid_value }`, with `kid_value: bstr`, only `kid_value` is conveyed.

5.3. EDHOC Message 2

5.3.1. Processing of Message 2

- o `COSE_Sign1` is not used.
- o `COSE_Encrypt0` is computed as defined in Section 5.3 of [RFC8152], with the AEAD algorithm in the selected cipher suite, `K_2`, `IV_2`, and the following parameters. The protected header SHALL be empty. The unprotected header MAY contain parameters (e.g. 'alg').
 - * `external_aad = TH_2`
 - * `plaintext = ? UAD_2`
 - * `UAD_2 = bstr` containing opaque unprotected application data

5.4. EDHOC Message 3

5.4.1. Processing of Message 3

- o `COSE_Sign1` is not used.
- o `COSE_Encrypt0` is computed as defined in Section 5.3 of [RFC8152], with the AEAD algorithm in the selected cipher suite, `K_3`, `IV_3`, and the following parameters. The protected header SHALL be empty. The unprotected header MAY contain parameters (e.g. 'alg').
 - * `external_aad = TH_3`
 - * `plaintext = ? PAD_3`
 - * `PAD_3 = bstr` containing opaque protected application data

6. Error Handling

6.1. EDHOC Error Message

This section defines a message format for the EDHOC error message, used during the protocol. An EDHOC error message can be sent by both parties as a reply to any non-error EDHOC message. After sending an error message, the protocol MUST be discontinued. Errors at the EDHOC layer are sent as normal successful messages in the lower layers (e.g. CoAP POST and 2.04 Changed). An advantage of using such a construction is to avoid issues created by usage of cross protocol proxies (e.g. UDP to TCP).

error SHALL be a CBOR Sequence (see Appendix A.1) as defined below

```
error = (  
  ? C_x : bstr,  
  ERR_MSG : tstr,  
  ? SUITES_V : suite / [ 2* suite ],  
)
```

where:

- o C_x - if error is sent by Party V and TYPE mod 4 equals 0 or 2 then C_x is set to C_U, else if error is sent by Party U and TYPE mod 4 equals 0 or 1 then C_x is set to C_V, else C_x is omitted.
- o ERR_MSG - text string containing the diagnostic payload, defined in the same way as in Section 5.5.2 of [RFC7252]. ERR_MSG MAY be a 0-length text string.
- o SUITES_V - cipher suites from SUITES_U or the EDHOC cipher suites registry that V supports. Note that SUITES_V only contains the values from the EDHOC cipher suites registry and no index. SUITES_V MUST only be included in replies to message_1.

6.1.1. Example Use of EDHOC Error Message with SUITES_V

Assuming that Party U supports the five cipher suites {5, 6, 7, 8, 9} in decreasing order of preference, Figures 6 and 7 show examples of how Party U can truncate SUITES_U and how SUITES_V is used by Party V to give Party U information about the cipher suites that Party V supports. In Figure 6, Party V supports cipher suite 6 but not the selected cipher suite 5.

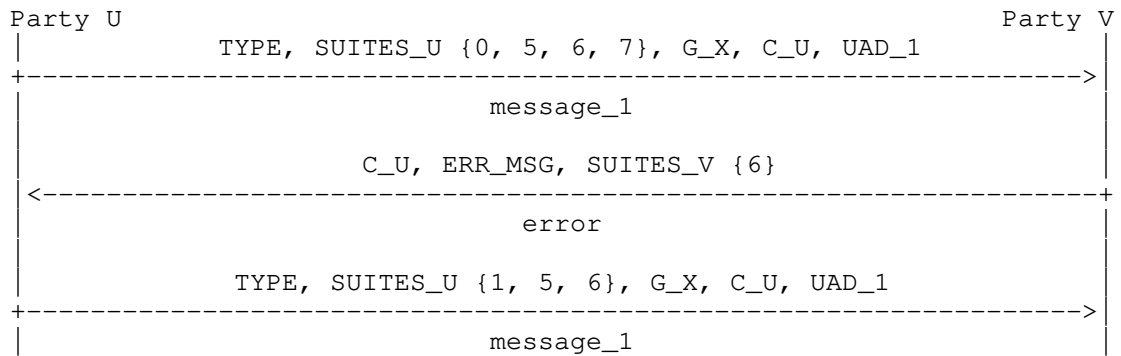


Figure 6: Example use of error message with SUITES_V.

In Figure 7, Party V supports cipher suite 7 but not cipher suites 5 and 6.

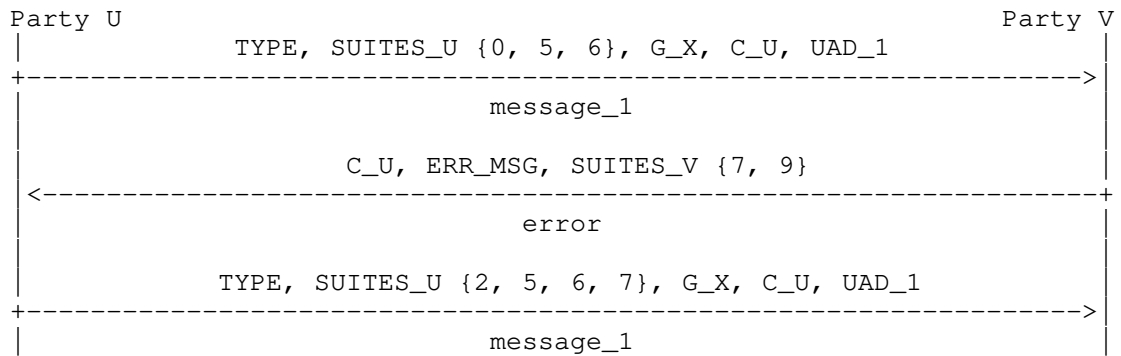


Figure 7: Example use of error message with SUITES_V.

As Party U's list of supported cipher suites and order of preference is fixed, and Party V only accepts message_1 if the selected cipher suite is the first cipher suite in SUITES_U that Party V supports, the parties can verify that the selected cipher suite is the most preferred (by Party U) cipher suite supported by both parties. If the selected cipher suite is not the first cipher suite in SUITES_U that Party V supports, Party V will discontinue the protocol.

7. Transferring EDHOC and Deriving Application Keys

7.1. Transferring EDHOC in CoAP

It is recommended to transport EDHOC as an exchange of CoAP [RFC7252] messages. CoAP is a reliable transport that can preserve packet ordering and handle message duplication. CoAP can also perform

fragmentation and protect against denial of service attacks. It is recommended to carry the EDHOC flights in Confirmable messages, especially if fragmentation is used.

By default, the CoAP client is Party U and the CoAP server is Party V, but the roles SHOULD be chosen to protect the most sensitive identity, see Section 8. By default, EDHOC is transferred in POST requests and 2.04 (Changed) responses to the Uri-Path: `"/.well-known/edhoc"`, but an application may define its own path that can be discovered e.g. using resource directory [I-D.ietf-core-resource-directory].

By default, the message flow is as follows: EDHOC message_1 is sent in the payload of a POST request from the client to the server's resource for EDHOC. EDHOC message_2 or the EDHOC error message is sent from the server to the client in the payload of a 2.04 (Changed) response. EDHOC message_3 or the EDHOC error message is sent from the client to the server's resource in the payload of a POST request. If needed, an EDHOC error message is sent from the server to the client in the payload of a 2.04 (Changed) response.

An example of a successful EDHOC exchange using CoAP is shown in Figure 8. In this case the CoAP Token enables Party U to correlate message_1 and message_2 so the correlation parameter `corr = 1`.

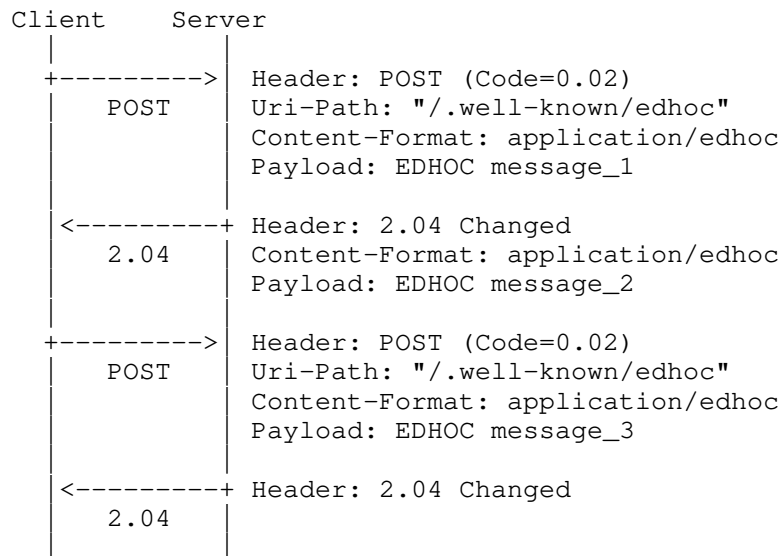


Figure 8: Transferring EDHOC in CoAP

The exchange in Figure 8 protects the client identity against active attackers and the server identity against passive attackers. An alternative exchange that protects the server identity against active attackers and the client identity against passive attackers is shown in Figure 9. In this case the CoAP Token enables Party V to correlate message_2 and message_3 so the correlation parameter corr = 2.

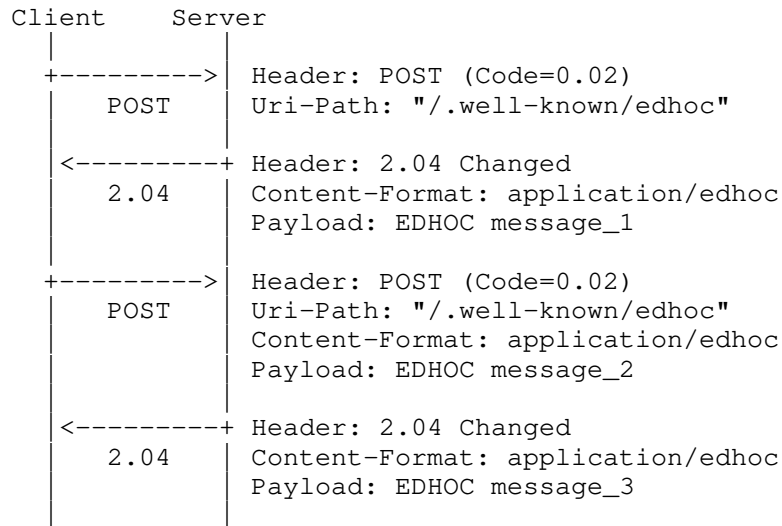


Figure 9: Transferring EDHOC in CoAP

To protect against denial-of-service attacks, the CoAP server MAY respond to the first POST request with a 4.01 (Unauthorized) containing an Echo option [I-D.ietf-core-echo-request-tag]. This forces the initiator to demonstrate its reachability at its apparent network address. If message fragmentation is needed, the EDHOC messages may be fragmented using the CoAP Block-Wise Transfer mechanism [RFC7959].

7.1.1. Deriving an OSCORE Context from EDHOC

When EDHOC is used to derive parameters for OSCORE [RFC8613], the parties must make sure that the EDHOC connection identifiers are unique, i.e. C_V MUST NOT be equal to C_U. The CoAP client and server MUST be able to retrieve the OSCORE protocol state using its chosen connection identifier and optionally other information such as the 5-tuple. In case that the CoAP client is party U and the CoAP server is party V:

- o The client's OSCORE Sender ID is C_V and the server's OSCORE Sender ID is C_U, as defined in this document
- o The AEAD Algorithm and the HMAC algorithms are the AEAD and HMAC algorithms in the selected cipher suite.
- o The Master Secret and Master Salt are derived as follows where length is the key length (in bytes) of the AEAD Algorithm.

```
Master Secret = EDHOC-Exporter( "OSCORE Master Secret", length )
Master Salt   = EDHOC-Exporter( "OSCORE Master Salt", 8 )
```

7.2. Transferring EDHOC over Other Protocols

EDHOC may be transported over a different transport than CoAP. In this case the lower layers need to handle message loss, reordering, message duplication, fragmentation, and denial of service protection.

8. Security Considerations

8.1. Security Properties

EDHOC inherits its security properties from the theoretical SIGMA-I protocol [SIGMA]. Using the terminology from [SIGMA], EDHOC provides perfect forward secrecy, mutual authentication with aliveness, consistency, peer awareness, and identity protection. As described in [SIGMA], peer awareness is provided to Party V, but not to Party U. EDHOC also inherits Key Compromise Impersonation (KCI) resistance from SIGMA-I.

EDHOC with asymmetric authentication offers identity protection of Party U against active attacks and identity protection of Party V against passive attacks. The roles should be assigned to protect the most sensitive identity, typically that which is not possible to infer from routing information in the lower layers.

Compared to [SIGMA], EDHOC adds an explicit method type and expands the message authentication coverage to additional elements such as algorithms, application data, and previous messages. This protects against an attacker replaying messages or injecting messages from another session.

EDHOC also adds negotiation of connection identifiers and downgrade protected negotiation of cryptographic parameters, i.e. an attacker cannot affect the negotiated parameters. A single session of EDHOC does not include negotiation of cipher suites, but it enables Party V to verify that the selected cipher suite is the most preferred cipher suite by U which is supported by both U and V.

As required by [RFC7258], IETF protocols need to mitigate pervasive monitoring when possible. One way to mitigate pervasive monitoring is to use a key exchange that provides perfect forward secrecy. EDHOC therefore only supports methods with perfect forward secrecy. To limit the effect of breaches, it is important to limit the use of symmetrical group keys for bootstrapping. EDHOC therefore strives to make the additional cost of using raw public keys and self-signed certificates as small as possible. Raw public keys and self-signed certificates are not a replacement for a public key infrastructure, but SHOULD be used instead of symmetrical group keys for bootstrapping.

Compromise of the long-term keys (PSK or private authentication keys) does not compromise the security of completed EDHOC exchanges. Compromising the private authentication keys of one party lets the attacker impersonate that compromised party in EDHOC exchanges with other parties, but does not let the attacker impersonate other parties in EDHOC exchanges with the compromised party. Compromising the PSK lets the attacker impersonate Party U in EDHOC exchanges with Party V and impersonate Party V in EDHOC exchanges with Party U. Compromise of the HDKF input parameters (ECDH shared secret and/or PSK) leads to compromise of all session keys derived from that compromised shared secret. Compromise of one session key does not compromise other session keys.

8.2. Cryptographic Considerations

The security of the SIGMA protocol requires the MAC to be bound to the identity of the signer. Hence the message authenticating functionality of the authenticated encryption in EDHOC is critical: authenticated encryption MUST NOT be replaced by plain encryption only, even if authentication is provided at another level or through a different mechanism. EDHOC implements SIGMA-I using the same Sign-then-MAC approach as TLS 1.3.

To reduce message overhead EDHOC does not use explicit nonces and instead rely on the ephemeral public keys to provide randomness to each session. A good amount of randomness is important for the key generation, to provide liveness, and to protect against interleaving attacks. For this reason, the ephemeral keys MUST NOT be reused, and both parties SHALL generate fresh random ephemeral key pairs.

The choice of key length used in the different algorithms needs to be harmonized, so that a sufficient security level is maintained for certificates, EDHOC, and the protection of application data. Party U and V should enforce a minimum security level.

The data rates in many IoT deployments are very limited. Given that the application keys are protected as well as the long-term authentication keys they can often be used for years or even decades before the cryptographic limits are reached. If the application keys established through EDHOC need to be renewed, the communicating parties can derive application keys with other labels or run EDHOC again.

8.3. Cipher Suites

Cipher suite number 0 (AES-CCM-64-64-128, ECDH-SS + HKDF-256, X25519, Ed25519) is mandatory to implement. For many constrained IoT devices it is problematic to support more than one cipher suites, so some deployments with P-256 may not support the mandatory cipher suite. This is not a problem for local deployments.

The HMAC algorithm HMAC 256/64 (HMAC w/ SHA-256 truncated to 64 bits) SHALL NOT be supported for use in EDHOC.

8.4. Unprotected Data

Party U and V must make sure that unprotected data and metadata do not reveal any sensitive information. This also applies for encrypted data sent to an unauthenticated party. In particular, it applies to UAD_1, ID_CRED_V, UAD_2, and ERR_MSG in the asymmetric case, and ID_PSK, UAD_1, and ERR_MSG in the symmetric case. Using the same ID_PSK or UAD_1 in several EDHOC sessions allows passive eavesdroppers to correlate the different sessions. The communicating parties may therefore anonymize ID_PSK. Another consideration is that the list of supported cipher suites may be used to identify the application.

Party U and V must also make sure that unauthenticated data does not trigger any harmful actions. In particular, this applies to UAD_1 and ERR_MSG in the asymmetric case, and ID_PSK, UAD_1, and ERR_MSG in the symmetric case.

8.5. Denial-of-Service

EDHOC itself does not provide countermeasures against Denial-of-Service attacks. By sending a number of new or replayed message_1 an attacker may cause Party V to allocate state, perform cryptographic operations, and amplify messages. To mitigate such attacks, an implementation SHOULD rely on lower layer mechanisms such as the Echo option in CoAP [I-D.ietf-core-echo-request-tag] that forces the initiator to demonstrate reachability at its apparent network address.

8.6. Implementation Considerations

The availability of a secure pseudorandom number generator and truly random seeds are essential for the security of EDHOC. If no true random number generator is available, a truly random seed must be provided from an external source. As each pseudorandom number must only be used once, an implementation need to get a new truly random seed after reboot, or continuously store state in nonvolatile memory, see ([RFC8613], Appendix B.1.1) for issues and solution approaches for writing to nonvolatile memory. If ECDSA is supported, "deterministic ECDSA" as specified in [RFC6979] is RECOMMENDED.

The referenced processing instructions in [SP-800-56A] must be complied with, including deleting the intermediate computed values along with any ephemeral ECDH secrets after the key derivation is completed. The ECDH shared secret, keys (K₂, K₃), and IVs (IV₂, IV₃) MUST be secret. Implementations should provide countermeasures to side-channel attacks such as timing attacks.

Party U and V are responsible for verifying the integrity of certificates. The selection of trusted CAs should be done very carefully and certificate revocation should be supported. The private authentication keys and the PSK (even though it is used as salt) MUST be kept secret.

Party U and V are allowed to select the connection identifiers C_U and C_V, respectively, for the other party to use in the ongoing EDHOC protocol as well as in a subsequent application protocol (e.g. OSCORE [RFC8613]). The choice of connection identifier is not security critical in EDHOC but intended to simplify the retrieval of the right security context in combination with using short identifiers. If the wrong connection identifier of the other party is used in a protocol message it will result in the receiving party not being able to retrieve a security context (which will terminate the protocol) or retrieve the wrong security context (which also terminates the protocol as the message cannot be verified).

Party V MUST finish the verification step of message₃ before passing PAD₃ to the application.

If two nodes unintentionally initiate two simultaneous EDHOC message exchanges with each other even if they only want to complete a single EDHOC message exchange, they MAY terminate the exchange with the lexicographically smallest G_X. If the two G_X values are equal, the received message₁ MUST be discarded to mitigate reflection attacks. Note that in the case of two simultaneous EDHOC exchanges where the nodes only complete one and where the nodes have different preferred

cipher suites, an attacker can affect which of the two nodes' preferred cipher suites will be used by blocking the other exchange.

8.7. Other Documents Referencing EDHOC

EDHOC has been analyzed in several other documents. A formal verification of EDHOC was done in [SSR18], an analysis of EDHOC for certificate enrollment was done in [Kron18], the use of EDHOC in LoRaWAN is analyzed in [LoRa1] and [LoRa2], the use of EDHOC in IoT bootstrapping is analyzed in [Perez18], and the use of EDHOC in 6TiSCH is described in [I-D.ietf-6tisch-dtsecurity-zerotouch-join].

9. IANA Considerations

9.1. EDHOC Cipher Suites Registry

IANA has created a new registry titled "EDHOC Cipher Suites" under the new heading "EDHOC". The registration procedure is "Expert Review". The columns of the registry are Value, Array, Description, and Reference, where Value is an integer and the other columns are text strings. The initial contents of the registry are:

Value: 1
Array: [10, 5, 1, -7, 1]
Desc: AES-CCM-16-64-128, HMAC 256/256, P-256, ES256, P-256
Reference: [[this document]]

Value: 0
Array: [10, 5, 4, -8, 6]
Desc: AES-CCM-16-64-128, HMAC 256/256, X25519, EdDSA, Ed25519
Reference: [[this document]]

Value: -5
Array:
Desc: Reserved for Private Use
Reference: [[this document]]

Value: -6
Array:
Desc: Reserved for Private Use
Reference: [[this document]]

9.2. EDHOC Method Type Registry

IANA has created a new registry titled "EDHOC Method Type" under the new heading "EDHOC". The registration procedure is "Expert Review". The columns of the registry are Value, Description, and Reference,

where Value is an integer and the other columns are text strings.
The initial contents of the registry are:

Value	Specification	Reference
0	EDHOC Authenticated with Asymmetric Keys	[[this document]]
1	EDHOC Authenticated with Symmetric Keys	[[this document]]

9.3. The Well-Known URI Registry

IANA has added the well-known URI 'edhoc' to the Well-Known URIs registry.

- o URI suffix: edhoc
- o Change controller: IETF
- o Specification document(s): [[this document]]
- o Related information: None

9.4. Media Types Registry

IANA has added the media type 'application/edhoc' to the Media Types registry.

- o Type name: application
- o Subtype name: edhoc
- o Required parameters: N/A
- o Optional parameters: N/A
- o Encoding considerations: binary
- o Security considerations: See Section 7 of this document.
- o Interoperability considerations: N/A
- o Published specification: [[this document]] (this document)
- o Applications that use this media type: To be identified
- o Fragment identifier considerations: N/A

- o Additional information:
 - * Magic number(s): N/A
 - * File extension(s): N/A
 - * Macintosh file type code(s): N/A
- o Person & email address to contact for further information: See "Authors' Addresses" section.
- o Intended usage: COMMON
- o Restrictions on usage: N/A
- o Author: See "Authors' Addresses" section.
- o Change Controller: IESG

9.5. CoAP Content-Formats Registry

IANA has added the media type 'application/edhoc' to the CoAP Content-Formats registry.

- o Media Type: application/edhoc
- o Encoding:
- o ID: TBD42
- o Reference: [[this document]]

9.6. Expert Review Instructions

The IANA Registries established in this document is 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 Clarity and correctness of registrations. Experts are expected to check the clarity of purpose and use of the requested entries. Expert needs to make sure the values of algorithms are taken from the right registry, when that's required. Expert should consider requesting an opinion on the correctness of registered parameters from relevant IETF working groups. Encodings that do not meet

these objective of clarity and completeness should not be registered.

- o Experts should take into account the expected usage of fields when approving point assignment. 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.
- o Specifications are recommended. When specifications are not provided, the description provided needs to have sufficient information to verify the points above.

10. References

10.1. Normative References

- [I-D.ietf-cbor-7049bis]
Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", draft-ietf-cbor-7049bis-07 (work in progress), August 2019.
- [I-D.ietf-cbor-sequence]
Bormann, C., "Concise Binary Object Representation (CBOR) Sequences", draft-ietf-cbor-sequence-01 (work in progress), August 2019.
- [I-D.ietf-core-echo-request-tag]
Amsuess, C., Mattsson, J., and G. Selander, "CoAP: Echo, Request-Tag, and Token Processing", draft-ietf-core-echo-request-tag-05 (work in progress), May 2019.
- [I-D.ietf-cose-x509]
Schaad, J., "CBOR Object Signing and Encryption (COSE): Headers for carrying and referencing X.509 certificates", draft-ietf-cose-x509-03 (work in progress), August 2019.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <<https://www.rfc-editor.org/info/rfc5116>>.

- [RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <<https://www.rfc-editor.org/info/rfc5869>>.
- [RFC6090] McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms", RFC 6090, DOI 10.17487/RFC6090, February 2011, <<https://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, <<https://www.rfc-editor.org/info/rfc6979>>.
- [RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <<https://www.rfc-editor.org/info/rfc7252>>.
- [RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <<https://www.rfc-editor.org/info/rfc7748>>.
- [RFC7959] Bormann, C. and Z. Shelby, Ed., "Block-Wise Transfers in the Constrained Application Protocol (CoAP)", RFC 7959, DOI 10.17487/RFC7959, August 2016, <<https://www.rfc-editor.org/info/rfc7959>>.
- [RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <<https://www.rfc-editor.org/info/rfc8152>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <<https://www.rfc-editor.org/info/rfc8610>>.
- [RFC8613] Selander, G., Mattsson, J., Palombini, F., and L. Seitz, "Object Security for Constrained RESTful Environments (OSCORE)", RFC 8613, DOI 10.17487/RFC8613, July 2019, <<https://www.rfc-editor.org/info/rfc8613>>.

[SIGMA] Krawczyk, H., "SIGMA - The 'SIGn-and-MAC' Approach to Authenticated Diffie-Hellman and Its Use in the IKE-Protocols (Long version)", June 2003, <<http://webee.technion.ac.il/~hugo/sigma-pdf.pdf>>.

[SP-800-56A] Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography", NIST Special Publication 800-56A Revision 3, April 2018, <<http://doi.org/10.6028/NIST.SP.800-56Ar3>>.

10.2. Informative References

[CborMe] Bormann, C., "CBOR Playground", May 2018, <<http://cbor.me/>>.

[I-D.hartke-core-e2e-security-reqs] Selander, G., Palombini, F., and K. Hartke, "Requirements for CoAP End-To-End Security", draft-hartke-core-e2e-security-reqs-03 (work in progress), July 2017.

[I-D.ietf-6tisch-dtsecurity-zerotouch-join] Richardson, M., "6tisch Zero-Touch Secure Join protocol", draft-ietf-6tisch-dtsecurity-zerotouch-join-04 (work in progress), July 2019.

[I-D.ietf-ace-oauth-authz] Seitz, L., Selander, G., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework (ACE-OAuth)", draft-ietf-ace-oauth-authz-24 (work in progress), March 2019.

[I-D.ietf-ace-oscore-profile] Palombini, F., Seitz, L., Selander, G., and M. Gunnarsson, "OSCORE profile of the Authentication and Authorization for Constrained Environments Framework", draft-ietf-ace-oscore-profile-08 (work in progress), July 2019.

[I-D.ietf-core-resource-directory] Shelby, Z., Koster, M., Bormann, C., Stok, P., and C. Amsuess, "CoRE Resource Directory", draft-ietf-core-resource-directory-23 (work in progress), July 2019.

- [I-D.ietf-lwig-security-protocol-comparison]
Mattsson, J. and F. Palombini, "Comparison of CoAP Security Protocols", draft-ietf-lwig-security-protocol-comparison-03 (work in progress), March 2019.
- [I-D.ietf-tls-dtls13]
Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", draft-ietf-tls-dtls13-32 (work in progress), July 2019.
- [Kron18] Krontiris, A., "Evaluation of Certificate Enrollment over Application Layer Security", May 2018, <https://www.nada.kth.se/~ann/exjobb/alexandros_krontiris.pdf>.
- [LoRa1] Sanchez-Iborra, R., Sanchez-Gomez, J., Perez, S., Fernandez, P., Santa, J., Hernandez-Ramos, J., and A. Skarmeta, "Enhancing LoRaWAN Security through a Lightweight and Authenticated Key Management Approach", June 2018, <<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6021899/pdf/sensors-18-01833.pdf>>.
- [LoRa2] Sanchez-Iborra, R., Sanchez-Gomez, J., Perez, S., Fernandez, P., Santa, J., Hernandez-Ramos, J., and A. Skarmeta, "Internet Access for LoRaWAN Devices Considering Security Issues", June 2018, <<https://ants.inf.um.es/~josesanta/doc/GIoTSl.pdf>>.
- [OPTLS] Krawczyk, H. and H. Wee, "The OPTLS Protocol and TLS 1.3", October 2015, <<https://eprint.iacr.org/2015/978.pdf>>.
- [Perez18] Perez, S., Garcia-Carrillo, D., Marin-Lopez, R., Hernandez-Ramos, J., Marin-Perez, R., and A. Skarmeta, "Architecture of security association establishment based on bootstrapping technologies for enabling critical IoT infrastructures", October 2018, <http://www.anastacia-h2020.eu/publications/Architecture_of_security_association_establishment_based_on_bootstrapping_technologies_for_enabling_critical_IoT_infrastructures.pdf>.
- [RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <<https://www.rfc-editor.org/info/rfc7228>>.

- [RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an Attack", BCP 188, RFC 7258, DOI 10.17487/RFC7258, May 2014, <<https://www.rfc-editor.org/info/rfc7258>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.
- [SSR18] Bruni, A., Sahl Joergensen, T., Groenbech Petersen, T., and C. Schuermann, "Formal Verification of Ephemeral Diffie-Hellman Over COSE (EDHOC)", November 2018, <<https://www.springerprofessional.de/en/formal-verification-of-ephemeral-diffie-hellman-over-cose-edhoc/16284348>>.

Appendix A. Use of CBOR, CDDL and COSE in EDHOC

This Appendix is intended to simplify for implementors not familiar with CBOR [I-D.ietf-cbor-7049bis], CDDL [RFC8610], COSE [RFC8152], and HKDF [RFC5869].

A.1. CBOR and CDDL

The Concise Binary Object Representation (CBOR) [I-D.ietf-cbor-7049bis] is a data format designed for small code size and small message size. CBOR builds on the JSON data model but extends it by e.g. encoding binary data directly without base64 conversion. In addition to the binary CBOR encoding, CBOR also has a diagnostic notation that is readable and editable by humans. The Concise Data Definition Language (CDDL) [RFC8610] provides a way to express structures for protocol messages and APIs that use CBOR. [RFC8610] also extends the diagnostic notation.

CBOR data items are encoded to or decoded from byte strings using a type-length-value encoding scheme, where the three highest order bits of the initial byte contain information about the major type. CBOR supports several different types of data items, in addition to integers (int, uint), simple values (e.g. null), byte strings (bstr), and text strings (tstr), CBOR also supports arrays [] of data items, maps {} of pairs of data items, and sequences [I-D.ietf-cbor-sequence] of data items. Some examples are given below. For a complete specification and more examples, see [I-D.ietf-cbor-7049bis] and [RFC8610]. We recommend implementors to get used to CBOR by using the CBOR playground [CborMe].

Diagnostic	Encoded	Type
1	0x01	unsigned integer
24	0x1818	unsigned integer
-24	0x37	negative integer
-25	0x3818	negative integer
null	0xf6	simple value
h'12cd'	0x4212cd	byte string
'12cd'	0x4431326364	byte string
"12cd"	0x6431326364	text string
{ 4 : h'cd' }	0xa10441cd	map
<< 1, 2, null >>	0x430102f6	byte string
[1, 2, null]	0x830102f6	array
(1, 2, null)	0x0102f6	sequence
1, 2, null	0x0102f6	sequence

EDHOC messages are CBOR Sequences [I-D.ietf-cbor-sequence]. The message format specification uses the construct `'.cbor'` enabling conversion between different CDDL types matching different CBOR items with different encodings. Some examples are given below.

A type (e.g. an uint) may be wrapped in a byte string (bstr):

CDDL Type	Diagnostic	Encoded
uint	24	0x1818
bstr .cbor uint	<< 24 >>	0x421818

A.2. COSE

CBOR Object Signing and Encryption (COSE) [RFC8152] describes how to create and process signatures, message authentication codes, and encryption using CBOR. COSE builds on JOSE, but is adapted to allow more efficient processing in constrained devices. EDHOC makes use of COSE_Key, COSE_Encrypt0, COSE_Sign1, and COSE_KDF_Context objects.

Appendix B. EDHOC Authenticated with Diffie-Hellman Keys

The SIGMA protocol is mainly optimized for PKI and certificates. The OPTLS protocol [OPTLS] shows how authentication can be provided by a MAC computed from an ephemeral-static ECDH shared secret. Instead of signature authentication keys, U and V would have Diffie-Hellman authentication keys G_U and G_V, respectively. This type of authentication keys could easily be used with RPK and would provide significant reductions in message sizes as the 64 bytes signature would be replaced by an 8 bytes MAC.

EDHOC authenticated with asymmetric Diffie-Hellman keys should have similar security properties as EDHOC authenticated with asymmetric signature keys with a few differences:

- o Repudiation: In EDHOC authenticated with asymmetric signature keys, Party U could theoretically prove that Party V performed a run of the protocol by presenting the private ephemeral key, and vice versa. Note that storing the private ephemeral keys violates the protocol requirements. With asymmetric Diffie-Hellman key authentication, both parties can always deny having participated in the protocol, this is similar to EDHOC with symmetric key authentication.
- o Key compromise impersonation (KCI): In EDHOC authenticated with asymmetric signature keys, EDHOC provides KCI protection against an attacker having access to the long term key or the ephemeral secret key. In EDHOC authenticated with symmetric keys, EDHOC provides KCI protection against an attacker having access to the ephemeral secret key, but not against an attacker having access to the long-term PSK. With asymmetric Diffie-Hellman key authentication, KCI protection would be provided against an attacker having access to the long-term Diffie-Hellman key, but not to an attacker having access to the ephemeral secret key. Note that the term KCI has typically been used for compromise of long-term keys, and that an attacker with access to the ephemeral secret key can only attack that specific protocol run.

TODO: Initial suggestion for key derivation, message formats, and processing

Appendix C. Test Vectors

This appendix provides detailed test vectors to ease implementation and ensure interoperability. In addition to hexadecimal, all CBOR data items and sequences are given in CBOR diagnostic notation. The test vectors use 1 byte key identifiers, 1 byte connection IDs, and the default mapping to CoAP where Party U is CoAP client (this means that `corr = 1`).

C.1. Test Vectors for EDHOC Authenticated with Asymmetric Keys (RPK)

Asymmetric EDHOC is used:

```
method (Asymmetric Authentication)
0
```

CoAP is used as transport:

corr (Party U is CoAP client)
1

No unprotected opaque application data is sent in the message exchanges.

The pre-defined Cipher Suite 0 is in place both on Party U and Party V, see Section 3.1.

C.1.1.1. Input for Party U

The following are the parameters that are set in Party U before the first message exchange.

Party U's private authentication key (32 bytes)

53 21 fc 01 c2 98 20 06 3a 72 50 8f c6 39 25 1d c8 30 e2 f7 68 3e b8 e3 8a
f1 64 a5 b9 af 9b e3

Party U's public authentication key (32 bytes)

42 4c 75 6a b7 7c c6 fd ec f0 b3 ec fc ff b7 53 10 c0 15 bf 5c ba 2e c0 a2
36 e6 65 0c 8a b9 c7

kid value to identify U's public authentication key (1 bytes)
a2

This test vector uses COSE_Key objects to store the raw public keys. Moreover, EC2 keys with curve Ed25519 are used. That is in agreement with the Cipher Suite 0.

CRED_U =

```
<< {  
  1:  1,  
 -1:  6,  
 -2:  h'424c756ab77cc6fdecf0b3ecfcffb75310c015bf5cba2ec0a236e6650c8ab9c7'  
}>>
```

CRED_U (COSE_Key) (CBOR-encoded) (42 bytes)

58 28 a3 01 01 20 06 21 58 20 42 4c 75 6a b7 7c c6 fd ec f0 b3 ec fc ff b7
53 10 c0 15 bf 5c ba 2e c0 a2 36 e6 65 0c 8a b9 c7

Because COSE_Keys are used, and because kid = h'a2':

```
ID_CRED_U =  
{  
  4:  h'a2'  
}
```

Note that since the map for ID_CRED_U contains a single 'kid' parameter, ID_CRED_U is used when transported in the protected header of the COSE Object, but only the kid_value is used when added to the plaintext (see Section 4.4.2):

ID_CRED_U (in protected header) (CBOR-encoded) (4 bytes)
a1 04 41 a2

kid_value (in plaintext) (CBOR-encoded) (2 bytes)
41 a2

C.1.2. Input for Party V

The following are the parameters that are set in Party V before the first message exchange.

Party V's private authentication key (32 bytes)

74 56 b3 a3 e5 8d 8d 26 dd 36 bc 75 d5 5b 88 63 a8 5d 34 72 f4 a0 1f 02 24
62 1b 1c b8 16 6d a9

Party V's public authentication key (32 bytes)

1b 66 1e e5 d5 ef 16 72 a2 d8 77 cd 5b c2 0f 46 30 dc 78 a1 14 de 65 9c 7e
50 4d 0f 52 9a 6b d3

kid value to identify U's public authentication key (1 bytes)
a3

This test vector uses COSE_Key objects to store the raw public keys. Moreover, EC2 keys with curve Ed25519 are used. That is in agreement with the Cipher Suite 0.

CRED_V =

```
<< {  
  1:  1,  
 -1:  6,  
 -2:  h'1b661ee5d5ef1672a2d877cd5bc20f4630dc78a114de659c7e504d0f529a6bd3'  
}>>
```

CRED_V (COSE_Key) (CBOR-encoded) (42 bytes)

58 28 a3 01 01 20 06 21 58 20 1b 66 1e e5 d5 ef 16 72 a2 d8 77 cd 5b c2 0f
46 30 dc 78 a1 14 de 65 9c 7e 50 4d 0f 52 9a 6b d3

Because COSE_Keys are used, and because kid = h'a3':

```
ID_CRED_V =  
{  
  4:  h'a3'  
}
```

Note that since the map for ID_CRED_U contains a single 'kid' parameter, ID_CRED_U is used when transported in the protected header of the COSE Object, but only the kid_value is used when added to the plaintext (see Section 4.4.2):

ID_CRED_V (in protected header) (CBOR-encoded) (4 bytes)
a1 04 41 a3

kid_value (in plaintext) (CBOR-encoded) (2 bytes)
41 a3

C.1.3. Message 1

From the input parameters (in Appendix C.1.1):

TYPE (4 * method + corr)
1

suite
0

SUITES_U : suite
0

G_X (X-coordinate of the ephemeral public key of Party U) (32 bytes)
b1 a3 e8 94 60 e8 8d 3a 8d 54 21 1d c9 5f 0b 90 3f f2 05 eb 71 91 2d 6d b8
f4 af 98 0d 2d b8 3a

C_U (Connection identifier chosen by U) (1 bytes)
c3

No UAD_1 is provided, so UAD_1 is absent from message_1.

Message_1 is constructed, as the CBOR Sequence of the CBOR data items above.

```
message_1 =  
(  
  1,  
  0,  
  h'b1a3e89460e88d3a8d54211dc95f0b903ff205eb71912d6db8f4af980d2db83a',  
  h'c3'  
)
```

message_1 (CBOR Sequence) (38 bytes)
01 00 58 20 b1 a3 e8 94 60 e8 8d 3a 8d 54 21 1d c9 5f 0b 90 3f f2 05 eb 71
91 2d 6d b8 f4 af 98 0d 2d b8 3a 41 c3

C.1.4. Message 2

Since $\text{TYPE} \bmod 4$ equals 1, C_U is omitted from data_2 .

G_Y (X-coordinate of the ephemeral public key of Party V) (32 bytes)

8d b5 77 f9 b9 c2 74 47 98 98 7d b5 57 bf 31 ca 48 ac d2 05 a9 db 8c 32 0e
5d 49 f3 02 a9 64 74

C_V (Connection identifier chosen by V) (1 bytes)

c4

Data_2 is constructed, as the CBOR Sequence of the CBOR data items above.

$\text{data_2} =$

```
(  
  h'8db577f9b9c2744798987db557bf31ca48acd205a9db8c320e5d49f302a96474',  
  h'c4'  
)
```

data_2 (CBOR Sequence) (36 bytes)

58 20 8d b5 77 f9 b9 c2 74 47 98 98 7d b5 57 bf 31 ca 48 ac d2 05 a9 db 8c
32 0e 5d 49 f3 02 a9 64 74 41 c4

From data_2 and message_1 (from Appendix C.1.3), compute the input to the transcript hash $\text{TH_2} = H(\text{message_1}, \text{data_2})$, as a CBOR Sequence of these 2 data items.

(message_1 , data_2) (CBOR Sequence)

(74 bytes)

01 00 58 20 b1 a3 e8 94 60 e8 8d 3a 8d 54 21 1d c9 5f 0b 90 3f f2 05 eb 71
91 2d 6d b8 f4 af 98 0d 2d b8 3a 41 c3 58 20 8d b5 77 f9 b9 c2 74 47 98 98
7d b5 57 bf 31 ca 48 ac d2 05 a9 db 8c 32 0e 5d 49 f3 02 a9 64 74 41 c4

And from there, compute the transcript hash $\text{TH_2} = \text{SHA-256}(\text{message_1}, \text{data_2})$

TH_2 value (32 bytes)

55 50 b3 dc 59 84 b0 20 9a e7 4e a2 6a 18 91 89 57 50 8e 30 33 2b 11 da 68
1d c2 af dd 87 03 55

When encoded as a CBOR bstr, that gives:

TH_2 (CBOR-encoded) (34 bytes)

58 20 55 50 b3 dc 59 84 b0 20 9a e7 4e a2 6a 18 91 89 57 50 8e 30 33 2b 11
da 68 1d c2 af dd 87 03 55

C.1.4.1. Signature Computation

COSE_Sign1 is computed with the following parameters. From Appendix C.1.2:

- o protected = bstr .cbor ID_CRED_V
- o payload = CRED_V

And from Appendix C.1.4:

- o external_aad = TH_2

The Sig_structure M_V to be signed is: ["Signature1", << ID_CRED_V >>, TH_2, CRED_V], as defined in Section 4.3.2:

```
M_V =
[
  "Signature1",
  << { 4: h'a3' } >>,
  h'5550b3dc5984b0209ae74ea26a18918957508e30332b11da681dc2afdd870355',
  << {
    1: 1,
    -1: 6,
    -2: h'1b661ee5d5ef1672a2d877cd5bc20f4630dc78a114de659c7e504d0f529a6b
        d3'
  } >>
]
```

Which encodes to the following byte string ToBeSigned:

M_V (message to be signed with Ed25519) (CBOR-encoded) (93 bytes)

```
84 6a 53 69 67 6e 61 74 75 72 65 31 44 a1 04 41 a3 58 20 55 50 b3 dc 59 84
b0 20 9a e7 4e a2 6a 18 91 89 57 50 8e 30 33 2b 11 da 68 1d c2 af dd 87 03
55 58 28 a3 01 01 20 06 21 58 20 1b 66 1e e5 d5 ef 16 72 a2 d8 77 cd 5b c2
0f 46 30 dc 78 a1 14 de 65 9c 7e 50 4d 0f 52 9a 6b d3
```

The message is signed using the private authentication key of V, and produces the following signature:

V's signature (64 bytes)

```
52 3d 99 6d fd 9e 2f 77 c7 68 71 8a 30 c3 48 77 8c 5e b8 64 dd 53 7e 55 5e
4a 00 05 e2 09 53 07 13 ca 14 62 0d e8 18 7e 81 99 6e e8 04 d1 53 b8 a1 f6
08 49 6f dc d9 3d 30 fc 1c 8b 45 be cc 06
```


C.1.4.2. Key and Nonce Computation

The key and nonce for calculating the ciphertext are calculated as follows, as specified in Section 3.3.

HKDF SHA-256 is the HKDF used (as defined by cipher suite 0).

$PRK = \text{HMAC-SHA-256}(\text{salt}, G_{XY})$

Since this is the asymmetric case, salt is the empty byte string.

G_{XY} is the shared secret, and since the curve25519 is used, the ECDH shared secret is the output of the X25519 function.

G_{XY} (32 bytes)

```
c6 1e 09 09 a1 9d 64 24 01 63 ec 26 2e 9c c4 f8 8c e7 7b e1 23 c5 ab 53 8d
26 b0 69 22 a5 20 67
```

From there, PRK is computed:

PRK (32 bytes)

```
ba 9c 2c a1 c5 62 14 a6 e0 f6 13 ed a8 91 86 8a 4c a3 e3 fa bc c7 79 8f dc
01 60 80 07 59 16 71
```

Key K_2 is the output of $\text{HKDF-Expand}(PRK, \text{info}, L)$.

info is defined as follows:

info for K_2

```
[
  10,
  [ null, null, null ],
  [ null, null, null ],
  [ 128, h'', h'5550b3dc5984b0209ae74ea26a18918957508e30332b11da681dc2afdd
    870355' ]
]
```

Which as a CBOR encoded data item is:

info (K_2) (CBOR-encoded) (48 bytes)

```
84 0a 83 f6 f6 f6 83 f6 f6 f6 83 18 80 40 58 20 55 50 b3 dc 59 84 b0 20 9a
e7 4e a2 6a 18 91 89 57 50 8e 30 33 2b 11 da 68 1d c2 af dd 87 03 55
```

L is the length of K_2 , so 16 bytes.

From these parameters, K_2 is computed:

K_2 (16 bytes)

da d7 44 af 07 c4 da 27 d1 f0 a3 8a 0c 4b 87 38

Nonce IV_2 is the output of HKDF-Expand(PRK, info, L).

info is defined as follows:

info for IV_2

```
[
  "IV-GENERATION",
  [ null, null, null ],
  [ null, null, null ],
  [ 104, h'', h'5550b3dc5984b0209ae74ea26a18918957508e30332b11da681dc2afdd
    870355' ]
]
```

Which as a CBOR encoded data item is:

info (IV_2) (CBOR-encoded) (61 bytes)

84 6d 49 56 2d 47 45 4e 45 52 41 54 49 4f 4e 83 f6 f6 f6 83 f6 f6 f6 83 18
68 40 58 20 55 50 b3 dc 59 84 b0 20 9a e7 4e a2 6a 18 91 89 57 50 8e 30 33
2b 11 da 68 1d c2 af dd 87 03 55

L is the length of IV_2, so 13 bytes.

From these parameters, IV_2 is computed:

IV_2 (13 bytes)

fb a1 65 d9 08 da a7 8e 4f 84 41 42 d0

C.1.4.3. Ciphertext Computation

COSE_Encrypt0 is computed with the following parameters. Note that UAD_2 is omitted.

- o empty protected header
- o external_aad = TH_2
- o plaintext = CBOR Sequence of the items kid_value, signature, in this order.

with kid_value taken from Appendix C.1.2, and signature as calculated in Appendix C.1.4.1.

The plaintext is the following:

P_2 (68 bytes)

```
41 a3 58 40 52 3d 99 6d fd 9e 2f 77 c7 68 71 8a 30 c3 48 77 8c 5e b8 64 dd
53 7e 55 5e 4a 00 05 e2 09 53 07 13 ca 14 62 0d e8 18 7e 81 99 6e e8 04 d1
53 b8 a1 f6 08 49 6f dc d9 3d 30 fc 1c 8b 45 be cc 06
```

From the parameters above, the Enc_structure A_2 is computed.

A_2 =

```
[
  "Encrypt0",
  h'',
  h'5550b3dc5984b0209ae74ea26a18918957508e30332b11da681dc2afdd870355'
]
```

Which encodes to the following byte string to be used as Additional Authenticated Data:

A_2 (CBOR-encoded) (45 bytes)

```
83 68 45 6e 63 72 79 70 74 30 40 58 20 55 50 b3 dc 59 84 b0 20 9a e7 4e a2
6a 18 91 89 57 50 8e 30 33 2b 11 da 68 1d c2 af dd 87 03 55
```

The key and nonce used are defined in Appendix C.1.4.2:

- o key = K_2

- o nonce = IV_2

Using the parameters above, the ciphertext CIPHERTEXT_2 can be computed:

CIPHERTEXT_2 (76 bytes)

```
1e 6b fe 0e 77 99 ce f0 66 a3 4f 08 ef aa 90 00 6d b4 4c 90 1c f7 9b 23 85
3a b9 7f d8 db c8 53 39 d5 ed 80 87 78 3c f7 a4 a7 e0 ea 38 c2 21 78 9f a3
71 be 64 e9 3c 43 a7 db 47 d1 e3 fb 14 78 8e 96 7f dd 78 d8 80 78 e4 9b 78
bf
```

C.1.4.4. message_2

From the parameter computed in Appendix C.1.4 and Appendix C.1.4.3, message_2 is computed, as the CBOR Sequence of the following items: (G_Y, C_V, CIPHERTEXT_2).

```
message_2 =
(
  h'8db577f9b9c2744798987db557bf31ca48acd205a9db8c320e5d49f302a96474',
  h'c4',
  h'1e6bfe0e7799cef066a34f08efaa90006db44c901cf79b23853ab97fd8dbc85339d5ed
8087783cf7a4a7e0ea38c221789fa371be64e93c43a7db47d1e3fb14788e967fdd78d880
78e49b78bf'
)
```

Which encodes to the following byte string:

```
message_2 (CBOR Sequence) (114 bytes)
58 20 8d b5 77 f9 b9 c2 74 47 98 98 7d b5 57 bf 31 ca 48 ac d2 05 a9 db 8c
32 0e 5d 49 f3 02 a9 64 74 41 c4 58 4c 1e 6b fe 0e 77 99 ce f0 66 a3 4f 08
ef aa 90 00 6d b4 4c 90 1c f7 9b 23 85 3a b9 7f d8 db c8 53 39 d5 ed 80 87
78 3c f7 a4 a7 e0 ea 38 c2 21 78 9f a3 71 be 64 e9 3c 43 a7 db 47 d1 e3 fb
14 78 8e 96 7f dd 78 d8 80 78 e4 9b 78 bf
```

C.1.5. Message 3

Since $\text{TYPE} \bmod 4$ equals 1, C_V is not omitted from data_3 .

```
C_V (1 bytes)
c4
```

data_3 is constructed, as the CBOR Sequence of the CBOR data item above.

```
data_3 =
(
  h'c4'
)
```

```
data_3 (CBOR Sequence) (2 bytes)
41 c4
```

From data_3 , CIPHERTEXT_2 (Appendix C.1.4.3), and TH_2 (Appendix C.1.4), compute the input to the transcript hash $\text{TH_2} = \text{H}(\text{TH_2}, \text{CIPHERTEXT_2}, \text{data_3})$, as a CBOR Sequence of these 3 data items.

```
( TH_2, CIPHERTEXT_2, data_3 )
(CBOR Sequence) (114 bytes)
58 20 55 50 b3 dc 59 84 b0 20 9a e7 4e a2 6a 18 91 89 57 50 8e 30 33 2b 11
da 68 1d c2 af dd 87 03 55 58 4c 1e 6b fe 0e 77 99 ce f0 66 a3 4f 08 ef aa
90 00 6d b4 4c 90 1c f7 9b 23 85 3a b9 7f d8 db c8 53 39 d5 ed 80 87 78 3c
f7 a4 a7 e0 ea 38 c2 21 78 9f a3 71 be 64 e9 3c 43 a7 db 47 d1 e3 fb 14 78
8e 96 7f dd 78 d8 80 78 e4 9b 78 bf 41 c4
```

And from there, compute the transcript hash $TH_3 = \text{SHA-256}(TH_2, \text{CIPHERTEXT_2}, \text{data_3})$

TH_3 value (32 bytes)

```
21 cc b6 78 b7 91 14 96 09 55 88 5b 90 a2 b8 2e 3b 2c a2 7e 8e 37 4a 79 07
f3 e7 85 43 67 fc 22
```

When encoded as a CBOR bstr, that gives:

TH_3 (CBOR-encoded) (34 bytes)

```
58 20 21 cc b6 78 b7 91 14 96 09 55 88 5b 90 a2 b8 2e 3b 2c a2 7e 8e 37 4a
79 07 f3 e7 85 43 67 fc 22
```

C.1.5.1. Signature Computation

COSE_Sign1 is computed with the following parameters. From Appendix C.1.2:

- o protected = bstr .cbor ID_CRED_U
- o payload = CRED_U

And from Appendix C.1.4:

- o external_aad = TH_3

The Sig_structure M_V to be signed is: ["Signature1", << ID_CRED_U >>, TH_3, CRED_U], as defined in Section 4.4.2:

M_U =

```
[
  "Signature1",
  << { 4: h'a2' } >>,
  h'734bef323d867a12956127c2e62ade42c0f119e5487750c0c31fd093376dceed',
  << {
    1: 1,
    -1: 6,
    -2: h'424c756ab77cc6fdecf0b3ecfcffb75310c015bf5cba2ec0a236e6650c8ab9
      c7'
  } >>
]
```

Which encodes to the following byte string ToBeSigned:

M_U (message to be signed with Ed25519) (CBOR-encoded) (93 bytes)

```
84 6a 53 69 67 6e 61 74 75 72 65 31 44 a1 04 41 a2 58 20 73 4b ef 32 3d 86
7a 12 95 61 27 c2 e6 2a de 42 c0 f1 19 e5 48 77 50 c0 c3 1f d0 93 37 6d ce
ed 58 28 a3 01 01 20 06 21 58 20 42 4c 75 6a b7 7c c6 fd ec f0 b3 ec fc ff
b7 53 10 c0 15 bf 5c ba 2e c0 a2 36 e6 65 0c 8a b9 c7
```

The message is signed using the private authentication key of U, and produces the following signature:

U's signature (64 bytes)

```
5c 7d 7d 64 c9 61 c5 f5 2d cf 33 91 25 92 a1 af f0 2c 33 62 b0 e7 55 0e 4b
c5 66 b7 0c 20 61 f3 c5 f6 49 e5 ed 32 3d 30 a2 6c 61 2f bb 5c bd 25 f3 1c
27 22 8c ea ec 64 29 31 95 41 fe 07 8e 0e
```

C.1.5.2. Key and Nonce Computation

The key and nonce for calculating the ciphertext are calculated as follows, as specified in Section 3.3.

HKDF SHA-256 is the HKDF used (as defined by cipher suite 0).

$PRK = \text{HMAC-SHA-256}(\text{salt}, G_{XY})$

Since this is the asymmetric case, salt is the empty byte string.

G_{XY} is the shared secret, and since the curve25519 is used, the ECDH shared secret is the output of the X25519 function.

G_{XY} (32 bytes)

```
c6 1e 09 09 a1 9d 64 24 01 63 ec 26 2e 9c c4 f8 8c e7 7b e1 23 c5 ab 53 8d
26 b0 69 22 a5 20 67
```

From there, PRK is computed:

PRK (32 bytes)

```
ba 9c 2c a1 c5 62 14 a6 e0 f6 13 ed a8 91 86 8a 4c a3 e3 fa bc c7 79 8f dc
01 60 80 07 59 16 71
```

Key K_3 is the output of $\text{HKDF-Expand}(PRK, \text{info}, L)$.

info is defined as follows:

info for K_3

```
[
  10,
  [ null, null, null ],
  [ null, null, null ],
  [ 128, h'', h'21ccb678b79114960955885b90a2b82e3b2ca27e8e374a7907f3e78543
    67fc22' ]
]
```

Which as a CBOR encoded data item is:

info (K_3) (CBOR-encoded) (48 bytes)

```
84 0a 83 f6 f6 f6 83 f6 f6 f6 83 18 80 40 58 20 21 cc b6 78 b7 91 14 96 09
55 88 5b 90 a2 b8 2e 3b 2c a2 7e 8e 37 4a 79 07 f3 e7 85 43 67 fc 22
```

L is the length of K_3, so 16 bytes.

From these parameters, K_3 is computed:

K_3 (16 bytes)

```
e1 ac d4 76 f5 96 a4 60 72 44 a8 da 8c ff 49 df
```

Nonce IV_3 is the output of HKDF-Expand(PRK, info, L).

info is defined as follows:

info for IV_3

```
[
  "IV-GENERATION",
  [ null, null, null ],
  [ null, null, null ],
  [ 104, h'', h'21ccb678b79114960955885b90a2b82e3b2ca27e8e374a7907f3e78543
    67fc22' ]
]
```

Which as a CBOR encoded data item is:

info (IV_3) (CBOR-encoded) (61 bytes)

```
84 6d 49 56 2d 47 45 4e 45 52 41 54 49 4f 4e 83 f6 f6 f6 83 f6 f6 f6 83 18
68 40 58 20 21 cc b6 78 b7 91 14 96 09 55 88 5b 90 a2 b8 2e 3b 2c a2 7e 8e
37 4a 79 07 f3 e7 85 43 67 fc 22
```

L is the length of IV_3, so 13 bytes.

From these parameters, IV_3 is computed:

IV_3 (13 bytes)

```
de 53 02 13 ab a2 6a 47 1a 51 f3 d6 fb
```

C.1.5.3. Ciphertext Computation

COSE_Encrypt0 is computed with the following parameters. Note that PAD_3 is omitted.

- o empty protected header
- o external_aad = TH_3
- o plaintext = CBOR Sequence of the items kid_value, signature, in this order.

with kid_value taken from Appendix C.1.1, and signature as calculated in Appendix C.1.5.1.

The plaintext is the following:

P_3 (68 bytes)

```
41 a2 58 40 5c 7d 7d 64 c9 61 c5 f5 2d cf 33 91 25 92 a1 af f0 2c 33 62 b0
e7 55 0e 4b c5 66 b7 0c 20 61 f3 c5 f6 49 e5 ed 32 3d 30 a2 6c 61 2f bb 5c
bd 25 f3 1c 27 22 8c ea ec 64 29 31 95 41 fe 07 8e 0e
```

From the parameters above, the Enc_structure A_3 is computed.

```
A_3 =
[
  "Encrypt0",
  h'',
  h'21ccb678b79114960955885b90a2b82e3b2ca27e8e374a7907f3e7854367fc22'
]
```

Which encodes to the following byte string to be used as Additional Authenticated Data:

A_2 (CBOR-encoded) (45 bytes)

```
83 68 45 6e 63 72 79 70 74 30 40 58 20 21 cc b6 78 b7 91 14 96 09 55 88 5b
90 a2 b8 2e 3b 2c a2 7e 8e 37 4a 79 07 f3 e7 85 43 67 fc 22
```

The key and nonce used are defined in Appendix C.1.4.2:

- o key = K_3
- o nonce = IV_3

Using the parameters above, the ciphertext CIPHERTEXT_3 can be computed:

CIPHERTEXT_3 (76 bytes)

```
de 4a 83 3d 48 b6 64 74 14 2c c9 bd ce 87 d9 3a f8 35 57 9c 2d bf 1b 9e 2f
b4 dc 66 60 0d ba c6 bb 3c c0 5c 29 0e f3 5d 51 5b 4d 7d 64 83 f5 09 61 43
b5 56 44 cf af d1 ff aa 7f 2b a3 86 36 57 83 1d d2 e5 bd 04 04 38 60 14 0d
c8
```

C.1.5.4. message_3

From the parameter computed in Appendix C.1.5 and Appendix C.1.5.3, message_3 is computed, as the CBOR Sequence of the following items: (C_V, CIPHERTEXT_3).

message_3 =

```
(
  h'c4',
  h'de4a833d48b66474142cc9bdce87d93af835579c2dbf1b9e2fb4dc66600dbac6bb3cc0
5c290ef35d515b4d7d6483f5096143b55644cfafd1ffaa7f2ba3863657831dd2e5bd0404
3860140dc8'
)
```

Which encodes to the following byte string:

message_3 (CBOR Sequence) (80 bytes)

```
41 c4 58 4c de 4a 83 3d 48 b6 64 74 14 2c c9 bd ce 87 d9 3a f8 35 57 9c 2d bf 1b
9e 2f b4 dc 66 60 0d ba c6 bb 3c c0 5c 29 0e f3 5d 51 5b 4d 7d 64 83 f5 09 61 4
3 b5 56 44 cf af d1 ff aa 7f 2b a3 86 36 57 83 1d d2 e5 bd 04 04 38 60 14 0d c8
```

C.1.5.5. OSCORE Security Context Derivation

From the previous message exchange, the Common Security Context for OSCORE [RFC8613] can be derived, as specified in Section 3.3.1.

First of all, TH_4 is computed: $TH_4 = H(TH_3, CIPHERTEXT_3)$, where the input to the hash function is the CBOR Sequence of TH_3 and CIPHERTEXT_3

(TH_3, CIPHERTEXT_3)

(CBOR Sequence) (112 bytes)

```
58 20 21 cc b6 78 b7 91 14 96 09 55 88 5b 90 a2 b8 2e 3b 2c a2 7e 8e 37 4a
79 07 f3 e7 85 43 67 fc 22 58 4c de 4a 83 3d 48 b6 64 74 14 2c c9 bd ce 87
d9 3a f8 35 57 9c 2d bf 1b 9e 2f b4 dc 66 60 0d ba c6 bb 3c c0 5c 29 0e f3
5d 51 5b 4d 7d 64 83 f5 09 61 43 b5 56 44 cf af d1 ff aa 7f 2b a3 86 36 57
83 1d d2 e5 bd 04 04 38 60 14 0d c8
```

And from there, compute the transcript hash $TH_4 = \text{SHA-256}(TH_3, CIPHERTEXT_3)$

TH_4 value (32 bytes)

```
51 ed 39 32 bc ba e8 90 1c 1d 4d eb 94 bd 67 3a b4 d3 8c 34 81 96 09 ee 0d
5c 9d a6 e9 80 7f e5
```

When encoded as a CBOR bstr, that gives:

TH_4 (CBOR-encoded) (34 bytes)

```
58 20 51 ed 39 32 bc ba e8 90 1c 1d 4d eb 94 bd 67 3a b4 d3 8c 34 81 96 09
ee 0d 5c 9d a6 e9 80 7f e5
```

To derive the Master Secret and Master Salt the same HKDF-Expand (PRK, info, L) is used, with different info and L.

For Master Secret:

L for Master Secret = 16

Info for Master Secret =

```
[
  "OSCORE Master Secret",
  [ null, null, null ],
  [ null, null, null ],
  [ 128, h'', h'51ed3932bcbae8901c1d4deb94bd673ab4d38c34819609ee0d5c9da6e9
    807fe5' ]
]
```

When encoded as a CBOR bstr, that gives:

info (OSCORE Master Secret) (CBOR-encoded) (68 bytes)

```
84 74 4f 53 43 4f 52 45 20 4d 61 73 74 65 72 20 53 65 63 72 65 74 83 f6 f6
f6 83 f6 f6 f6 83 18 80 40 58 20 51 ed 39 32 bc ba e8 90 1c 1d 4d eb 94 bd
67 3a b4 d3 8c 34 81 96 09 ee 0d 5c 9d a6 e9 80 7f e5
```

Finally, the Master Secret value computed is:

OSCORE Master Secret (16 bytes)

```
09 02 9d b0 0c 3e 01 27 42 c3 a8 69 04 07 4c 0e
```

For Master Salt:

L for Master Secret = 8

Info for Master Salt =

```
[
  "OSCORE Master Salt",
  [ null, null, null ],
  [ null, null, null ],
  [ 64, h'', h'51ed3932bcbae8901c1d4deb94bd673ab4d38c34819609ee0d5c9da6e98
    07fe5' ]
]
```

When encoded as a CBOR bstr, that gives:

info (OSCORE Master Salt) (CBOR-encoded) (66 bytes)

84 72 4f 53 43 4f 52 45 20 4d 61 73 74 65 72 20 53 61 6c 74 83 f6 f6 f6 83
f6 f6 f6 83 18 40 40 58 20 51 ed 39 32 bc ba e8 90 1c 1d 4d eb 94 bd 67 3a
b4 d3 8c 34 81 96 09 ee 0d 5c 9d a6 e9 80 7f e5

Finally, the Master Secret value computed is:

OSCORE Master Salt (8 bytes)

81 02 97 22 a2 30 4a 06

The Client's Sender ID takes the value of C_V:

Client's OSCORE Sender ID (1 bytes)

c4

The Server's Sender ID takes the value of C_U:

Server's OSCORE Sender ID (1 bytes)

c3

The algorithms are those negotiated in the cipher suite:

AEAD Algorithm

10

HMAC Algorithm

5

C.2. Test Vectors for EDHOC Authenticated with Symmetric Keys (PSK)

Symmetric EDHOC is used:

method (Symmetric Authentication)

1

CoAP is used as transport:

corr (Party U is CoAP client)

1

No unprotected opaque application data is sent in the message exchanges.

The pre-defined Cipher Suite 0 is in place both on Party U and Party V, see Section 3.1.

C.2.1. Input for Party U

The following are the parameters that are set in Party U before the first message exchange.

Party U's ephemeral private key (32 bytes)

f4 0c ea f8 6e 57 76 92 33 32 b8 d8 fd 3b ef 84 9c ad b1 9c 69 96 bc 27 2a
f1 f6 48 d9 56 6a 4c

Party U's ephemeral public key (value of X_U) (32 bytes)

ab 2f ca 32 89 83 22 c2 08 fb 2d ab 50 48 bd 43 c3 55 c6 43 0f 58 88 97 cb
57 49 61 cf a9 80 6f

Connection identifier chosen by U (value of C_U) (1 bytes)

c1

Pre-shared Key (PSK) (16 bytes)

a1 1f 8f 12 d0 87 6f 73 6d 2d 8f d2 6e 14 c2 de

kid value to identify PSK (1 bytes)

a1

So ID_PSK is defined as the following:

```
ID_PSK =  
{  
  4:  h'a1'  
}
```

This test vector uses COSE_Key objects to store the pre-shared key.

Note that since the map for ID_PSK contains a single 'kid' parameter, ID_PSK is used when transported in the protected header of the COSE Object, but only the kid_value is used when added to the plaintext (see Section 5.1):

ID_PSK (in protected header) (CBOR-encoded) (4 bytes)

a1 04 41 a1

kid_value (in plaintext) (CBOR-encoded) (2 bytes)

41 a1

C.2.2. Input for Party V

The following are the parameters that are set in Party U before the first message exchange.

Party V's ephemeral private key (32 bytes)

d9 81 80 87 de 72 44 ab c1 b5 fc f2 8e 55 e4 2c 7f f9 c6 78 c0 60 51 81 f3
7a c5 d7 41 4a 7b 95

Party V's ephemeral public key (value of X_V) (32 bytes)

fc 3b 33 93 67 a5 22 5d 53 a9 2d 38 03 23 af d0 35 d7 81 7b 6d 1b e4 7d 94
6f 6b 09 a9 cb dc 06

Connection identifier chosen by V (value of C_V) (1 bytes)

c2

Pre-shared Key (PSK) (16 bytes)

a1 1f 8f 12 d0 87 6f 73 6d 2d 8f d2 6e 14 c2 de

kid value to identify PSK (1 bytes)

a1

So ID_PSK is defined as the following:

```
ID_PSK =  
{  
  4:  h'a1'  
}
```

This test vector uses COSE_Key objects to store the pre-shared key.

Note that since the map for ID_PSK contains a single 'kid' parameter, ID_PSK is used when transported in the protected header of the COSE Object, but only the kid_value is used when added to the plaintext (see Section 5.1):

ID_PSK (in protected header) (CBOR-encoded) (4 bytes)

a1 04 41 a1

kid_value (in plaintext) (CBOR-encoded) (2 bytes)

41 a1

C.2.3. Message 1

From the input parameters (in Appendix C.2.1):

TYPE (4 * method + corr)

5

suite

0

SUITES_U : suite
0

G_X (X-coordinate of the ephemeral public key of Party U) (32 bytes)
ab 2f ca 32 89 83 22 c2 08 fb 2d ab 50 48 bd 43 c3 55 c6 43 0f 58 88 97 cb
57 49 61 cf a9 80 6f

C_U (Connection identifier chosen by U) (CBOR encoded) (2 bytes)
41 c1

kid_value of ID_PSK (CBOR encoded) (2 bytes)
41 a1

No UAD_1 is provided, so UAD_1 is absent from message_1.

Message_1 is constructed, as the CBOR Sequence of the CBOR data items above.

message_1 =
(
 5,
 0,
 h'ab2fca32898322c208fb2dab5048bd43c355c6430f588897cb574961cfa9806f',
 h'c1',
 h'a1'
)

message_1 (CBOR Sequence) (40 bytes)
05 00 58 20 ab 2f ca 32 89 83 22 c2 08 fb 2d ab 50 48 bd 43 c3 55 c6 43 0f
58 88 97 cb 57 49 61 cf a9 80 6f 41 c1 41 a1

C.2.4. Message 2

Since TYPE mod 4 equals 1, C_U is omitted from data_2.

G_Y (X-coordinate of the ephemeral public key of Party V) (32 bytes)
fc 3b 33 93 67 a5 22 5d 53 a9 2d 38 03 23 af d0 35 d7 81 7b 6d 1b e4 7d 94
6f 6b 09 a9 cb dc 06

C_V (Connection identifier chosen by V) (1 bytes)
c2

Data_2 is constructed, as the CBOR Sequence of the CBOR data items above.

```
data_2 =  
(  
  h'fc3b339367a5225d53a92d380323afd035d7817b6d1be47d946f6b09a9cbdc06',  
  h'c2'  
)
```

data_2 (CBOR Sequence) (36 bytes)

```
58 20 fc 3b 33 93 67 a5 22 5d 53 a9 2d 38 03 23 af d0 35 d7 81 7b 6d 1b e4  
7d 94 6f 6b 09 a9 cb dc 06 41 c2
```

From data_2 and message_1 (from Appendix C.2.3), compute the input to the transcript hash TH_2 = H(message_1, data_2), as a CBOR Sequence of these 2 data items.

(message_1, data_2) (CBOR Sequence)

(76 bytes)

```
05 00 58 20 ab 2f ca 32 89 83 22 c2 08 fb 2d ab 50 48 bd 43 c3 55 c6 43 0f  
58 88 97 cb 57 49 61 cf a9 80 6f 41 c1 41 a1 58 20 fc 3b 33 93 67 a5 22 5d  
53 a9 2d 38 03 23 af d0 35 d7 81 7b 6d 1b e4 7d 94 6f 6b 09 a9 cb dc 06 41  
c2
```

And from there, compute the transcript hash TH_2 = SHA-256(
message_1, data_2)

TH_2 value (32 bytes)

```
16 4f 44 d8 56 dd 15 22 2f a4 63 f2 02 d9 c6 0b e3 c6 9b 40 f7 35 8d 1c  
db 7b 07 de e1 70 ca
```

When encoded as a CBOR bstr, that gives:

TH_2 (CBOR-encoded) (34 bytes)

```
58 20 16 4f 44 d8 56 dd 15 22 2f a4 63 f2 02 d9 c6 0b e3 c6 9b 40 f7 35 8d  
34 1c db 7b 07 de e1 70 ca
```

C.2.4.1. Key and Nonce Computation

The key and nonce for calculating the ciphertext are calculated as follows, as specified in Section 3.3.

HKDF SHA-256 is the HKDF used (as defined by cipher suite 0).

PRK = HMAC-SHA-256(salt, G_XY)

Since this is the symmetric case, salt is the PSK:

salt (16 bytes)

```
a1 1f 8f 12 d0 87 6f 73 6d 2d 8f d2 6e 14 c2 de
```

G_{XY} is the shared secret, and since the curve25519 is used, the ECDH shared secret is the output of the X25519 function.

G_{XY} (32 bytes)
d5 75 05 50 6d 8f 30 a8 60 a0 63 d0 1b 5b 7a d7 6a 09 4f 70 61 3b 4a e6 6c
5a 90 e5 c2 1f 23 11

From there, PRK is computed:

PRK (32 bytes)
aa b2 f1 3c cb 1a 4f f7 96 a9 7a 32 a4 d2 fb 62 47 ef 0b 6b 06 da 04 d3 d1
06 39 4b 28 76 e2 8c

Key K₂ is the output of HKDF-Expand(PRK, info, L).

info is defined as follows:

info for K₂
[
 10,
 [null, null, null],
 [null, null, null],
 [128, h'', h'164f44d856dd15222fa463f202d9c60be3c69b40f7358d341cdb7b07de
 e170ca']
]

Which as a CBOR encoded data item is:

info (K₂) (CBOR-encoded) (48 bytes)
84 0a 83 f6 f6 f6 83 f6 f6 f6 83 18 80 40 58 20 16 4f 44 d8 56 dd 15 22 2f
a4 63 f2 02 d9 c6 0b e3 c6 9b 40 f7 35 8d 34 1c db 7b 07 de e1 70 ca

L is the length of K₂, so 16 bytes.

From these parameters, K₂ is computed:

K₂ (16 bytes)
ac 42 6e 5e 7d 7a d6 ae 3b 19 aa bd e0 f6 25 57

Nonce IV₂ is the output of HKDF-Expand(PRK, info, L).

info is defined as follows:


```
info for IV_2
[
  "IV-GENERATION",
  [ null, null, null ],
  [ null, null, null ],
  [ 104, h'', h'164f44d856dd15222fa463f202d9c60be3c69b40f7358d341cdb7b07de
    e170ca' ]
]
```

Which as a CBOR encoded data item is:

```
info (IV_2) (CBOR-encoded) (61 bytes)
84 6d 49 56 2d 47 45 4e 45 52 41 54 49 4f 4e 83 f6 f6 f6 83 f6 f6 f6 83 18
68 40 58 20 16 4f 44 d8 56 dd 15 22 2f a4 63 f2 02 d9 c6 0b e3 c6 9b 40 f7
35 8d 34 1c db 7b 07 de e1 70 ca
```

L is the length of IV_2, so 13 bytes.

From these parameters, IV_2 is computed:

```
IV_2 (13 bytes)
ff 11 2e 1c 26 8a a2 a7 7c c3 ee 6c 4d
```

C.2.4.2. Ciphertext Computation

COSE_Encrypt0 is computed with the following parameters. Note that UAD_2 is omitted.

- o empty protected header
- o external_aad = TH_2
- o empty plaintext, since UAD_2 is omitted

From the parameters above, the Enc_structure A_2 is computed.

```
A_2 =
[
  "Encrypt0",
  h'',
  h'164f44d856dd15222fa463f202d9c60be3c69b40f7358d341cdb7b07dee170ca'
]
```

Which encodes to the following byte string to be used as Additional Authenticated Data:

A_2 (CBOR-encoded) (45 bytes)

```
83 68 45 6e 63 72 79 70 74 30 40 58 20 16 4f 44 d8 56 dd 15 22 2f a4 63 f2
02 d9 c6 0b e3 c6 9b 40 f7 35 8d 34 1c db 7b 07 de e1 70 ca
```

The key and nonce used are defined in Appendix C.2.4.1:

- o key = K_2

- o nonce = IV_2

Using the parameters above, the ciphertext CIPHERTEXT_2 can be computed:

CIPHERTEXT_2 (8 bytes)

```
ba 38 b9 a3 fc 1a 58 e9
```

C.2.4.3. message_2

From the parameter computed in Appendix C.2.4 and Appendix C.2.4.2, message_2 is computed, as the CBOR Sequence of the following items: (G_Y, C_V, CIPHERTEXT_2).

message_2 =

```
(
  h'fc3b339367a5225d53a92d380323afd035d7817b6d1be47d946f6b09a9cbdc06',
  h'c2',
  h'ba38b9a3fc1a58e9'
)
```

Which encodes to the following byte string:

message_2 (CBOR Sequence) (45 bytes)

```
58 20 fc 3b 33 93 67 a5 22 5d 53 a9 2d 38 03 23 af d0 35 d7 81 7b 6d 1b e4
7d 94 6f 6b 09 a9 cb dc 06 41 c2 48 ba 38 b9 a3 fc 1a 58 e9
```

C.2.5. Message 3

Since TYPE mod 4 equals 1, C_V is not omitted from data_3.

C_V (1 bytes)

```
c2
```

Data_3 is constructed, as the CBOR Sequence of the CBOR data item above.

```
data_3 =  
(  
  h'c2'  
)
```

```
data_3 (CBOR Sequence) (2 bytes)  
41 c2
```

From data_3, CIPHERTEXT_2 (Appendix C.2.4.2), and TH_2 (Appendix C.2.4), compute the input to the transcript hash TH_2 = H(TH_2 , CIPHERTEXT_2, data_3), as a CBOR Sequence of these 3 data items.

```
( TH_2, CIPHERTEXT_2, data_3 ) (CBOR Sequence) (45 bytes)  
58 20 16 4f 44 d8 56 dd 15 22 2f a4 63 f2 02 d9 c6 0b e3 c6 9b 40 f7 35 8d  
34 1c db 7b 07 de e1 70 ca 48 ba 38 b9 a3 fc 1a 58 e9 41 c2
```

And from there, compute the transcript hash TH_3 = SHA-256(TH_2 , CIPHERTEXT_2, data_3)

```
TH_3 value (32 bytes)  
11 98 aa b3 ed db 61 b8 a1 b1 93 a9 e5 60 2b 5d 5f ea 76 bc 28 52 89 54 81  
b5 2b 8a f5 66 d7 fe
```

When encoded as a CBOR bstr, that gives:

```
TH_3 (CBOR-encoded) (34 bytes)  
58 20 11 98 aa b3 ed db 61 b8 a1 b1 93 a9 e5 60 2b 5d 5f ea 76 bc 28 52 89  
54 81 b5 2b 8a f5 66 d7 fe
```

C.2.5.1. Key and Nonce Computation

The key and nonce for calculating the ciphertext are calculated as follows, as specified in Section 3.3.

HKDF SHA-256 is the HKDF used (as defined by cipher suite 0).

PRK = HMAC-SHA-256(salt, G_XY)

Since this is the symmetric case, salt is the PSK:

```
salt (16 bytes)  
a1 1f 8f 12 d0 87 6f 73 6d 2d 8f d2 6e 14 c2 de
```

G_XY is the shared secret, and since the curve25519 is used, the ECDH shared secret is the output of the X25519 function.

G_XY (32 bytes)

```
d5 75 05 50 6d 8f 30 a8 60 a0 63 d0 1b 5b 7a d7 6a 09 4f 70 61 3b 4a e6 6c
5a 90 e5 c2 1f 23 11
```

From there, PRK is computed:

PRK (32 bytes)

```
aa b2 f1 3c cb 1a 4f f7 96 a9 7a 32 a4 d2 fb 62 47 ef 0b 6b 06 da 04 d3 d1
06 39 4b 28 76 e2 8c
```

Key K_3 is the output of HKDF-Expand(PRK, info, L).

info is defined as follows:

info for K_3

```
[
  10,
  [ null, null, null ],
  [ null, null, null ],
  [ 128, h'', h'1198aab3eddb61b8a1b193a9e5602b5d5fea76bc2852895481b52b8af5
    66d7fe' ]
]
```

Which as a CBOR encoded data item is:

info (K_3) (CBOR-encoded) (48 bytes)

```
84 0a 83 f6 f6 f6 83 f6 f6 f6 83 18 80 40 58 20 11 98 aa b3 ed db 61 b8 a1
b1 93 a9 e5 60 2b 5d 5f ea 76 bc 28 52 89 54 81 b5 2b 8a f5 66 d7 fe
```

L is the length of K_3, so 16 bytes.

From these parameters, K_3 is computed:

K_3 (16 bytes)

```
fe 75 e3 44 27 f8 3a ad 84 16 83 c6 6f a3 8a 62
```

Nonce IV_3 is the output of HKDF-Expand(PRK, info, L).

info is defined as follows:

info for IV_3

```
[
  "IV-GENERATION",
  [ null, null, null ],
  [ null, null, null ],
  [ 104, h'', h'1198aab3eddb61b8a1b193a9e5602b5d5fea76bc2852895481b52b8af5
    66d7fe' ]
]
```

Which as a CBOR encoded data item is:

```
info (IV_3) (CBOR-encoded) (61 bytes)
84 6d 49 56 2d 47 45 4e 45 52 41 54 49 4f 4e 83 f6 f6 f6 83 f6 f6 f6 83 18
68 40 58 20 11 98 aa b3 ed db 61 b8 a1 b1 93 a9 e5 60 2b 5d 5f ea 76 bc 28
52 89 54 81 b5 2b 8a f5 66 d7 fe
```

L is the length of IV_3, so 13 bytes.

From these parameters, IV_3 is computed:

```
IV_3 (13 bytes)
60 0a 33 b4 16 de 08 23 52 67 71 ec 8a
```

C.2.5.2. Ciphertext Computation

COSE_Encrypt0 is computed with the following parameters. Note that PAD_2 is omitted.

- o empty protected header
- o external_aad = TH_3
- o empty plaintext, since PAD_2 is omitted

From the parameters above, the Enc_structure A_3 is computed.

```
A_3 =
[
  "Encrypt0",
  h'',
  h'1198aab3eddb61b8a1b193a9e5602b5d5fea76bc2852895481b52b8af566d7fe'
]
```

Which encodes to the following byte string to be used as Additional Authenticated Data:

```
A_3 (CBOR-encoded) (45 bytes)
83 68 45 6e 63 72 79 70 74 30 40 58 20 11 98 aa b3 ed db 61 b8 a1 b1 93 a9
e5 60 2b 5d 5f ea 76 bc 28 52 89 54 81 b5 2b 8a f5 66 d7 fe
```

The key and nonce used are defined in Appendix C.2.5.1:

- o key = K_3
- o nonce = IV_3

Using the parameters above, the ciphertext CIPHERTEXT_3 can be computed:

CIPHERTEXT_3 (8 bytes)
51 29 07 92 61 45 40 04

C.2.5.3. message_3

From the parameter computed in Appendix C.2.5 and Appendix C.2.5.2, message_3 is computed, as the CBOR Sequence of the following items: (C_V, CIPHERTEXT_3).

message_3 =
(
 h'c2',
 h'5129079261454004'
)

Which encodes to the following byte string:

message_3 (CBOR Sequence) (11 bytes)
41 c2 48 51 29 07 92 61 45 40 04

C.2.5.4. OSCORE Security Context Derivation

From the previous message exchange, the Common Security Context for OSCORE [RFC8613] can be derived, as specified in Section 3.3.1.

First of all, TH_4 is computed: $TH_4 = H(TH_3, CIPHERTEXT_3)$, where the input to the hash function is the CBOR Sequence of TH_3 and CIPHERTEXT_3

(TH_3, CIPHERTEXT_3)
(CBOR Sequence) (43 bytes)
58 20 11 98 aa b3 ed db 61 b8 a1 b1 93 a9 e5 60 2b 5d 5f ea 76 bc 28 52 89
54 81 b5 2b 8a f5 66 d7 fe 48 51 29 07 92 61 45 40 04

And from there, compute the transcript hash $TH_4 = \text{SHA-256}(TH_3, CIPHERTEXT_3)$

TH_4 value (32 bytes)
df 7c 9b 06 f5 dc 0e e8 86 0b 39 6c 78 c5 be b7 57 41 3f a7 b6 a9 cf 28 3d
db 4c d4 c1 fd e4 3c

When encoded as a CBOR bstr, that gives:

TH_4 (CBOR-encoded) (34 bytes)

```
58 20 df 7c 9b 06 f5 dc 0e e8 86 0b 39 6c 78 c5 be b7 57 41 3f a7 b6 a9 cf
28 3d db 4c d4 c1 fd e4 3c
```

To derive the Master Secret and Master Salt the same HKDF-Expand (PRK, info, L) is used, with different info and L.

For Master Secret:

L for Master Secret = 16

Info for Master Secret =

```
[
  "OSCORE Master Secret",
  [ null, null, null ],
  [ null, null, null ],
  [ 128, h'', h'df7c9b06f5dc0ee8860b396c78c5beb757413fa7b6a9cf283ddb4cd4c1
    fde43c' ]
]
```

When encoded as a CBOR bstr, that gives:

info (OSCORE Master Secret) (CBOR-encoded) (68 bytes)

```
84 74 4f 53 43 4f 52 45 20 4d 61 73 74 65 72 20 53 65 63 72 65 74 83 f6 f6
f6 83 f6 f6 f6 83 18 80 40 58 20 df 7c 9b 06 f5 dc 0e e8 86 0b 39 6c 78 c5
be b7 57 41 3f a7 b6 a9 cf 28 3d db 4c d4 c1 fd e4 3c
```

Finally, the Master Secret value computed is:

OSCORE Master Secret (16 bytes)

```
8d 36 8f 09 26 2d c5 52 7f e7 19 e6 6c 91 63 75
```

For Master Salt:

L for Master Secret = 8

Info for Master Salt =

```
[
  "OSCORE Master Salt",
  [ null, null, null ],
  [ null, null, null ],
  [ 64, h'', h'df7c9b06f5dc0ee8860b396c78c5beb757413fa7b6a9cf283ddb4cd4c1f
    de43c' ]
]
```

When encoded as a CBOR bstr, that gives:

info (OSCORE Master Salt) (CBOR-encoded) (66 bytes)

```
84 72 4f 53 43 4f 52 45 20 4d 61 73 74 65 72 20 53 61 6c 74 83 f6 f6 f6 83
f6 f6 f6 83 18 40 40 58 20 df 7c 9b 06 f5 dc 0e e8 86 0b 39 6c 78 c5 be b7
57 41 3f a7 b6 a9 cf 28 3d db 4c d4 c1 fd e4 3c
```

Finally, the Master Secret value computed is:

OSCORE Master Salt (8 bytes)
4d b7 06 58 c5 e9 9f b6

The Client's Sender ID takes the value of C_V:

Client's OSCORE Sender ID (1 bytes)
c2

The Server's Sender ID takes the value of C_U:

Server's OSCORE Sender ID (1 bytes)
c1

The algorithms are those negotiated in the cipher suite:

AEAD Algorithm
10

HMAC Algorithm
5

Acknowledgments

The authors want to thank Alessandro Bruni, Martin Disch, Theis Groenbech Petersen, Dan Harkins, Klaus Hartke, Russ Housley, Alexandros Krontiris, Ilari Liusvaara, Karl Norrman, Salvador Perez, Eric Rescorla, Michael Richardson, Thorvald Sahl Joergensen, Jim Schaad, Carsten Schuermann, Ludwig Seitz, Stanislav Smyshlyaev, Valery Smyslov, Rene Struik, and Erik Thormarker for reviewing and commenting on intermediate versions of the draft. We are especially indebted to Jim Schaad for his continuous reviewing and implementation of different versions of the draft.

Authors' Addresses

Goeran Selander
Ericsson AB

Email: goran.selander@ericsson.com

John Mattsson
Ericsson AB

Email: john.mattsson@ericsson.com

Francesca Palombini
Ericsson AB

Email: francesca.palombini@ericsson.com

ACE Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 22, 2016

G. Selander
J. Mattsson
F. Palombini
Ericsson AB
L. Seitz
SICS Swedish ICT
March 21, 2016

Object Security of CoAP (OSCOAP)
draft-selander-ace-object-security-04

Abstract

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

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 22, 2016.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of

publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Terminology	4
2. The Object-Security Option	5
3. The Security Context	6
4. Protected CoAP Message Fields	8
5. The COSE Object	10
5.1. Plaintext	11
5.2. Additional Authenticated Data	12
6. Protecting CoAP Messages	13
6.1. Replay and Freshness Protection	13
6.2. Protecting the Request	13
6.3. Verifying the Request	14
6.4. Protecting the Response	15
6.5. Verifying the Response	16
7. Security Considerations	16
8. Privacy Considerations	18
9. IANA Considerations	18
9.1. CoAP Option Number Registration	18
9.2. Media Type Registrations	19
9.3. CoAP Content Format Registration	20
10. Acknowledgments	21
11. References	21
11.1. Normative References	21
11.2. Informative References	21
Appendix A. Overhead	22
A.1. Length of the Object-Security Option	22
A.2. Size of the COSE Object	23
A.3. Message Expansion	24
A.4. Example	24
Appendix B. Examples	25
B.1. Secure Access to Actuator	25
B.2. Secure Subscribe to Sensor	27
Appendix C. Object Security of Content (OSCON)	28
C.1. Overhead OSCON	30
C.2. MAC Only	30
C.3. Signature Only	31
C.4. Authenticated Encryption with Additional Data (AEAD)	32
C.5. Symmetric Encryption with Asymmetric Signature (SEAS)	33
Authors' Addresses	33

1. Introduction

The Constrained Application Protocol (CoAP) [RFC7252] is a web application protocol, designed for constrained nodes and networks [RFC7228]. CoAP specifies the use of proxies, to improve scalability, efficiency, and uses. At the same time CoAP references DTLS [RFC6347] for security. Proxy operations on CoAP messages require DTLS to be terminated at the proxy. The proxy therefore not only has access to the data required for performing the intended proxy functionality, but is also able to eavesdrop on, or manipulate any part of the CoAP payload and metadata, in transit between client and server. The proxy can also inject, delete, or reorder packages without being protected or detected by DTLS.

This memo defines Object Security of CoAP (OSCOAP), a data object based security protocol, protecting CoAP message exchanges end-to-end, across intermediary nodes. An analysis of end-to-end security for CoAP messages through intermediary nodes is performed in [I-D.hartke-core-e2e-security-reqs]; OSCOAP targets the requirements in Sections 3.1 and 3.2.

OSCOAP builds on the CBOR Encoded Message Syntax (COSE) [I-D.ietf-cose-msg], providing end-to-end encryption, integrity, and replay protection. The use of OSCOAP is signaled with the CoAP option Object-Security, also defined in this memo.

OSCOAP transforms an unprotected CoAP message into a protected CoAP message in the following way: the unprotected CoAP message is protected by including payload (if present), certain options, and header fields in a COSE object. The message fields that have been encrypted are removed from the message whereas the Object-Security option and the COSE object are added. We call the result the "protected" CoAP message. Thus OSCOAP is a security protocol based on the exchange of protected CoAP messages (see Figure 1).

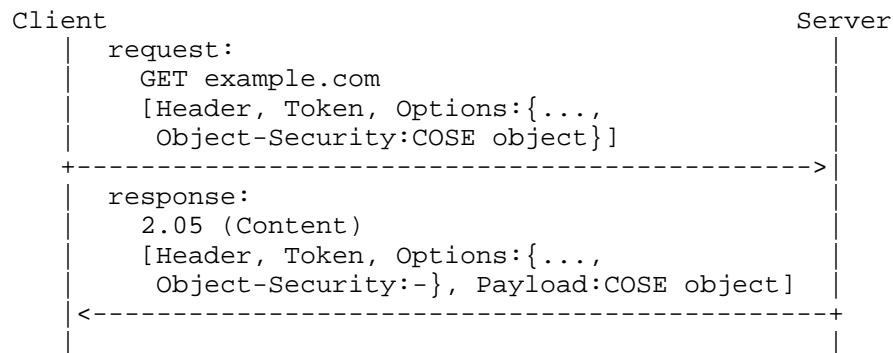


Figure 1: Sketch of OSCOAP

OSCOAP provides protection of CoAP payload, certain options, and header fields, as well as a secure binding between CoAP request and response messages, and freshness of requests and responses.

OSCOAP may be used in constrained settings, where DTLS cannot be supported. Alternatively, OSCOAP can be combined with DTLS, thereby enabling 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. Examples of the use of OSCOAP are given in Appendix B.

The message protection provided by OSCOAP can alternatively be applied to payload only of individual messages. We call this object security of content (OSCON) and it is defined in Appendix C. OSCON targets the requirements in Sections 3.3 - 3.5 of [I-D.hartke-core-e2e-security-reqs].

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.

Readers are expected to be familiar with the terms and concepts described in [RFC7252] and [RFC7641].

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

Two different scopes of object security are defined:

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

OSCON is defined in Appendix C.

2. The Object-Security Option

The Object-Security option indicates that OSCOAP is used to protect the CoAP message exchange.

The Object-Security option is critical, safe to forward, part of the cache key, and not repeatable. Figure 2 illustrates the structure of the Object-Security option.

A CoAP proxy SHOULD NOT cache a response to a request with an Object-Security option, since the response is only applicable to the original client's request. The Object-Security option is included in the cache key for backward compatibility with proxies not recognizing the Object-Security option. The effect of this is that messages with the Object-Security option will never generate cache hits. To further prevent caching, a Max-Age option with value zero can be added to the protected CoAP responses.

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

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

Figure 2: The Object-Security Option

The length of the Object-Security option depends on whether the unprotected message has payload, on the set of options that are included in the unprotected message, the length of the integrity tag, and the length of the information identifying the security context.

- o If the unprotected message has payload, then the COSE object is the payload of the protected message (see Section 6.2 and Section 6.4), and the Object-Security option has length zero.
- o If the unprotected message does not have payload, then the COSE object is the value of the Object-Security option and the length of the Object-Security option is equal to the size of the COSE object.

An example of option length is given in Appendix A.

3. The Security Context

The security context is the set of information elements necessary to carry out the cryptographic operations in OSCOAP. A security context needs to be pre-established and agreed upon between client and server. How this is done is out of scope of this memo, an example is given in the appendices of [I-D.selander-ace-cose-ecdh]. Each security context is identified by a Context Identifier, which is unique within a given server. A Context Identifier that is no longer in use can be reassigned to a new security context.

The security context has a "Client Write" part and a "Server Write" part. The client initiating a transaction uses the Client Write part of the context to protect the request; the server receiving the request first uses the Client Write part of the context to verify the request, then the Server Write part of the context to protect the response. Finally, the client uses the Server Write part of the context to verify the response.

OSCOAP is very similar to TLS and borrows mechanisms such as key derivation, and nonce construction from [I-D.ietf-tls-tls13]. The main differences is that OSCOAP uses COSE [I-D.ietf-cose-msg] instead of the TLS record layer, which allows OSCOAP to use a context identifier, and sequence numbers of variable length.

It should be noted that how the context is retrieved within the client and server is linked to the resource discovery, may be implementation specific, and is out of scope of this memo.

An example is shown in Figure 3.

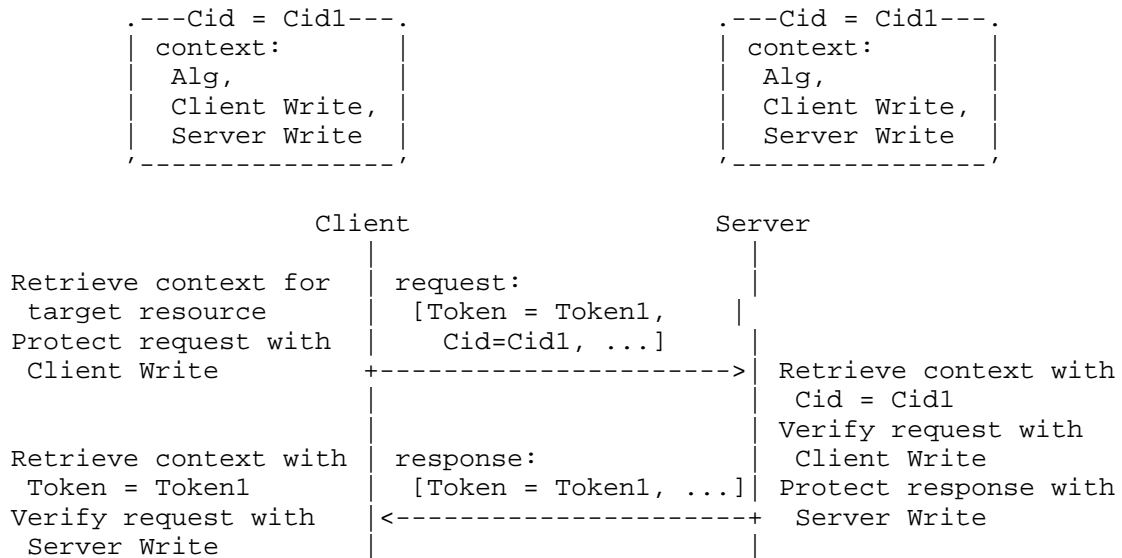


Figure 3: Retrieval and use of the Security Context

The security context structure contains the following parameters:

- o Context Identifier (Cid). Variable length byte string that identifies the security context. Immutable.
- o Algorithm (Alg). Value that identifies the COSE AEAD algorithm to use for encryption. Immutable.
- o Client Write Key. Byte string containing the symmetric key to use in client-sent messages. Length is determined by Algorithm. Immutable.
- o Client Write IV. Byte string containing the static IV to use in cryptographic operations on client-sent messages. Length is determined by Algorithm. Immutable.
- o Client Write Sequence Number. Non-negative integer enumerating the COSE objects that the client sent, associated to the Context Identifier. It is used for replay protection, and to generate unique nonces. Initiated to 0. Maximum value is determined by Algorithm.
- o Server Write Key. Byte string containing the symmetric key to use in server-sent messages. Length is determined by the Algorithm. Immutable.

- o Server Write IV. Byte string containing the static IV to use in cryptographic operations on server-sent messages. Length is determined by Algorithm. Immutable.
- o Server Write Sequence Number. Non-negative integer enumerating the COSE objects that the server sent, associated to the Context Identifier. It is used for replay protection, and to generate unique nonces. Initiated to 0. Maximum value is determined by Algorithm.
- o Replay Window. The replay protection window for messages received, equivalent to the functionality described in Section 4.1.2.6 of [RFC6347]. The default window size is 64.

The size of Cid depends on the number of simultaneous clients, and must be chosen so that the server can uniquely identify the requesting client. Cids of different lengths can be used by different client. In the case of an ACE-based authentication and authorization model [I-D.ietf-ace-oauth-authz], the Authorization Server can define the context identifier of all clients, interacting with a particular server, in which case the size of Cid can be proportional to the logarithm of the number of authorized clients. It is RECOMMENDED to start assigning Cids of length 1 byte (0x00, 0x01, ..., 0xff), and then when all 1 byte Cids are in use, start handling out Cids with a length of two bytes (0x0000, 0x0001, ..., 0xffff), and so on.

The ordered pair (Cid, Client Write Sequence Number) is called Transaction Identifier (Tid), and SHALL be unique for each COSE object and server. The Tid is used as a unique challenge in the COSE object of the protected CoAP request, and in part of the Additional Authenticated Data (AAD, see Section 5) of the protected CoAP response message.

4. Protected CoAP Message Fields

This section defines how the CoAP message fields are protected. OSCOAP protects as much of the unprotected CoAP message as possible, while still allowing forward proxy operations [I-D.hartke-core-e2e-security-reqs].

The CoAP Payload SHALL be encrypted and integrity protected.

The CoAP Header fields Version and Code SHALL be integrity protected but not encrypted. The CoAP Message Layer parameters, Type and Message ID, as well as Token and Token Length SHALL neither be integrity protected nor encrypted.

Protection of CoAP Options can be summarized as follows:

- o To prevent information leakage, Uri-Path and Uri-Query SHALL be encrypted. As a consequence, if Proxy-Uri is used, those parts of the URI SHALL be removed from the Proxy-Uri. The CoAP Options Uri-Host, Uri-Port, Proxy-Uri, and Proxy-Scheme SHALL neither be encrypted, nor integrity protected (cf. protection of request URI in Section 5.2).
- o The other CoAP options listed in Figure 4 SHALL be encrypted and integrity protected.

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

C=Critical, U=Unsafe, N=NoCacheKey, R=Repeatable,
E=Encrypt, I=Integrity Protect, D=Duplicate.

Figure 4: Protected CoAP Options

Unless specified otherwise, CoAP options not listed in Figure 4 SHALL be encrypted and integrity protected.

Specifications of new CoAP options SHOULD specify how they are processed with OSCOAP. New COAP options SHOULD be encrypted and integrity protected. New COAP options SHALL be integrity protected unless a proxy needs to change the option, and SHALL be encrypted unless a proxy needs to read the option.

The encrypted options are in general omitted from the protected CoAP message and not visible to intermediary nodes (see Section 6.2 and

Section 6.4). Hence the actions resulting from the use of corresponding options is analogous to the case of communicating directly with the endpoint. For example, a client using an ETag option will not be served by a proxy.

However, some options which are encrypted need to be present in the protected CoAP message to support certain proxy functions. A CoAP option which may be both encrypted in the COSE object of the protected CoAP message, and also unencrypted as CoAP option in the protected CoAP message, is called "duplicate". The "encrypted" value of a duplicate option is intended for the destination endpoint and the "unencrypted" value is intended for a proxy. The unencrypted value is not integrity protected.

- o The Max-Age option is duplicate. The unencrypted Max-Age SHOULD have value zero to prevent caching of responses. The encrypted Max-Age is used as defined in [RFC7252] taking into account that it is not accessible proxies.
- o The Observe option is duplicate. If used, then the encrypted Observe and the unencrypted Observe SHALL have the same value. The Observe option as used here targets the requirements of Section 3.2 of [I-D.hartke-core-e2e-security-reqs].

Specifications of new CoAP options SHOULD specify if the option is duplicate and how it are processed with OSCOAP. New COAP options SHOULD NOT be duplicate.

5. The COSE Object

This section defines how to use the COSE format [I-D.ietf-cose-msg] to wrap and protect data in the unprotected CoAP message. OSCOAP uses the COSE_Encrypted structure with an Authenticated Encryption with Additional Data (AEAD) algorithm.

The mandatory to support AEAD algorithm is AES-CCM-64-64-128 defined in Section 10.2 of [I-D.ietf-cose-msg]. For AES-CCM-64-64-128 the length of Client Write Key and the Server Write Key SHALL be 128 bits, the length of the nonce, Client Write IV, and the Server Write IV SHALL be 7 bytes, and the maximum Client Write Sequence Number and Server Write Sequence Number SHALL be $2^{56}-1$. The nonce is constructed exactly like in Section 5.2.2 of [I-D.ietf-tls-tls13], i.e. by padding the Client Write Sequence Number or the Server Write Sequence Number with zeroes and XORing it with the static Client Write IV or Server Write IV, respectively.

Since OSCOAP only makes use of a single COSE structure, there is no need to explicitly specify the structure, and OSCOAP uses the

untagged version of the COSE_Encrypted structure (Section 2. of [I-D.ietf-cose-msg]). If the COSE object has a different structure, the receiver MUST reject the message, treating it as malformed.

We denote by Plaintext the data that is encrypted and integrity protected, and by Additional Authenticated Data (AAD) the data that is integrity protected only, in the COSE object.

The fields of COSE_Encrypted structure are defined as follows (see example in Appendix C.4).

- o The "Headers" field is formed by:
 - * The "protected" field, which SHALL include:
 - + The "Partial Initialization Vector" parameter. The value is set to the Client Write Sequence Number, or the Server Write Sequence Number, depending on whether the client or server is sending the message. The Partial IV is a byte string (type: bstr), where the length is the minimum length needed to encode the sequence number.
 - + If the message is a CoAP request, the "kid" parameter. The value is set to the Context Identifier (see Section 3).
 - * The "unprotected" field, which SHALL be empty.
- o The "ciphertext" field is computed from the Plaintext and the Additional Authenticated Data (AAD) and encoded as a byte string (type: bstr), following Section 5.2 of [I-D.ietf-cose-msg].

5.1. Plaintext

The Plaintext is formatted as a CoAP message without Header (see Figure 5) consisting of:

- o all CoAP Options present in the unprotected message which are encrypted (see Section 4), in the order as given by the Option number (each Option with Option Header including delta to previous included encrypted option); and
- o the CoAP Payload, if present, and in that case prefixed by the one-byte Payload Marker (0xFF).

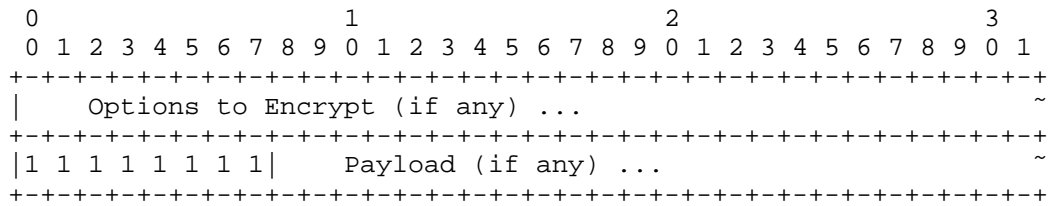


Figure 5: Plaintext

5.2. Additional Authenticated Data

The Additional Authenticated Data ("Enc_structure") as described in Section 5.3 of [I-D.ietf-cose-msg] includes (see Figure 6):

- o the "context" parameter, which has value "Encrypted"
- o the "protected" parameter, which includes the "protected" part of the "Headers" field;
- o the "external_aad" includes:
 - * the two first bytes of the CoAP header in the unprotected message (including Version and Code) with Type and Token Length bits set to 0;
 - * The Algorithm from the security context used for the exchange;
 - * the plaintext request URI composed from the request scheme and Uri-* options according to the method described in Section 6.5 of [RFC7252], if the message is a CoAP request; and
 - * the Transaction Identifier (Tid) of the associated CoAP request, if the message is a CoAP response (see Section 3).

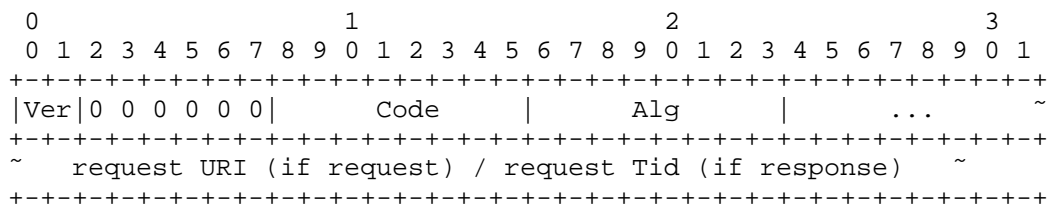


Figure 6: Additional Authenticated Data

The encryption process is described in Section 5.3 of [I-D.ietf-cose-msg].

6. Protecting CoAP Messages

6.1. Replay and Freshness Protection

In order to protect from replay of messages and verify freshness, a CoAP endpoint SHALL maintain a Client Write Sequence Number, and a Server Write Sequence Number associated to a security context, which is identified with a Context Identifier (Cid). The two sequence numbers are the highest sequence number the endpoint has sent and the highest sequence number the endpoint has received. A client uses the Client Write Sequence Number for protecting sent messages and the Server Write Sequence Number for verifying received messages, and vice versa for the server, as described in Section 3.

Depending on use case and ordering of messages provided by underlying layers, an endpoint MAY maintain a sliding replay window for Sequence Numbers of received messages associated to each Cid.

A receiving endpoint SHALL verify that the Sequence Number received in the COSE object has not been received before in the security context identified by the Cid. Note that for the server, the relevant Sequence Number here is the Client Write Sequence Number and vice versa for the client.

OSCOAP is a challenge-response protocol, where the response is verified to match a prior request, by including the unique transaction identifier (Tid as defined in Section 3) of the request in the Additional Authenticated Data of the response message.

If a CoAP server receives a request with the Object-Security option, then the server SHALL include the Tid of the request in the AAD of the response, as described in Section 6.4.

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 AAD, as described in Section 6.5.

6.2. Protecting the Request

Given an unprotected CoAP request, including header, options and payload, the client SHALL perform the following steps to create a protected CoAP request using a security context associated with the target resource:

1. Increment the Client Write Sequence Number by one (note that this means that sequence number 0 is never used). If the Client Write Sequence Number exceeds the maximum number for the AEAD

algorithm, the client MUST NOT process any requests with the given security context. The client SHOULD acquire a new security context before this happens. The latter is out of scope of this memo.

2. Compute the COSE object as specified in Section 5
 - * the nonce in the AEAD is created by XORing the static IV (Client Write IV) with the partial IV (Client Write Sequence Number).
3. Format the protected CoAP message as an ordinary CoAP message, with the following Header, Options, and Payload, based on the unprotected CoAP message:
 - * The CoAP header is the same as the unprotected CoAP message.
 - * The CoAP options which are encrypted and not duplicate (Section 4) are removed. Any duplicate option which is present has its unencrypted value. The Object-Security option is added.
 - * If the unprotected CoAP message has no Payload, then the value of the Object-Security option is the COSE object. If the unprotected CoAP message has Payload, then the Object-Security option is empty and the Payload of the protected CoAP message is the COSE object.

The Client SHALL be able to find the correct security context with use of the Token of the message exchange.

6.3. Verifying the Request

A CoAP server receiving a message containing the Object-Security option SHALL perform the following steps, using the security context identified by the Context Identifier in the "kid" parameter in the received COSE object:

1. Verify the Sequence Number in the Partial IV parameter, as described in Section 6.1. If it cannot be verified that the Sequence Number has not been received before, the server MUST stop processing the request.
2. Recreate the Additional Authenticated Data, as described in Section 5.
3. Compose the nonce by XORing the static IV (Client Write IV) with the Partial IV parameter, received in the COSE Object.

4. Retrieve the Client Write Key.
5. Verify and decrypt the message. If the verification fails, the server MUST stop processing the request.
6. If the message verifies, update the Client Write Sequence Number or Replay Window, as described in Section 6.1.
7. Restore the unprotected request by adding any decrypted options or payload from the plaintext. Any duplicate options (Section 4) are overwritten. The Object-Security option is removed.

6.4. Protecting the Response

A server receiving a valid request with a protected CoAP message (i.e. containing an Object-Security option) SHALL respond with a protected CoAP message.

Given an unprotected CoAP response, including header, options, and payload, the server SHALL perform the following steps to create a protected CoAP response, using the security context identified by the Context Identifier of the received request:

1. Increment the Server Write Sequence Number by one (note that this means that sequence number 0 is never used). If the Server Write Sequence Number exceeds the maximum number for the AEAD algorithm, the server MUST NOT process any more responses with the given security context. The server SHOULD acquire a new security context before this happens. The latter is out of scope of this memo.
2. Compute the COSE object as specified in Section 5
 - * The nonce in the AEAD is created by XORing the static IV (Server Write IV) and the Server Write Sequence Number.
3. Format the protected CoAP message as an ordinary CoAP message, with the following Header, Options, and Payload based on the unprotected CoAP message:
 - * The CoAP header is the same as the unprotected CoAP message.
 - * The CoAP options which are encrypted and not duplicate (Section 4) are removed. Any duplicate option which is present has its unencrypted value. The Object-Security option is added.

- * If the unprotected CoAP message has no Payload, then the value of the Object-Security option is the COSE object. If the unprotected CoAP message has Payload, then the Object-Security option is empty, and the Payload of the protected CoAP message is the COSE object.

Note the differences between generating a protected request, and a protected response, for example whether "kid" is present in the header, or whether Destination URI or Tid is present in the AAD, of the COSE object.

6.5. Verifying the Response

A CoAP client receiving a message containing the Object-Security option SHALL perform the following steps, using the security context identified by the Token of the received response:

1. Verify the Sequence Number in the Partial IV parameter as described in Section 6.1. If it cannot be verified that the Sequence Number has not been received before, the client MUST stop processing the response.
2. Recreate the Additional Authenticated Data as described in Section 5.
3. Compose the nonce by XORing the static IV (Server Write IV) with the Partial IV parameter, received in the COSE Object.
4. Retrieve the Server Write Key.
5. Verify and decrypt the message. If the verification fails, the client MUST stop processing the response.
6. If the message verifies, update the Client Write Sequence Number or Replay Window, as described in Section 6.1.
7. Restore the unprotected response by adding any decrypted options or payload from the plaintext. Any duplicate options (Section 4) are overwritten. The Object-Security option is removed.

7. Security Considerations

In scenarios with intermediary nodes such as proxies or brokers, transport layer security such as DTLS only protects data hop-by-hop. As a consequence the intermediary nodes can read and modify information. The trust model where all intermediate 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 brittle.

DTLS protects hop-by-hop the entire CoAP message, including header, options, and payload. OSCOAP protects end-to-end the payload, and all information in the options and header, that is not required for forwarding (see Section 4). DTLS and OSCOAP can be combined.

The CoAP message layer, however, cannot be protected end-to-end through intermediary devices 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 application layer security solution, since message layer is excluded from that.

The specification in this memo assumes that there is an established security context. [I-D.ietf-ace-oauth-authz] presents a method for a trusted third party (Authorization Server) to enable key establishment between potentially constrained nodes, using OAuth and PoP Tokens. [I-D.selander-ace-cose-ecdhe] describes a Diffie-Hellman key exchange, authenticated with pre-established keys, and a key derivation method for producing a security context, suitable for OSCOAP. The two methods can be combined, enabling a client and server with relation to a trusted third party to establish a security context with forward secrecy.

For symmetric encryption it is required to have a unique nonce for each message, for which the sequence numbers in the COSE message field "Partial IV" is used. The nonce SHALL be the XOR of a static IV and the sequence number. The static IVs (Client Write IV and Server Write IV) SHOULD be established between sender and receiver before the message is sent, to avoid the overhead of sending it in each message, for example using the method in [I-D.selander-ace-cose-ecdhe].

As the receiver accepts any sequence number larger than the one previously received, the problem of sequence number synchronization is avoided. The alternatives have issues: very constrained devices may not be able to support accurate time, or to 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.

Block-wise transfers as currently defined in [I-D.ietf-core-block] cannot be protected end-to-end because the payload as well as the Block1/Block2 options may be changed in an unpredictable way by a proxy. Since [I-D.ietf-core-block] allows for any proxy to fragment the payload, an endpoint receiving a message fragment with a block option is not able to verify integrity of that fragment. As a consequence, block-wise disables end-to-end security: an adversary may inject an unlimited number of messages with a block option claiming it to be a sequence of message fragments without the receiving endpoint being able to disprove the claim.

If instead the payload and block options Block1/Block2 were not allowed to be changed by intermediate devices, then the message fragments could be integrity protected end-to-end. In that case each individual block can be securely verified by the receiver, retransmission securely requested etc. Since the blocks are enumerated sequentially, and carry information about whether this fragment is the last, when all blocks have been securely received is enough to prove that the entire message has been securely transferred.

8. Privacy Considerations

Privacy threats executed through intermediate nodes are considerably reduced by means of OSCOAP. End-to-end integrity protection and encryption of CoAP payload and all options that are not used for forwarding, provides mitigation against attacks on sensor and actuator communication, which may have a direct impact the personal sphere.

CoAP headers sent in plaintext allow for example matching of CON and ACK (CoAP Message Identifier), matching of request and responses (Token) and traffic analysis.

9. IANA Considerations

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

9.1. CoAP Option Number Registration

The Object-Security option is added to the CoAP Option Numbers registry:

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

9.2. Media Type Registrations

The "application/oscon" media type is added to the Media Types registry:

Type name: application

Subtype name: cose

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: See the Security Considerations section of [[this document]].

Interoperability considerations: N/A

Published specification: [[this document]]

Applications that use this media type: To be identified

Fragment identifier considerations: N/A

Additional information:

- * Magic number(s): N/A

- * File extension(s): N/A

- * 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: Goeran Selander, goran.selander@ericsson.com

Change Controller: IESG

Provisional registration? No

9.3. CoAP Content Format Registration

The "application/oscon" content format is added to the CoAP Content Format registry:

Media type	Encoding	ID	Reference
application/oscon	-	70	[[this document]]

10. 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 previous versions of the draft.

11. References

11.1. Normative References

- [I-D.ietf-cose-msg]
Schaad, J., "CBOR Encoded Message Syntax", draft-ietf-cose-msg-10 (work in progress), February 2016.
- [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>>.
- [RFC7641] Hartke, K., "Observing Resources in the Constrained Application Protocol (CoAP)", RFC 7641, DOI 10.17487/RFC7641, September 2015, <<http://www.rfc-editor.org/info/rfc7641>>.

11.2. Informative References

- [I-D.hartke-core-e2e-security-reqs]
Selander, G., Palombini, F., Hartke, K., and L. Seitz, "Requirements for CoAP End-To-End Security", draft-hartke-core-e2e-security-reqs-00 (work in progress), March 2016.

[I-D.ietf-ace-oauth-authz]

Seitz, L., Selander, G., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "Authorization for the Internet of Things using OAuth 2.0", draft-ietf-ace-oauth-authz-01 (work in progress), February 2016.

[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-tls-tls13]

Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-11 (work in progress), December 2015.

[I-D.selander-ace-cose-ecdh]

Selander, G., Mattsson, J., and F. Palombini, "Ephemeral Diffie-Hellman Over COSE (EDHOC)", draft-selander-ace-cose-ecdh-00 (work in progress), March 2016.

[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>>.

Appendix A. Overhead

OSCOAP transforms an unprotected CoAP message to a protected CoAP message, and the protected CoAP message is larger than the unprotected CoAP message. This appendix illustrates the message expansion.

A.1. Length of the Object-Security Option

The protected CoAP message contains the COSE object. The COSE object is included in the payload if the unprotected CoAP message has payload or else in the Object-Security option. In the former case the Object-Security option is empty. So the length of the Object-Security option is either zero or the size of the COSE object, depending on whether the CoAP message has payload or not.

Length of Object-Security option = { 0, size of COSE Object }

A.2. Size of the COSE Object

The size of the COSE object is the sum of the sizes of

- o the Header parameters,
- o the Ciphertext (excluding the Tag),
- o the Tag, and
- o data incurred by the COSE format itself (including CBOR encoding).

Let's analyse the contributions one at a time:

- o The header parameters of the COSE object are the Context Identifier (Cid) and the Sequence Number (Seq) (also known as the Transaction Identifier (Tid)) if the message is a request, and Seq only if the message is a response (see Section 5).
 - * The size of Cid depends on the number of simultaneous clients, and must be chosen so that the server can uniquely identify the requesting client. For example, in the case of an ACE-based authentication and authorization model [I-D.ietf-ace-oauth-authz], the Authorization Server or the server itself can define the context identifier of all clients interacting with a particular server, in which case the size of Cid can be proportional to the logarithm of number of authorized clients.
 - + As Cids of different lengths can be used by different client, it is RECOMMENDED to start assigning Cids of length 1 byte (0x00, 0x01, ..., 0xff), and then when all 1 byte Cids are in use, start handling out Cids with a length of two bytes (0x0000, 0x0001, ..., 0xffff).
 - * The size of Seq is variable, and increases with the number of messages exchanged.
 - * As the nonce is generated from the padded Sequence Number and a previously agreed upon static IV it is not required to send the whole nonce in the message.
- o The Ciphertext, excluding the Tag, is the encryption of the payload and the encrypted options Section 4, which are present in the unprotected CoAP message.
- o The size of the Tag depends on the Algorithm. For the OSCOAP mandatory algorithm AES-CCM-64-64-128, the Tag is 8 bytes.

- o The overhead from the COSE format itself depends on the sizes of the previous fields, and is of the order of 10 bytes.

A.3. Message Expansion

The message expansion is not the size of the COSE object. The ciphertext in the COSE object is encrypted payload and options of the unprotected CoAP message - the plaintext of which is removed from the protected CoAP message. Since the size of the ciphertext is the same as the corresponding plaintext, there is no message expansion due to encryption; payload and options are just represented in a different way in the protected CoAP message:

- o The encrypted payload is in the payload of the protected CoAP message
- o The encrypted options are in the Object-Security option or within the payload.

Therefore the OSCOAP message expansion is due to Cid (if present), Seq, Tag, and COSE overhead:

$$\text{Message Overhead} = \text{Cid} + \text{Seq} + \text{Tag} + \text{COSE Overhead}$$

Figure 7: OSCOAP message expansion

A.4. Example

This section gives an example of message expansion in a request with OSCOAP.

In this example we assume an extreme 4-byte Cid, based on the assumption of an ACE deployment with billions of clients requesting access to this particular server. (A typical Cid, will be 1-2 byte as is discussed in Appendix A.2.)

- o Cid: 0xa1534e3c

In the example the sequence number is 225, requiring 1 byte to encode. (The size of Seq could be larger depending on how many messages that has been sent as is discussed in Appendix A.2.)

- o Seq: 225

The example is based on AES-CCM-64-64-128.

- o Tag is 8 bytes

The COSE object is represented in Figure 8 using CBOR's diagnostic notation.

```
[
  h'a20444a1534e3c0641e2', # protected:
                                {04:h'a1534e3c',
                                06:h'e2'}
  {},                        # unprotected: -
  Tag                        # ciphertext + 8 byte authentication tag
]
```

Figure 8: Example of message expansion

Note that the encrypted CoAP options and payload are omitted since we target the message expansion (see Appendix A.3). Therefore the size of the COSE Ciphertext equals the size of the Tag, which is 8 bytes.

The COSE object encodes to a total size of 22 bytes, which is the message expansion in this example. The COSE overhead in this example is $22 - (4 + 1 + 8) = 9$ bytes, according to the formula in Figure 7. Note that in this example two bytes in the COSE overhead are used to encode the length of Cid and the length of Seq.

Figure 9 summarizes these results.

Tid	Tag	COSE OH	Message OH
5 bytes	8 bytes	9 bytes	22 bytes

Figure 9: Message overhead for a 5-byte Tid and 8-byte Tag.

Appendix B. Examples

This section gives examples of OSCOAP. The message exchanges are made, based on the assumption that there is a security context established between client and server. For simplicity, these examples only indicate the content of the messages without going into detail of the COSE message format.

B.1. Secure Access to Actuator

Here is an example targeting the scenario in Section 3.1 of [I-D.hartke-core-e2e-security-reqs]. The example illustrates a client requesting valve 34 to be turned to position 3 (PUT /valve34 with payload value "3"), and getting a confirmation. The CoAP options Uri-Path and Payload are encrypted and integrity protected,

and the CoAP header field Code is integrity protected (see Section 4).

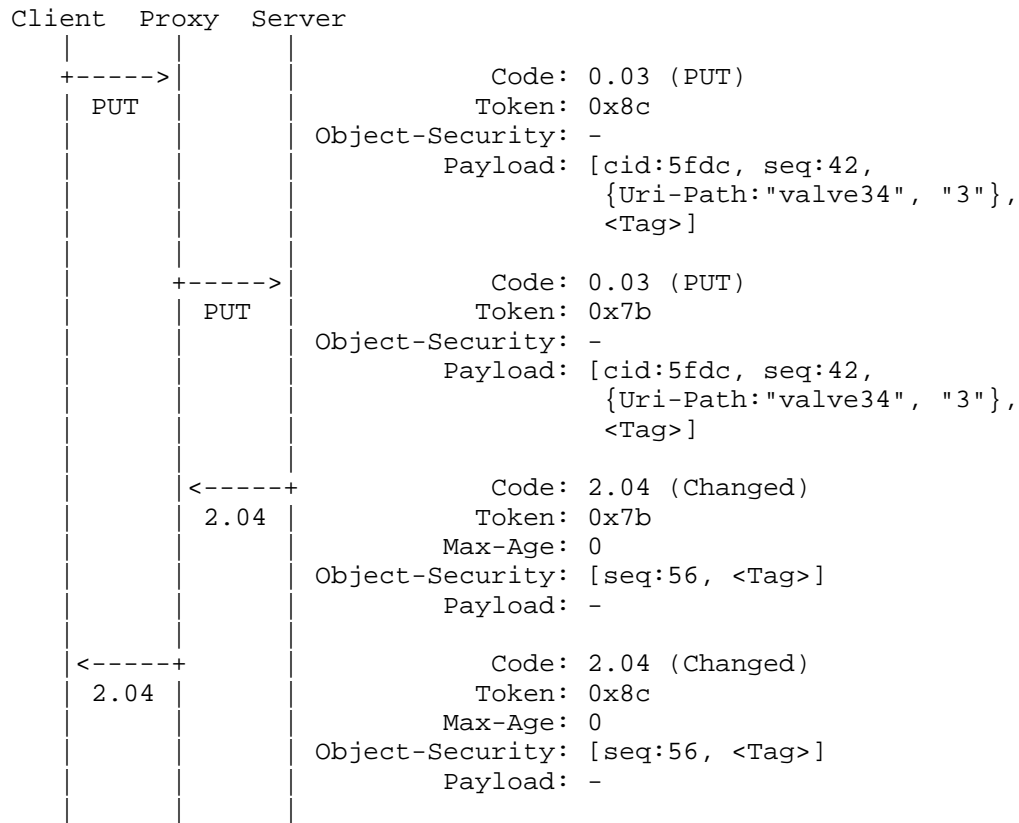


Figure 10: Indication of CoAP PUT protected with OSCOAP. The brackets [...] indicate a COSE object. The brackets { ... } indicate encrypted data.

Since the unprotected request message (PUT) has payload ("3"), the COSE object (indicated with [...]) is carried as the CoAP payload. Since the unprotected response message (Changed) has no payload, the Object-Security option carries the COSE object as its value.

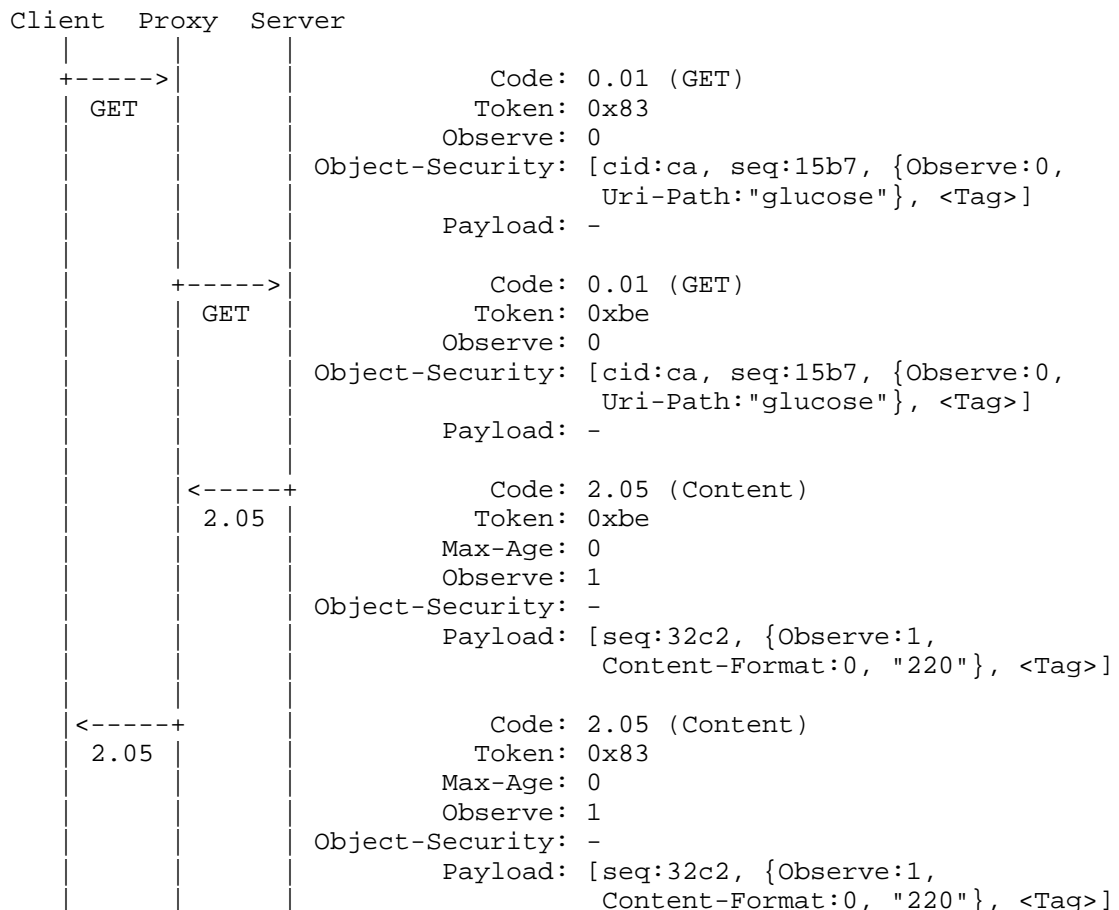
The COSE header of the request contains a Context Identifier (cid:5fdc), indicating which security context was used to protect the message and a Sequence Number (seq:42).

The option Uri-Path (valve34) and payload ("3") are formatted as indicated in Section 5, and encrypted in the COSE Ciphertext (indicated with { ... }).

The server verifies that the Sequence Number has not been received before (see Section 6.1). The client verifies that the Sequence Number has not been received before and that the response message is generated as a response to the sent request message (see Section 6.1).

B.2. Secure Subscribe to Sensor

Here is an example targeting the scenario in Section 3.2 of [I-D.hartke-core-e2e-security-reqs]. The example illustrates a client requesting subscription to a blood sugar measurement resource (GET /glucose), and first receiving the value 220 mg/dl, and then a second reading with value 180 mg/dl. The CoAP options Observe, Uri-Path, Content-Format, and Payload are encrypted and integrity protected, and the CoAP header field Code is integrity protected (see Section 4).



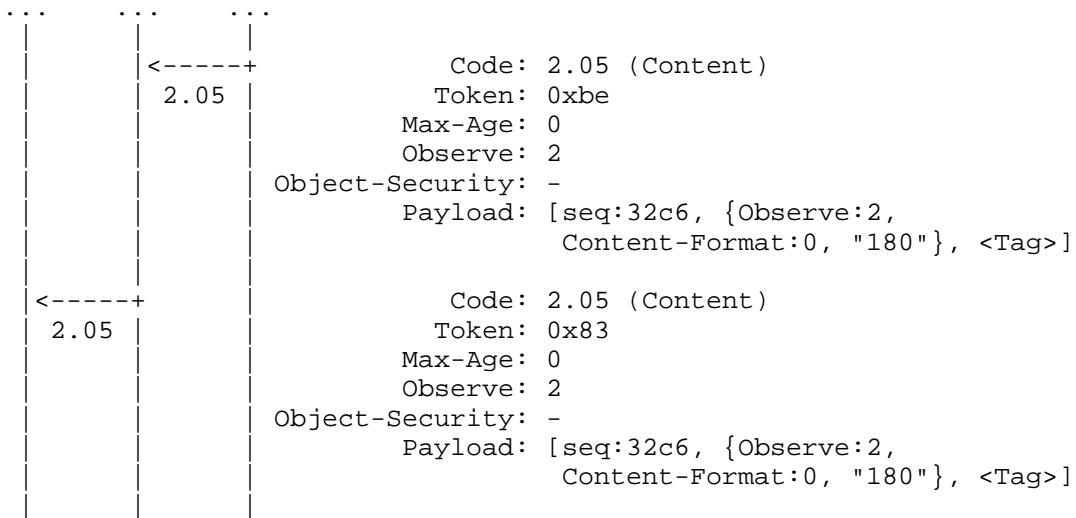


Figure 11: Indication of CoAP GET protected with OSCOAP. The brackets [...] indicates COSE object. The bracket { ... } indicates encrypted data.

Since the unprotected request message (GET) has no payload, the COSE object (indicated with [...]) is carried in the Object-Security option value. Since the unprotected response message (Content) has payload, the Object-Security option is empty, and the COSE object is carried as the payload.

The COSE header of the request contains a Context Identifier (cid:ca), indicating which security context was used to protect the message and a Sequence Number (seq:15b7).

The options Observe, Content-Format and the payload are formatted as indicated in Section 5, and encrypted in the COSE ciphertext (indicated with { ... }).

The server verifies that the Sequence Number has not been received before (see Section 6.1). The client verifies that the Sequence Number has not been received before and that the response message is generated as a response to the subscribe request.

Appendix C. Object Security of Content (OSCON)

OSCOAP protects message exchanges end-to-end between a certain client and a certain server, targeting the security requirements in Section 3.1 and 3.2 of [I-D.hartke-core-e2e-security-reqs]. In contrast, many use cases require one and the same message to be

protected for, and verified by, multiple endpoints, see Sections 3.3 - 3.5 of [I-D.hartke-core-e2e-security-reqs]. Those security requirements can be addressed by protecting essentially the payload/content of individual messages using the COSE format ([I-D.ietf-cose-msg]), rather than the entire request/response message exchange. This is referred to as Object Security of Content (OSCON).

OSCON transforms an unprotected CoAP message into a protected CoAP message in the following way: the payload of the unprotected CoAP message is wrapped by a COSE object, which replaces the payload of the unprotected CoAP message. We call the result the "protected" CoAP message.

The unprotected payload SHALL be the plaintext/payload of the COSE object. The 'protected' field of the COSE object 'Headers' SHALL include the context identifier, both for requests and responses. If the unprotected CoAP message includes a Content-Format option, then the COSE object SHALL include a protected 'content type' field, whose value is set to the unprotected message Content-Format value. The Content-Format option of the protected CoAP message SHALL be replaced with "application/oscon" (Section 9)

The COSE object SHALL be protected (encrypted) and verified (decrypted) as described in ([I-D.ietf-cose-msg]).

In the case of symmetric encryption, the same key and nonce SHALL NOT be used twice. The use of sequence numbers for partial IV as specified for OSCOAP MAY be used. of sequence numbers for replay protection as described in Section 6.1 MAY be used. The use of time stamps in the COSE header parameter 'operation time' [I-D.ietf-cose-msg] for freshness MAY be used.

OSCON SHALL NOT be used in cases where CoAP header fields (such as Code or Version) or CoAP options need to be integrity protected or encrypted. OSCON SHALL NOT be used in cases which require a secure binding between request and response.

The scenarios in Sections 3.3 - 3.5 of [I-D.hartke-core-e2e-security-reqs] assume multiple receivers for a particular content. In this case the use of symmetric keys does not provide data origin authentication. Therefore the COSE object SHOULD in general be protected with a digital signature.

C.1. Overhead OSCON

In general there are four different kinds of ciphersuites that need to be supported: message authentication code, digital signature, authenticated encryption, and symmetric encryption + digital signature. The use of digital signature is necessary for applications with many legitimate recipients of a given message, and where data origin authentication is required.

To distinguish between these different cases, the tagged structures of COSE are used (see Section 2 of [I-D.ietf-cose-msg]).

The size of the COSE message for selected algorithms are detailed in this section.

The size of the header is shown separately from the size of the MAC/signature. A 4-byte Context Identifier and a 1-byte Sequence Number are used throughout all examples, with these values:

- o Cid: 0xa1534e3c
- o Seq: 0xa3

For each scheme, we indicate the fixed length of these two parameters ("Cid+Seq" column) and of the Tag ("MAC"/"SIG"/"TAG"). The "Message OH" column shows the total expansions of the CoAP message size, while the "COSE OH" column is calculated from the previous columns following the formula in Figure 7.

Overhead incurring from CBOR encoding is also included in the COSE overhead count.

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

C.2. MAC Only

This example is based on HMAC-SHA256, with truncation to 8 bytes (HMAC 256/64).

Since the key is implicitly known by the recipient, the COSE_Mac0_Tagged structure is used (Section 6.2 of [I-D.ietf-cose-msg]).

The object in COSE encoding gives:

```

996(                                     # COSE_Mac0_Tagged
[
    h'a20444a1534e3c0641a3', # protected:
                                {04:h'a1534e3c',
                                06:h'a3'}
    {},                         # unprotected
    h'',                        # payload
    MAC                         # truncated 8-byte MAC
]
)

```

This COSE object encodes to a total size of 26 bytes.

Figure 12 summarizes these results.

Structure	Tid	MAC	COSE OH	Message OH
COSE_Mac0_Tagged	5 B	8 B	13 B	26 B

Figure 12: Message overhead for a 5-byte Tid using HMAC 256/64

C.3. Signature Only

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

Since only one signature is used, the COSE_Sign1_Tagged structure is used (Section 4.2 of [I-D.ietf-cose-msg]).

The object in COSE encoding gives:

```

997(                                     # COSE_Sign1_Tagged
[
    h'a20444a1534e3c0641a3', # protected:
                                {04:h'a1534e3c',
                                06:h'a3'}
    {},                         # unprotected
    h'',                        # payload
    SIG                         # 64-byte signature
]
)

```

This COSE object encodes to a total size of 83 bytes.

Figure 13 summarizes these results.

Structure	Tid	SIG	COSE OH	Message OH
COSE_Sign1_Tagged	5 B	64 B	14 B	83 bytes

Figure 13: Message overhead for a 5-byte Tid using 64 byte ECDSA signature.

C.4. Authenticated Encryption with Additional Data (AEAD)

This example is based on AES-CCM with the MAC truncated to 8 bytes.

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

Since the key is implicitly known by the recipient, the COSE_Encrypted_Tagged structure is used (Section 5.2 of [I-D.ietf-cose-msg]).

The object in COSE encoding gives:

```

993(                                     # COSE_Encrypted_Tagged
[
  h'a20444a1534e3c0641a3', # protected:
                           {04:h'a1534e3c',
                           06:h'a3'}
  {},                      # unprotected
  TAG                      # ciphertext + truncated 8-byte TAG
]
)

```

This COSE object encodes to a total size of 25 bytes.

Figure 14 summarizes these results.

Structure	Tid	TAG	COSE OH	Message OH
COSE_Encrypted_Tagged	5 B	8 B	12 B	25 bytes

Figure 14: Message overhead for a 5-byte Tid using AES_128_CCM_8.

C.5. Symmetric Encryption with Asymmetric Signature (SEAS)

This example is based on AES-CCM and ECDSA with 64 bytes signature. The same assumption on the security context as in Appendix C.4. COSE defines the field 'counter signature' that is used here to sign a COSE_Encrypted_Tagged message (see Section 3 of [I-D.ietf-cose-msg]).

The object in COSE encoding gives:

```

993(                                     # COSE_Encrypted_Tagged
  [
    h'a20444a1534e3c0641a3', # protected:
                                {04:h'a1534e3c',
                                06:h'a3'}
    {7:SIG},                    # unprotected:
                                07: 64 bytes signature
    TAG                        # ciphertext + truncated 8-byte TAG
  ]
)
```

This COSE object encodes to a total size of 92 bytes.

Figure 15 summarizes these results.

Structure	Tid	TAG	SIG	COSE OH	Message OH
COSE_Encrypted_Tagged	5 B	8 B	64 B	15 B	92 B

Figure 15: Message overhead for a 5-byte Tid using AES-CCM countersigned with ECDSA.

Authors' Addresses

Goeran Selander
Ericsson AB
Farogatan 6
Kista SE-16480 Stockholm
Sweden

Email: goran.selander@ericsson.com

John Mattsson
Ericsson AB
Farogatan 6
Kista SE-16480 Stockholm
Sweden

Email: john.mattsson@ericsson.com

Francesca Palombini
Ericsson AB
Farogatan 6
Kista SE-16480 Stockholm
Sweden

Email: francesca.palombini@ericsson.com

Ludwig Seitz
SICS Swedish ICT
Scheelevagen 17
Lund 22370
Sweden

Email: ludwig@sics.se