COSE Working Group
Internet-Draft
Intended status: Informational
Expires: March 24, 2016

J. Schaad
August Cellars
B. Campbell
Ping Identity

September 21, 2015

CBOR Encoded Message Syntax
draft-ietf-cose-msg-05

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 the basic security services defined for this data format. This document specifies how to do signatures, message authentication codes and encryption using this data format.

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 March 24, 2016.

Copyright Notice

Table of Contents

1. Introduction

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

The JOSE working group produced a set of documents [RFC7515][RFC7516][RFC7517][RFC7518] that defined how to perform encryption, signatures and message authentication (MAC) operations for JSON documents and then to encode the results using the JSON format [RFC7159]. This document does the same work for use with the CBOR [RFC7049] document format. While there is a strong attempt to keep the flavor of the original JOSE documents, two considerations are taken into account:

- CBOR has capabilities that are not present in JSON and should be used. One example of this is the fact that CBOR has a method of

encoding binary directly without first converting it into a base64 encoded string.

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

1.1. Design changes from JOSE

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

- Signed messages separate the concept of protected and unprotected parameters that are for the content and the signature.

- Recipient processing has been made more uniform. A recipient structure is required for all recipients rather than only for some.

- MAC messages are separated from signed messages.

- MAC messages have the ability to do use all recipient algorithms on the MAC authentication key.

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

- Combine the authentication tag for encryption algorithms with the ciphertext.

- Remove the flattened mode of encoding. Forcing the use of an array of recipients at all times forces the message size to be two bytes larger, but one gets a corresponding decrease in the implementation size that should compensate for this. [CREF1]

1.2. Requirements Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

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

1.3. CBOR Grammar

There currently is no standard CBOR grammar available for use by specifications. We therefore describe the CBOR structures in prose. There is a version of a CBOR grammar in the CBOR Data Definition Language (CDDL) [I-D.greevenbosch-appsawg-cbor-cddl]. An informational version of the CBOR grammar that reflects what is in the prose can be found in Appendix A. CDDL has not been fixed, so this grammar may will only work with the version of CDDL at the time of publishing.

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 early versions CDDL. In this specification the following primitive types are used:

bstr - byte string (major type 2).

int - an unsigned integer or a negative integer.

nil - a null value (tag 7.22).

nint - a negative integer (major type 1).

tstr - a UTF-8 text string (major type 3).

uint - an unsigned integer (major type 0).

Text from here to start of next section to be removed

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

```
start = COSE_MSG / COSE_Tagged_MSG / COSE_Key / COSE_KeySet
```

1.4. CBOR Related Terminology

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

cryptographic key, we use the term "label" for this usage of either an integer or a string to identify map keys and choose data items.

Text from here to start of next section to be removed

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

1.5. Document Terminology

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

Byte is a synonym for octet.

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

1.6. Mandatory to Implement Algorithms

One of the issues that needs to be addressed is a requirement that a standard specify a set of algorithms that are required to be implemented. [CREF3] This is done to promote interoperability as it provides a minimal set of algorithms that all devices can be sure will exist at both ends. However, we have elected not to specify a set of mandatory algorithms in this document.

It is expected that COSE is going to be used in a wide variety of applications and on a wide variety of devices. Many of the constrained devices are going to be setup to used a small fixed set of algorithms, and this set of algorithms may not match those available on a device. We therefore have deferred to the application protocols the decision of what to specify for mandatory algorithms.

Since the set of algorithms in an environment of constrained devices may depend on what the set of devices are and how long they have been in operation, we want to highlight that application protocols will need to specify some type of discovery method of algorithm capabilities. The discovery method may be as simple as requiring preconfiguration of the set of algorithms to providing a discovery method built into the protocol. S/MIME provided a number of different ways to approach the problem:

- Advertising in the message (S/MIME capabilities) [RFC5751].

- Advertising in the certificate (capabilities extension) [RFC4262]

- Minimum requirements for the S/MIME which have been updated over time [RFC2633][RFC5751]

2. The COSE_MSG structure

The COSE_MSG structure is a top level CBOR object that corresponds to the DataContent type in the Cryptographic Message Syntax (CMS) [RFC5652]. [CREF4] This structure allows for a top level message to be sent that could be any of the different security services. The security service is identified within the message.

The COSE_Tagged_MSG CBOR type takes the COSE_MSG and prepends a CBOR tag of TBD1 to the encoding of COSE_MSG. By having both a tagged and untagged version of the COSE_MSG structure, it becomes easy to either use COSE_MSG as a top level object or embedded in another object. The tagged version allows for a method of placing the COSE_MSG structure into a choice, using a consistent tag value to determine that this is a COSE object.

The existence of the COSE_MSG and COSE_Tagged_MSG CBOR data types are not intended to prevent protocols from using the individual security primitives directly. Where only a single service is required, that structure can be used directly.

Each of the top-level security objects use a CBOR array as the base structure. For each of the top-level security objects, the first field is a 'msg_type'. The CBOR type for a 'msg_type' is 'int'. The 'msg_type' is defined to distinguish between the different structures when they appear as part of a COSE_MSG object. [CREF5] [CREF6] [CREF7]

The message types defined in this document are:

0 - Reserved.

1 - Signed Message.

2 - Enveloped Message

3 - Authenticated Message (MACed message)

4 - Encrypted Message

Implementations MUST be prepared to find an integer in this field that does not correspond to the values 1 to 3. If a message type is found then the client does not support the associated security object, the client MUST stop attempting to process the structure and fail. The value of 0 is reserved and not assigned to a security

object. If the value of 0 is found, then clients MUST fail processing the structure. Implementations need to recognize that the set of values might be extended at a later date, but they should not provide a security service based on guesses of what the security object might be.

Text from here to start of next section to be removed

```
COSE_MSG = COSE_Sign /
    COSE_enveloped /
    COSE_encryptData /
    COSE_mac

COSE_Tagged_MSG = #6.999(COSE_MSG)   ; Replace 999 with TBD1

; msg_type values
msg_type_reserved=0
msg_type_signed=1
msg_type_enveloped=2
msg_type_mac=3
msg_type_encryptData=4
```

3. Header Parameters

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

Both buckets are implemented as CBOR maps. The map key is a 'label' (Section 1.4). The value portion is dependent on the definition for the label. Both maps use the same set of label/value pairs. The integer and string values for labels has been divided into several sections with a standard range, a private range, and a range that is dependent on the algorithm selected. The defined labels can be found in the 'COSE Header Parameters' IANA registry (Section 15.2).

Two buckets are provided for each layer:

protected contains parameters about the current layer that are to be cryptographically protected. This bucket MUST be empty if it is

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

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

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

The buckets are present in all of the security objects defined in this document. The fields in order are the 'protected' bucket (as a CBOR 'bstr' type) and then the 'unprotected' bucket (as a CBOR 'map' type). The presence of both buckets is required. The parameters that go into the buckets come from the IANA "COSE Header Parameters" (Section 15.2). Some common parameters are defined in the next section, but a number of parameters are defined throughout this document.

Text from here to start of next section to be removed [CREF8]

```
header_map = {+ label => any }

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

3.1. Common COSE Headers Parameters

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

The set of header parameters defined in this section are:

alg This parameter is used to indicate the algorithm used for the security processing. This parameter MUST be present at each level of a signed, encrypted or authenticated message. The value is taken from the 'COSE Algorithm Registry' (see Section 15.4).

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

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

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

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

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

kid This parameter one of the ways that can be used to find the key to be used. The value of this parameter is matched against the 'kid' member in a COSE_Key structure. Applications MUST NOT

assume that 'kid' values are unique. There may be more than one key with the same 'kid' value, it may be required that all of the keys need to be checked to find the correct one. The internal structure of 'kid' values is not defined and generally cannot be relied on by applications. Key identifier values are hints about which key to use, they are not directly a security critical field, for this reason they can be placed in the unprotected headers bucket.

nonce This parameter holds either a nonce or Initialization Vector value. The value can be used either as a counter value for a protocol or as an IV for an algorithm.

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

name	label	value type	value registry	description
alg	1	int / tstr	COSE Algorithm Registry	Integers are taken from table --POINT TO REGISTRY--
crit	2	[+ label]	COSE Header Label Registry	integer values are from -- POINT TO REGISTRY --
content type	3	tstr / int	CoAP Content-Formats or Media Types registry	Value is either a Media Type or an integer from the CoAP Content Format registry
kid	4	bstr		key identifier
nonce	5	bstr		Nonce or Initialization Vector (IV)
counter signature	6	COSE_signature		CBOR encoded signature structure
zip	*	int / tstr		Integers are taken from the table --POINT TO REGISTRY--

Table 1: Common Header Parameters

OPEN ISSUES:

1. Do we want to have a zip/compression header standardized in this document?

2. I am currently torn on the question "Should epk and iv/nonce be algorithm specific or generic headers?" They are really specific to an algorithm and can potentially be defined in different ways for different algorithms. As an example, it would make sense to defined nonce for CCM and GCM modes that can have the leading zero bytes stripped, while for other algorithms this might be undesirable.

3. We might want to define some additional items. What are they? A possible example would be a sequence number as this might be common. On the other hand, this is the type of things that is frequently used as the nonce in some places and thus should not be used in the same way. Other items might be challenge/response fields for freshness as these are likely to be common.

4. Signing Structure

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

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

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

msg_type identifies this as providing the signed security service. The value MUST be msg_type_signed (1).

protected is described in Section 3.

unprotected is described in Section 3.

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

signatures is an array of signature items. Each of these items uses the COSE_signature structure for its representation.

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

protected is described in Section 3.

unprotected is described in Section 3.

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

Text from here to start of next section to be removed

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

COSE_signature =  [
    Headers,
    signature : bstr
]
```

4.1. Externally Supplied Data

One of the features that we supply in the COSE document is the ability for applications to provide additional data to be authenticated as part of the security, but that is not carried as part of the COSE object. The primary reason for supporting this can be seen by looking at the CoAP message struture [RFC7252] where the facility exists for options to be carried before the payload. An example of data that can be placed in this location would be transaction ids and nonces to check for replay protection. If the data is in the options section, then it is available for routers to help in performing the replay detection and prevention. However, it may also be desired to protect these values so that they cannot be modified in transit. This is the purpose of the externally supplied data field.

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

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

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

4.2. Signing and Verification Process

In order to create a signature, a consistent byte stream is needed in order to process. This document uses a CBOR array to construct the byte stream to be processed. The fields of the array in order are:

1. The body protected attributes. This is a bstr type containing the protected attributes of the body.

2. The signature protected attributes. This is a bstr type containing the protected attributes of the signature.

3. The external protected attributes. This is a bstr type containing the protected attributes external to the COSE_Signature structure.

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

How to compute a signature:

1. Create a CBOR array and populate it with the appropriate fields. For body_protected and sign_protected, if the fields are not present in their corresponding maps, an bstr of length zero is used.

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

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

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

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

How to verify a signature:

1. Create a Sig_structure object and populate it with the appropriate fields. For body_protected and sign_protected, if the fields are not present in their corresponding maps, an bstr of length zero is used.

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

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

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

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

Text from here to start of next section to be removed

The COSE structure used to create the byte stream to be signed uses the following CDDL grammar structure:

```
Sig_structure = [
    body_protected: bstr,
    sign_protected: bstr,
    external_aad: bstr,
    payload: bstr
]
```

4.3. Computing Counter Signatures

Counter signatures provide a method of having a different signature occur on some piece of content. This is normally used to provide a signature on a signature allowing for a proof that a signature existed at a given time. In this document we allow for counter signatures to exist in a greater number of environments. A counter signature can exist, for example, on a COSE_encyptedData object and allow for a signature to be present on the encrypted content of a message.

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

While one can create a counter signature for a COSE_Sign structure, there is not much of a point to doing so. It is equivalent to create a new COSE_signature structure and placing it in the signatures array. It is strongly suggested that it not be done, but it is not banned.

5. Encryption objects

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

5.1. Enveloped COSE structure

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

In COSE, the same techniques and structures for encrypting both the plain text and the keys used to protect the text. This is different from the approach used by both [RFC5652] and [RFC7516] where different structures are used for the content layer and for the recipient layer.

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

msg_type identifies this as providing the encrypted security service. The value MUST be msg_type_encrypted (2).

protected is described in Section 3.

unprotected is described in Section 3.

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

recipients contains an array of recipient information structures. The type for the recipient information structure is a COSE_recipient.

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

protected is described in Section 3.

unprotected is described in Section 3.

ciphertext contains the encrypted key encoded as a bstr. If there is not an encrypted key, then this field is encoded as a nil type.

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

Text from here to start of next section to be removed

```
COSE_enveloped = [
    msg_type: msg_type_enveloped,
    COSE_encrypt_fields
    recipients: [+COSE_recipient]
]

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

COSE_recipient = [
    COSE_encrypt_fields
    ? recipients: [+COSE_recipient]
]
```

5.1.1. Recipient Algorithm Classes

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

none: The CEK is the same as as the identified previously distributed symmetric key or derived from a previously distributed secret.

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

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 in the recipient's public key

passwords: the CEK is encrypted in a key-encryption key that is derived from a password.

5.2. Encrypted COSE structure

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

The CDDL grammar structure for encrypted data is:

```
COSE_encryptData = [
    msg_type: msg_type_encryptData,
    COSE_encrypt_fields
]
```

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

msg_type identifies this as providing the encrypted data security service. This value MUST be mg_type_encrypted (4).

protected is described in Section 3.

unprotected is described in Section 3.

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

5.3. Encryption Algorithm for AEAD algorithms

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

1. Copy the protected header field from the message to be sent.

2. If the application has supplied external additional authenticated data to be included in the computation, then it is placed in the 'external_aad' field. If no data was supplied, then a zero length binary value is used. (See Section 4.1 for application guidance on constructing this field.)

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

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

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

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

   Other: The key is randomly generated.

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

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

Text from here to start of next section to be removed

```
Enc_structure = [
    protected: bstr,
    external_aad: bstr
]
```

5.4. Encryption algorithm for AE algorithms

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

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

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

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

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

   Other: The key is randomly generated.

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

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

6. MAC objects

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

When using MAC operations, there are two modes in which it can be used. The first is just a check that the content has not been changed since the MAC was computed. Any class of recipient algorithm can be used for this purpose. The second mode is to both check that the content has not been changed since the MAC was computed, and to use 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 MACing the message can be validated by a key binding. (The binding of identity assumes that there are only two parties involved and you did not send the message yourself.)

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

msg_type identifies this as providing the encrypted security service. The value MUST be msg_type_mac (3).

protected is described in Section 3.

unprotected is described in Section 3.

payload contains the serialized content to be MACed. If the payload is not present in the message, the application is required to supply the payload separately. The payload is wrapped in a bstr to ensure that it is transported without changes. If the payload is transported separately, then a null CBOR object is placed in this location and it is the responsibility of the application to ensure that it will be transported without changes.

tag contains the MAC value.

recipients contains the recipient information. See the description under COSE_Encryption for more info.

Text from here to start of next section to be removed

```
COSE_mac = [
   msg_type: msg_type_mac,
   Headers,
   payload: bstr / nil,
   tag: bstr,
   recipients: [+COSE_recipient]
]
```

6.1. How to compute a MAC

How to compute a MAC:

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

2. If the application has supplied external authenticated data, encode it as a binary value and place in the MAC_structure. If there is no external authenticated data, then use a zero length 'bstr'. (See Section 4.1 for application guidance on constructing this field.)

3. Encode the MAC_structure using a canonical CBOR encoder. The resulting bytes is the value to compute the MAC on.

4. Compute the MAC and place the result in the 'tag' field of the COSE_mac structure.

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

Text from here to start of next section to be removed

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

7. Key Structure

A COSE Key structure is built on a CBOR map object. The set of common parameters that can appear in a COSE Key can be found in the IANA registry 'COSE Key Common Parameter Registry' (Section 15.5). Additional parameters defined for specific key types can be found in the IANA registry 'COSE Key Type Parameters' (Section 15.6).

A COSE Key Set uses a CBOR array object as it's underlying type. The values of the array elements are COSE Keys. A Key Set MUST have at least one element in the array. [CREF9]

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

Text from here to start of next section to be removed

The CDDL grammar describing a COSE_Key and COSE_KeySet is: [CREF10]

```
COSE_Key = {
    key_kty => tstr / int,
    ? key_ops => [+ (tstr / int) ],
    ? key_alg => tstr / int,
    ? key_kid => bstr,
    * 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 2 provides a summary of the parameters defined in this section. There are also a set of parameters that are defined for a specific key type. Key type specific parameters can be found in Section 13.

name	label	CBOR type	registry	description
kty	1	tstr / int	COSE General Values	Identification of the key type
key_ops	4	[* (tstr/int)]		Restrict set of permissible operations
alg	3	tstr / int	COSE Algorithm Values	Key usage restriction to this algorithm
kid	2	bstr		Key Identification value - match to kid in message
use	*	tstr		deprecated - don't use

Table 2: Key Map Labels

kty: This parameter is used to identify the family of keys for this structure, and thus the set of key type specific parameters to be found. The set of values can be found in Table 20. This parameter MUST be present in a key object. Implementations MUST verify that the key type is appropriate for the algorithm being processed. The key type MUST be included as part of a trust decision process.

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

kid: This parameter is used to give an identifier for a key. The identifier is not structured and can be anything from a user provided string to a value computed on the public portion of the key. This field is intended for matching against a 'kid'

parameter in a message in order to filter down the set of keys that need to be checked.

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

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.
key agree	7	The key is used for key agreement.

Table 3: Key Operation Values

Text from here to start of next section to be removed

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

```
;key_labels
key_kty=1
key_kid=2
key_alg=3
key_ops=4

;key_ops values
key_ops_sign=1
key_ops_verify=2
key_ops_encrypt=3
key_ops_decrypt=4
key_ops_wrap=5
key_ops_unwrap=6
key_ops_agree=7
```

8. Signature Algorithms

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

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

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

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

signature algorithms are going to be required to consume the same number of bytes of message content.

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

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

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

8.1. ECDSA

ECDSA [DSS] defines a signature algorithm using ECC.

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

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

name	value	hash	description
ES256	-7	SHA-256	ECDSA w/ SHA-256
ES384	-8	SHA-384	ECDSA w/ SHA-384
ES512	-9	SHA-512	ECDSA w/ SHA-512

Table 4: ECDSA Algorithm Values

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

In order to promote interoperability, it is suggested that SHA-256 be used only with curve P-256, SHA-384 be used only with curve P-384 and

SHA-512 be used with curve P-521. This is aligned with the recommendation in Section 4 of [RFC5480].

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

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

8.1.1. Security Considerations

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

System which have poor random number generation can leak their keys by signing two different messages with the same value of 'k'. [RFC6979] provides a method to deal with this problem by making 'k' be deterministic based on the message content rather than randomly generated. Applications which specify ECDSA should evaluate the ability to get good random number generation and require this when it is not possible. Note: Use of this technique a good idea even when good random number generation exists. Doing so both reduces the possiblity of having the same value of 'k' in two signature operations, but allows for reproducable signature values which helps testing.

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

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

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

8.2. RSASSA-PSS

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

The RSASSA-PSS signature algorithm is parametized 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 following 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 5.

name	value	hash	salt length	description
PS256	-26	SHA-256	32	RSASSA-PSS w/ SHA-256
PS384	-27	SHA-384	48	RSASSA-PSS w/ SHA-384
PS512	-28	SHA-512	64	RSASSA-PSS w/ SHA-512

Table 5: RSASSA-PSS Algorithm Values

8.2.1. 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.

9. Message Authentication (MAC) Algorithms

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

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

```
tag = 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 for each of these two constructions.

9.1. Hash-based Message Authentication Codes (HMAC)

The Hash-base 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 vindicated as, while the security of hash algorithms such as MD5 has decreased over time, the security of HMAC combined with MD5 has not yet been shown to be compromised [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 when truncating, however the security strength of the message tag is correspondingly reduced in strength. When truncating, the left most tag length bits are kept and transmitted.

The algorithm defined in this document can be found in Table 6.

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

Table 6: HMAC Algorithm Values

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

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

9.1.1. Security Considerations

HMAC has proved to be resistant even when used with weakening hash algorithms. The current best method appears to be a brute force

attack on the key. This means that key size is going to be directly related to the security of an HMAC operation.

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

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

AES-CBC-MAC is parameterized by the key length, the authentication tag length and the IV used. For all of these algorithms, the IV is fixed to all zeros. We provide an array of algorithms for various key lengths and tag lengths. The algorithms defined in this document are found in Table 7.

name	value	key length	tag length	description
AES-MAC 128/64	*	128	64	AES-MAC 128 bit key, 64-bit tag
AES-MAC 256/64	*	256	64	AES-MAC 256 bit key, 64-bit tag
AES-MAC 128/128	*	128	128	AES-MAC 128 bit key, 128-bit tag
AES-MAC 256/128	*	256	128	AES-MAC 256 bit key, 128-bit tag

Table 7: AES-MAC Algorithm Values

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

9.2.1. Security Considerations

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

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

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

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

10. Content Encryption Algorithms

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

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

```
ciphertext = Encrypt(message content, key, additional data)

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

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

10.1. AES GCM

The GCM mode is 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 a an AEAD cipher.

The GCM mode is parameterized with by the size of the authentication tag. 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 8.

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

Table 8: Algorithm Value for AES-GCM

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

10.1.1. Security Considerations

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

- The key and nonce pair MUST be unique for every message encrypted.

- The total amount of data encrypted MUST NOT exceed 2^39 - 256 bits . An explicit check is required only in environments where it is expected that it might be exceeded.

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 constrainted devices.

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

It is unfortunate that the specification for CCM specified L and M as a count of bytes rather than a count of bits. This leads to possible misunderstandings where AES-CCM-8 is frequently used to refer to a version of CCM mode where the size of the authentication is 64-bits and not 8-bits. These values have traditionally been specified as bit counts rather than byte counts. This document will follow the

tradition of using bit counts so that it is easier to compare the different algorithms presented in this document.

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

The following values are used for L:

16-bits (2) limits messages to 2^16 bytes (64Kbyte) in length. This sufficently long for messages in the constrainted 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 byes 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 an 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 an 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	30	64	64	128	AES-CCM mode 128-bit key, 64-bit tag, 7-byte nonce
AES-CCM-64-64-256	31	64	64	256	AES-CCM mode 256-bit key, 64-bit tag, 7-byte nonce
AES-CCM-16-128-128	12	16	128	128	AES-CCM mode 128-bit key, 128-bit tag, 13-byte nonce
AES-CCM-16-128-256	13	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 9: Algorithm Values for AES-CCM

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

10.2.1. Security Considerations

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

- The key and nonce pair MUST be unique for every message encrypted.

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

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

10.3. ChaCha20 and Poly1305

ChaCha20 and Poly1305 combined together is a new AEAD mode that is defined in [RFC7539]. This is a new algorithm defined to be a cipher which 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 an 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 10.

name	value	description
ChaCha20/Poly1305	11	ChaCha20/Poly1305 w/ 256-bit key

Table 10: Algorithm Value for AES-GCM

Keys may be obtained either from a key structure or from a recipient structure. If the key obtained from a key structure, the key type MUST be 'Symmetric'. Implementations creating and validating MAC

values MUST validate that the key type, key length and algorithm are correct and appropriate for the entities involved.

10.3.1. Security Considerations

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

11. Key Derivation Functions (KDF)

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

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

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

- Secrets which are not random: This is the type of secret that people generate for things like passwords.

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

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

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

11.1. HMAC-based Extract-and-Expand Key Derivation Function (HKDF)

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

The HKDF algorithm is defined to take a number of inputs These inputs are:

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

salt - an optional public value that is used to change the generation process. If specified, the salt is carried using the 'salt' algorithm parameter. While [RFC5869] suggests that the length of the salt be the same as the length of the underlying hash value, any amount of salt will improve the security as different key values will be generated. A parameter to carry the salt is defined in Table 12. This parameter is protected by being included in the key computation and does not need to be separately authenticated.

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

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

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

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

The algorithms defined in this document are found in Table 11

name	hash	Skip extract	context
HKDF SHA-256	SHA-256	no	XXX
HKDF SHA-512	SHA-512	no	XXX
HKDF AES-MAC-128	AES-CBC-128	yes	HKDF using AES-MAC as the PRF w/ 128-bit key
HKDF AES-MAC-256	AES-CBC-128	yes	HKDF using AES-MAC as the PRF w/ 256-bit key

Table 11: HKDF algorithms

name	label	type	description
salt	-20	bstr	Random salt

Table 12: 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 of 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 JOSE encoding. [CREF11]

The context information structure refers to PartyU and PartyV as the two parties which are doing the key derivation. Unless the application protocol defines differently, we assign PartyU to the entity that is creating the message and PartyV to the entity that is receiving the message. By doing this association, different keys will be derived for each direction as the context information is different in each direction.

Application protocols are free to define the roles differently. For example, they could assign the PartyU role to the entity that

initiates the connection and allow directly sending multiple messages over the connection in both directions without changing the role information.

The use of a transaction identifier, either in one of the supplemental fields or as the salt if one is using HKDF, ensures that a unique key is generated for each set of transactions. Combining nonce fields with the transaction identifier provides a method so that a different key is used for each message in each direction.

The context structure is built from information that is known to both entities. Some of the information is known only to the two entities, some is implied based on the application and some is explicitly transported as part of the message. The information that can be carried in the message, parameters have been defined and can be found in Table 13. These parameters are designed to be placed in the unprotected bucket of the recipient structure. (They do not need to be in the protected bucket since they already are included in the cryptographic computation by virtue of being included in the context structure.)

We encode the context specific information using a CBOR array type. The fields in the array are:

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

PartyUInfo This field holds information about party U. The PartyUInfo is encoded as a CBOR struture. 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 they send or receive. The second way is for a protocol to assign identities is to use a name based on a naming system (i.e. DNS, X.509 names).

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

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

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

PartyVInfo M00TODO: Copy down from PartyUInfo when that text is ready.

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

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

protected This field contains the protected parameter field.

other The field other is for free form data defined by the application. An example is that an application could defined 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 include.

SuppPrivInfo This field contains private information that is mutually known information. An example of this information would be a pre-existing shared secret. The field is optional and will only be present if the application defines a structure for this information. Applications that define this SHOULD use CBOR to encode the data so that types and lengths are correctly include.

name	label	type	description
PartyU identity	-21	bstr	Party U identity Information
PartyU nonce	-22	bstr / int	Party U provided nonce
PartyU other	-23	bstr	Party U other provided information
PartyV identity	-24	bstr	Party V identity Information
PartyV nonce	-25	bstr / int	Party V provided nonce
PartyV other	-26	bstr	Party V other provided information

Table 13: Context Algorithm Parameters

Text from here to start of next section to be removed

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

12. Recipient Algorithm Classes

Recipient algorithms can be defined into a number of different classes. COSE has the ability to support many classes of recipient algorithms. In this section, a number of classes are listed and then a set of algorithms are specified for each of the classes. The names of the recipient algorithm classes used here are the same as are defined in [RFC7517]. Other specifications use different terms for the recipient algorithm classes or do not support some of our recipient algorithm classes.

12.1. Direct Encryption

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

The COSE_encrypt structure for the recipient is organized as follows:

- The 'protected' field MUST be a zero length item if it is not used in the computation of the content key.
- The 'alg' parameter MUST be present.
- A parameter identifying the shared secret SHOULD be present.
- The 'ciphertext' field MUST be a zero length item.

- The 'recipients' field MUST be absent.

12.1.1. Direct Key

This recipient algorithm is the simplest, the supplied 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 14.

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 14: Direct Key

12.1.1.1. Security Considerations

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

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

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

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

12.1.2. Direct Key with KDF

These recipient algorithms take a common shared secret between the two parties and applies the HKDF function (Section 11.1) using the context structure defined in Section 11.2 to transform the shared secret into the necessary key. Either the 'salt' parameter of HKDF or the partyU 'nonce' parameter of the context structure MUST be present. This parameter can be generated either randomly or

deterministically, the requirement is that it be a unique value for the key pair in question.

If the salt/nonce value is generated randomly, then it is suggested that the length of the random value be the same length as the hash function underlying HKDF. While there is no way to guarantee that it will be unique, there is a high probability that it will be unique. If the salt/nonce value is generated deterministically, it can be guaranteed to be unique and thus there is no length requirement.

A new IV must be used if the same key is used in more than one message. The IV can be modified in a predictable manner, a random manner or an unpredictable manner. One unpredictable manner that can be used is to use the HKDF function to generate the IV. If HKDF is used for generating the IV, the algorithm identifier is set to "IV-GENERATION".

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

The set of algorithms defined in this document can be found in Table 15.

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

Table 15: Direct Key

12.1.2.1. Security Considerations

The shared secret need to have some method to be regularly updated over time. The shared secret is forming the basis of trust, although not used directly it should still be subject to scheduled rotation.

12.2. Key Wrapping

In key wrapping mode, the CEK is randomly generated and that key is then encrypted by a shared secret between the sender and the recipient. All of the currently defined key wrapping algorithms for JOSE (and thus for COSE) are AE algorithms. Key wrapping mode is considered to be superior to direct encryption if the system has any capability for doing random key generation. This is because the shared key is used to wrap random data rather than data has some degree of organization and may in fact be repeating the same content.

The COSE_encrypt structure for the recipient is organized as follows:

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

- The 'recipients' field is normally absent, but can be used. Applications MUST deal with a recipients field present, not being able to decrypt that recipient is an acceptable way of dealing with it. Failing to process the message is not an acceptable way of dealing with it.

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

- At a minimum, the 'unprotected' field MUST contain the 'alg' parameter and SHOULD contain a parameter identifying the shared secret.

12.2.1. AES Key Wrapping

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

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

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 16: AES Key Wrap Algorithm Values

12.2.1.1. Security Considerations for AES-KW

The shared secret need to have some method to be regularly updated over time. The shared secret is forming the basis of trust, although not used directly it should still be subject to scheduled rotation.

12.3. Key Encryption

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

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

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

12.3.1. RSAES-OAEP

RSAES-OAEP is an asymmetric key encryption algorithm. The defintion 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:

- mgf is always set to MFG1 from [RFC3447] and uses the same hash function as h.

- P is always set to the empty octet string.

Table 17 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 17: RSAES-OAEP Algorithm Values

The key type MUST be 'RSA'.

12.3.1.1. 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.

12.4. Direct Key Agreement

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

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

Two variants of DH that are easily supported are:

- Ephemeral-Static DH: where the sender of the message creates a one time DH key and uses a static key for the recipient. The use of the ephemeral sender key means that no additional random input is needed as this is randomly generated for each message.

  Static-Static DH: where a static key is used for both the sender and the recipient. The use of static keys allows for recipient to get a weak version of data origination for the message. When static-static key agreement is used, then some piece of unique data is require to ensure that a different key is created for each message

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

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

The COSE_encrypt structure for the recipient is organized as follows:

- The 'protected' field MUST be absent.

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

- The 'unprotected' field MUST contain the 'epk' parameter.

12.4.1. ECDH

The basic mathematics for Elliptic Curve Diffie-Hellman can be found in [RFC6090]. Two new curves have been defined in [I-D.irtf-cfrg-curves].

ECDH is parameterized by the following:

- Curve Type/Curve: The curve selected controls not only the size of the shared secret, but the mathematics for computing the shared secret. The curve selected also controls how a point in the curve is represented and what happens for the identity points on the curve. In this specification we allow for a number of different curves to be used. The curves are defined in Table 21.

Since the only the math is changed by changing the curve, the curve is not fixed for any of the algorithm identifiers we define, instead it is defined by the points used.

- Ephemeral-static or static-static: The key agreement process may be done using either a static or an ephemeral key at the senders side. When using ephemeral keys, the sender MUST generate a new ephemeral key for every key agreement operation. The ephemeral key is placed in in the 'ephemeral key' parameter and MUST be present for all algorithm identifiers which use ephemeral keys. When using static keys, the sender MUST either generate a new random value placed in either in the KDF parameters or the context structure. For the KDF functions used, this means either in the 'salt' parameter for HKDF (Table 12) or in in the 'PartyU nonce' parameter for the context struture (Table 13) MUST be present. (Both may be present if desired.) The value in the parameter MUST be unique for the key pair being used. It is acceptable to use a global counter which is incremented for every static-static operation and use the resulting value. When using static keys, the static key needs to be identified to the recipient. The static key can be identified either by providing the key ('static key') or by providing a key identifier for the static key ('static key id'). Both of these parameters are defined in Table 19

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

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

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

name	value	KDF	Ephemeral-Static	Key Wrap	description

ECDH-ES + HKDF-256	50	HKDF - SHA -256	yes	none	ECDH ES w/ HKDF - generate key directly
ECDH-ES + HKDF-512	51	HKDF - SHA -256	yes	none	ECDH ES w/ HKDF - generate key directly
ECDH-SS + HKDF-256	52	HKDF - SHA -256	no	none	ECDH ES w/ HKDF - generate key directly
ECDH-SS + HKDF-512	53	HKDF - SHA -256	no	none	ECDH ES w/ HKDF - generate key directly
ECDH-ES+A128KW	54	HKDF - SHA -256	yes	A128KW	ECDH ES w/ Concat KDF and AES Key wrap w/ 128 bit key
ECDH-ES+A192KW	55	HKDF - SHA -256	yes	A192KW	ECDH ES w/ Concat KDF and AES Key wrap w/ 192 bit key
ECDH-ES+A256KW	56	HKDF - SHA -256	yes	A256KW	ECDH ES w/ Concat KDF and AES Key wrap w/ 256 bit key
ECDH-SS+A128KW	57	HKDF - SHA	no	A128KW	ECDH SS w/ Concat KDF

		-256				and AES Key wrap w/ 128 bit key
ECDH-SS+A192KW	58	HKDF - SHA -256	no		A192KW	ECDH SS w/ Concat KDF and AES Key wrap w/ 192 bit key
ECDH-SS+A256KW	59	HKDF - SHA -256	no		A256KW	ECDH SS w/ Concat KDF and AES Key wrap w/ 256 bit key

Table 18: ECDH Algorithm Values

name	label	type	algorithm	description
ephemeral key	-1	COSE_Key	ECDH-ES	Ephemeral Public key for the sender
static key	-2	COSE_Key	ECDH-ES	Static Public key for the sender
static key id	-3	bstr	ECDH-SS	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.

12.5. Key Agreement with KDF

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

The COSE_encrypt structure for the recipient is organized as follows:

- The 'protected' field is fed into the KDF context structure.
- The plain text to be encrypted is the key from next layer down (usually the content layer).
- The 'alg' parameter MUST be present in the layer.
- A parameter identifying the recipient's key SHOULD be present. A parameter identifying the senders key SHOULD be present.

12.5.1. ECDH

These algorithms are defined in Table 18.

13. Keys

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

For each of the key types, we define both public and private members. The public members are what is transmitted to others for their usage. We define private members mainly for the purpose of archival of keys by individuals. However, there are some circumstances 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
EC1	1	Elliptic Curve Keys w/ X Coordinate only
EC2	2	Elliptic Curve Keys w/ X,Y Coordinate pair
RSA	3	RSA Keys
Symmetric	4	Symmetric Keys
Reserved	0	This value is reserved

Table 20: Key Type Values

13.1. Elliptic Curve Keys

Two different key structures are being defined for Elliptic Curve keys. One version uses both an x and a y coordinate, potentially with point compression. This is the traditional EC point representation that is used in [RFC5480]. The other version uses only the x coordinate as the y coordinate is either to be recomputed or not needed for the key agreement operation. An example of this is Curve25519 [I-D.irtf-cfrg-curves]. [CREF12]

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
Curve25519	EC1	1	Curve 25519
Curve448	EC1	2	Curve 448

Table 21: EC Curves

13.1.1. Single Coordinate Curves

One class of Elliptic Curve mathematics allows for a point to be completely defined using the curve and the x coordinate of the point on the curve. The two curves that are initially setup to use is point format are Curve 25519 and Curve 448 which are defined in [I-D.irtf-cfrg-curves].

For EC keys with only the x coordinates, the 'kty' member is set to 1 (EC1). The key parameters defined in this section are summarized in Table 22. The members that are defined for this key type are:

crv contains an identifier of the curve to be used with the key. [CREF13] The curves defined in this document for this key type can be found in Table 21. 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 22: EC Key Parameters

13.1.2. 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 recommend in the IETF due to potential IPR issues with Certicom. 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 21. Other curves may be registered in the future and private curves can be used as well.

x contains the x coordinate for the EC point. The integer is converted to an octet string as defined in [SEC1]. Zero octets MUST NOT be removed from the front of the octet string. [CREF14]

y contains either the sign bit or the value of y coordinate for the EC point. For the value, the integer is converted to an octet string as defined in [SEC1]. Zero octets MUST NOT be removed from the front of the octet string. For the sign bit, the value is true if the value of y is positive.

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	value	type	description
crv	2	-1	int / tstr	EC Curve identifier - Taken from the COSE General 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. 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. [CREF15]

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 24 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:

- For all keys, 'kty' MUST be present and MUST have a value of 3.
- For public keys, the fields 'n' and 'e' MUST be present. All other fields defined in Table 24 MUST be absent.
- 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.
- 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 24: RSA Key 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 contains only private key information, care must be taken that it is never transmitted accidentally. For public keys, there are no required fields. For private keys, it is REQUIRED that 'k' be present in the structure.

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

Table 25: Symmetric Key Parameters

14. CBOR Encoder Restrictions

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

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

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

- All parsers used SHOULD fail on both parsing and generation if the same label is used twice as a key for the same map.

15. IANA Considerations

15.1. CBOR Tag assignment

It is requested that IANA assign a new tag from the "Concise Binary Object Representation (CBOR) Tags" registry. It is requested that the tag be assigned in the 0 to 23 value range.

Tag Value: TBD1

Data Item: COSE_Msg

Semantics: COSE security message.

15.2. COSE Header Parameter Registry

It is requested that IANA create a new registry entitled "COSE Header Parameters".

The columns of the registry are:

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

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

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

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

description This contains a brief description of the header field.

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

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

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

15.3. COSE Header Algorithm Label Table

It is requested that IANA create a new registry entitled "COSE Header Algorithm Labels".

The columns of the registry are:

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

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

label This is the value used for the label. The label is an integer in the range of -1 to -65536.

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

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

description This contains a brief description of the header field.

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

The initial contents of the registry can be found in: Table 12, Table 13, Table 19. The specification column for all rows in that table should be this document.

15.4. COSE Algorithm Registry

It is requested that IANA create a new registry entitled "COSE Algorithm Registry".

The columns of the registry are:

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

description A short description of the algorithm.

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

The initial contents of the registry can be found in the following: Table 9, Table 8, Table 10, Table 4, Table 5, Table 6, Table 7, Table 14, Table 15, Table 16, Table 17, Table 18. The specification column for all rows in that table should be this document.

15.5. COSE Key Common Parameter Registry

It is requested that IANA create a new registry entitled "COSE Key Common Parameter" Registry.

The columns of the registry are:

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

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

CBOR Type This field contains the CBOR type for the field

registry This field denotes the registry that values come from, if one exists.

description This field contains a brief description for the field

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

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

15.6. 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 22, Table 23, Table 24, and Table 25. The specification column for all of these entries will be this document.

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

15.8. Media Type Registration

15.8.1. COSE Security Message

This section registers the "application/cose" and "application/cose+cbor" media types in the "Media Types" registry. [CREF16] These media types are used to indicate that the content is a COSE_MSG.

Type name: application

Subtype name: cose

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

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

Interoperability considerations: N/A

Published specification: RFC TBD

Applications that use this media type: To be identified

Fragment identifier considerations: N/A

Additional information:

* Magic number(s): N/A

* File extension(s): cbor

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

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

Intended usage: COMMON

Restrictions on usage: N/A

Author: Jim Schaad, ietf@augustcellars.com

Change Controller: IESG

Provisional registration? No

Type name: application

Subtype name: cose+cbor

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

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

Interoperability considerations: N/A

Published specification: RFC TBD

Applications that use this media type: To be identified

Fragment identifier considerations: N/A

Additional information:

* Magic number(s): N/A

* File extension(s): cbor

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

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

Intended usage: COMMON

Restrictions on usage: N/A

Author: Jim Schaad, ietf@augustcellars.com

Change Controller: IESG

Provisional registration? No

15.8.2. COSE Key media type

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

Type name: application

Subtype name: cose-key+cbor

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

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

Interoperability considerations: N/A

Published specification: RFC TBD

Applications that use this media type: To be identified

Fragment identifier considerations: N/A

Additional information:

* Magic number(s): N/A

* File extension(s): cbor

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

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

Intended usage: COMMON

Restrictions on usage: N/A

Author: Jim Schaad, ietf@augustcellars.com

Change Controller: IESG

Provisional registration? No

Type name: application

Subtype name: cose-key-set+cbor

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

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

Interoperability considerations: N/A

Published specification: RFC TBD

Applications that use this media type: To be identified

Fragment identifier considerations: N/A

Additional information:

* Magic number(s): N/A

* File extension(s): cbor

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

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

Intended usage: COMMON

Restrictions on usage: N/A

Author: Jim Schaad, ietf@augustcellars.com

Change Controller: IESG

Provisional registration? No

16. 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.

17. References

17.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, October 2013.

17.2. 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., Birkholz, H., and R. Sun, "CBOR data definition language: a notational convention to express CBOR data structures.", draft-greevenbosch-appsawg-cbor-cddl-05 (work in progress), March 2015.

[I-D.irtf-cfrg-curves] Langley, A. and R. Salz, "Elliptic Curves for Security", draft-irtf-cfrg-curves-02 (work in progress), March 2015.

[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, February 1997.

[RFC2633] Ramsdell, B., "S/MIME Version 3 Message Specification", RFC 2633, June 1999.

[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, September 2002.

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.

[RFC3610] Whiting, D., Housley, R., and N. Ferguson, "Counter with CBC-MAC (CCM)", RFC 3610, September 2003.

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

[RFC4262] Santesson, S., "X.509 Certificate Extension for Secure/Multipurpose Internet Mail Extensions (S/MIME) Capabilities", RFC 4262, December 2005.

[RFC5480] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Elliptic Curve Cryptography Subject Public Key Information", RFC 5480, March 2009.

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

[RFC5751] Ramsdell, B. and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification", RFC 5751, January 2010.

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

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, May 2010.

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

[RFC6090] McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms", RFC 6090, February 2011.

[RFC6151] Turner, S. and L. Chen, "Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms", RFC 6151, March 2011.

[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., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.

[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, May 2015.

[RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, May 2015.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, May 2015.

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, May 2015.

[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. CDDL Grammar

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

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()
```

Appendix B. Three Levels of Recipient Information

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

These levels would be:

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

This is an example of what a triple layer message would look like. The message has the following layers:

- Level 0: Has a content encrypted with AES-GCM using a 128-bit key.

- Level 1: Uses the AES Key wrap algorithm with a 128-bit key.

- Level 2: Uses ECDH Ephemeral-Static direct to generate the level 1 key.

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

Size of binary file is 214 bytes

```
[
  2,
  h'a10101',
  {
    5: h'02d1f7e6f26c43d4868d87ce'
  },
  h'64f84d913ba60a76070a9a48f26e97e863e285295a44320878caceb0763a3
34806857c67',
  [
    [
      h'',
      {
        1: -3
      },
      h'5a15dbf5b282ecb31a6074ee3815c252405dd7583e078188',
      [
        [
          h'',
          {
            1: 50,
            4: h'6d65726961646f632e6272616e64796275636b406275636b
6c616e642e6578616d706c65',
            -1: {
              1: 2,
              -1: 1,
              -2: h'b2add44368ea6d641f9ca9af308b4079aeb519f11e9b8
a55a600b21233e86e68',
              -3: h'1a2cf118b9ee6895c8f415b686d4ca1cef362d4a7630a
31ef6019c0c56d33de0'
            }
          },
          h''
        ]
      ]
    ]
  ]
]
```

Appendix C. Examples

The examples can be found at https://github.com/cose-wg/Examples. The file names in each section correspond the the same file names in the repository. I am currently still in the process of getting the examples up there along with some control information for people to be able to check and reproduce the examples.

Examples may have some features that are in questions but not yet incorporated in the document.

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

```
gem install cbor-diag
```

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

```
diag2cbor < inputfile > outputfile
```

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

C.1. Examples of MAC messages

C.1.1. Shared Secret Direct MAC

This example users the following:

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

Size of binary file is 71 bytes

```
[
  3,
  h'a1016f4145532d434d41432d3235362f3634',
  {
  },
  h'5468697320697320746865520636f6e74656e742e',
  h'd9afa663dd740848',
  [
    [
      h'',
      {
        1: -6,
        4: h'6f75722d736563726574'
      },
      h''
    ]
  ]
]
```

C.1.2. ECDH Direct MAC

This example uses the following:

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

Size of binary file is 215 bytes

```
[
  3,
  h'a10104',
  {
  },
  h'5468697320697320746865520636f6e74656e742e',
  h'2ba937ca03d76c3dbad30cfcbaeef586f9c0f9ba616ad67e9205d38576ad9
930',
  [
    [
      h'',
      {
        1: 52,
        4: h'6d65726961646f632e6272616e64796275636b406275636b6c61
6e642e6578616d706c65',
        -3: h'706572656772696e2e746f6f6b407475636b626f726f7567682
e6578616d706c65',
        "apu": h'4d8553e7e74f3c6a3a9dd3ef286a8195cbf8a23d19558ccf
ec7d34b824f42d92bd06bd2c7f0271f0214e141fb779ae2856abf585a58368b01
7e7f2a9e5ce4db5'
      },
      h''
    ]
  ]
]
```

C.1.3. Wrapped MAC

This example uses the following:

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

Size of binary file is 122 bytes

```
[
  3,
  h'a1016e4145532d3132382d4d41432d3634',
  {
  },
  h'5468697320697320746865520636f6e74656e742e',
  h'6d1fa77b2dd9146a',
  [
    [
      h'',
      {
        1: -5,
        4: h'30313863306165352d346439622d343731622d626664362d6565
6633313462633730333337'
      },
      h'711ab0dc2fc4585dce27effa6781c8093eba906f227b6eb0'
    ]
  ]
]
```

C.1.4. Multi-recipient MAC message

This example uses the following:

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

Size of binary file is 670 bytes

```
[
  3,
  h'a10104',
  {
  },
  h'5468697320697320746865520636f6e74656e742e',
  h'7aaa6e74546873061f0a7de21ff0c0658d401a68da738dd893748651983ce
1d0',
  [
    [
      h'',
```

```
      {
        1: 55,
        4: h'62696c626f2e62616767696e7340686f626269746f6e2e657861
   6d706c65',
        -1: {
          1: 2,
          -1: 3,
          -2: h'43b12669acac3fd27898ffba0bcd2e6c366d53bc4db71f909
   a759304acfb5e18cdc7ba0b13ff8c7636271a6924b1ac63c02688075b55ef2d61
   3574e7dc242f79c3',
          -3: h'812dd694f4ef32b11014d74010a954689c6b6e8785b333d1a
   b44f22b9d1091ae8fc8ae40b687e5cfbe7ee6f8b47918a07bb04e9f5b1a51a334
   a16bc09777434113'
        }
      },
      h'f20ad9c96134f3c6be4f75e7101c0ecc5efa071ff20a87fd1ac285109
   41ee0376573e2b384b56b99'
    ],
    [
      h'',
      {
        1: -26,
        4: h'62696c626f2e62616767696e7340686f626269746f6e2e657861
   6d706c65'
      },
      h'46c4f88069b650909a891e84013614cd58a3668f88fa18f3852940a20
   b35098591d3aacf91c125a2595cda7bee75a490579f0e2f20fd6bc956623bfde3
   029c318f82c426dac3463b261c981ab18b72fe9409412e5c7f2d8f2b5abaf780d
   f6a282db033b3a863fa957408b81741878f466dcc437006ca21407181a016ca60
   8ca8208bd3c5a1ddc828531e30b89a67ec6bb97b0c3c3c92036c0cb84aa0f0ce8
   c3e4a215d173bfa668f116ca9f1177505afb7629a9b0b5e096e81d37900e06f56
   1a32b6bc993fc6d0cb5d4bb81b74e6ffb0958dac7227c2eb8856303d989f93b4a
   051830706a4c44e8314ec846022eab727e16ada628f12ee7978855550249ccb58
   '
    ],
    [
      h'',
      {
        1: -5,
        4: h'30313863306165352d346439622d343731622d626664362d6565
   66333134626337303337'
      },
      h'0b2c7cfce04e98276342d6476a7723c090dfdd15f9a518e7736549e99
   8370695e6d6a83b4ae507bb'
    ]
  ]
]
```

C.2. Examples of Encrypted Messages

C.2.1. Direct ECDH

This example uses the following:

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

Size of binary file is 182 bytes

```
[
  2,
  h'a10101',
  {
    5: h'c9cf4df2fe6c632bf7886413'
  },
  h'45fce2814311024d3a479e7d3eed063850f3f0b9f3f948677e3ae9869bcf9
ff4e1763812',
  [
    [
      h'',
      {
        1: 50,
        4: h'6d65726961646f632e6272616e64796275636b406275636b6c61
6e642e6578616d706c65',
        -1: {
          1: 2,
          -1: 1,
          -2: h'98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbfbf05
4e1c7b4d91d6280',
          -3: h'f01400b089867804b8e9fc96c3932161f1934f4223069170d
924b7e03bf822bb'
        }
      },
      h''
    ]
  ]
]
```

C.2.2. Direct plus Key Derivation

This example uses the following:

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

- Recipient class: Use HKDF on a shared secret with the following implicit fields as part of the context.
  - APU identity: "lighting-client"
  - APV identity: "lighting-server"
  - Supplimentary Public Other: "Encryption Example 02"

Size of binary file is 95 bytes

```
[
  2,
  h'a1010a',
  {
    5: h'89f52f65a1c580933b5261a7'
  },
  h'7b9dcfa42c4e1d3182c402dc18ef8b5637de4fb62cf1dd156ea6e6e0',
  [
    [
      h'',
      {
        1: "dir+kdf",
        4: h'6f75722d736563726574',
        -20: h'6161626263636464656566666767686 8'
      },
      h''
    ]
  ]
]
```

C.3. Examples of Signed Message

C.3.1. Single Signature

This example uses the following:

- Signature Algorithm: RSA-PSS w/ SHA-384, MGF-1

Size of binary file is 330 bytes

```
[
  1,
  h'',
  {
  },
  h'546869732069732074686520636f6e74656e742e',
  [
    [
      h'a2016550533383404581e62696c626f2e62616767696e7340686f626
269746f6e2e6578616d706c65',
      {
      },
      h'6d9d88a90ef4d6d7c0079fb11a33c855e2274c773f358df43b68f7873
eeda210692a61d70cd6a24ba0e3d82e359384be09faafea496bb0ed16f02091c4
8c02f33574edab5b3e334bae68d19580021327cc131fbee38eb0b28289dbce118
3f9067891b17fe752674b80437da02e9928ab7a155fef707b11d2bd38a71f224f
53170480116d96cc3f7266487b268679a13cdedffa93252a550371acc19971369
b58039056b308cc4e158bebe7c55db7874442d4321fd27f17dbb820ef19f43dcc
16cd50ccdd1b7dfd7cdde239a9245af41d949cdbbf1337ca254af20eeb167a62d
a5a51c83899c6f6e7c7e01dc3db21a250092a69fc635b74a2e54f5c98cb955d83
'
    ]
  ]
]
```

C.3.2. Multiple Signers

This example uses the following:

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

Size of binary file is 496 bytes

```
[
  1,
  h'',
  {
  },
  h'546869732069732074686520636f6e74656e742e',
  [
    [
      h'a1013819',
      {
        4: h'62696c626f2e62616767696e7340686f626269746f6e2e657861
6d706c65'
      },
      h'0ee972d931c7ab906e4bb71b80da0cc99c104fa53ebbf1f2cf7b668b9
3d766d3d2da28299f074675bb0db3cd0792ba83050c23c96795d58f9c7d68f66a
bbb8f35af8a0b5df369517b6db85e2cb62d852b666bc135c9022e46b538f78c26
adc2668963e74a019de718254385bb9cb137926ad6a88d1ff70043f85e555fb57
84107ce6e9de7c89c4fbadf8eca363a35f415f7a23523a8331b1aa2dfbac59a06
3e4357bde8e53fe34195d59bcda37d2c604804fffe60362e81476436aaa677129
f34b26639fc41b8e758e5edf273079c61b30130f0f83c57aa6856347e2556f718
eaf79a1fee1397a4f0b16b1b34db946eaaff10c793e5d1e681cb21c4fd20c5fdf
'
    ],
    [
      h'',
      {
        1: -9,
        4: h'62696c626f2e62616767696e7340686f626269746f6e2e657861
6d706c65'
      },
      h'0118eaa7d62778b5a9525a583f06b115d80cd246bc930f0c2850588ee
c85186b427026e096a076bfab738215f354be59f57643a7f6b2c92535cf3c37ee
2746a908ab1dcc673a63f327d9eff852b874f7a98b6638c7054fdeeaa3dce6542
4a21bd5dc728acedda7fcae6df6fc3298ff51ac911603a0f26d066935dccb85ea
eb0ae6d0e6'
    ]
  ]
]
```

C.4. COSE Keys

C.4.1. Public Keys

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

In order the keys are:

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

Size of binary file is 703 bytes

```
[
  {
    -1: 1,
    -2: h'65eda5a12577c2bae829437fe338701a10aaa375e1bb5b5de108de4
39c08551d',
    -3: h'1e52ed75701163f7f9e40ddf9f341b3dc9ba860af7e0ca7ca7e9eec
d0084d19c',
    1: 2,
    2: h'6d65726961646f632e6272616e64796275636b406275636b6c616e64
2e6578616d706c65'
  },
  {
    -1: 1,
    -2: h'98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbfbf054e1c7b
4d91d6280',
    -3: h'f01400b089867804b8e9fc96c3932161f1934f4223069170d924b7e
03bf822bb',
    1: 2,
    2: h'706572656772696e2e746f6f6b407475636b626f726f7567682e6578
616d706c65'
  },
  {
    -1: 3,
    -2: h'0072992cb3ac08ecf3e5c63dedec0d51a8c1f79ef2f82f94f3c737b
f5de7986671eac625fe8257bbd0394644caaa3aaf8f27a4585fbbcad0f2457620
085e5c8f42ad',
    -3: h'01dca6947bce88bc5790485ac97427342bc35f887d86d65a089377e
247e60baa55e4e8501e2ada5724ac51d6909008033ebc10ac999b9d7f5cc2519f
3fe1ea1d9475',
    1: 2,
    2: h'62696c626f2e62616767696e7340686f626269746f6e2e6578616d70
6c65'
  },
  {
    -2: h'9f810fb4038273d02591e4073f31d2b6001b82cedb4d92f050165d4
7cfcab8a3c41cb778ac7553793f8ef975768d1a2374d8712564c3bcd77b9ea434
544899407cff0099920a931a24c4414852ab29bdb0a95c0653f36c60e60bf90b6
258dda56f37047ba5c2d1d029af9c9d40bac7aa41c78a0dd1068add699e808fea
```

```
011ea1441d8a4f7bb4e97be39f55f1ddd44e9c4ba335159703d4d34b603e65147
a4f23d6d3c0996c75edee846a82d190ae10783c961cf0387aed2106d2d0555b6f
d937fad5535387e0ff72ffbe78941402b0b822ea2a74b6058c1dabf9b34a76cb6
3b87faa2c6847b8e2837fff91186e6b1c14911cf989a89092a81ce601ddacd3f9
cf',
    -1: h'010001',
    1: 3,
    2: h'62696c626f2e62616767696e7340686f626269746f6e2e6578616d70
6c65'
  }
]
```

C.4.2. Private Keys

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

In order the keys are:

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

Size of binary file is 1884 bytes

```
[
  {
    1: 2,
    2: h'6d65726961646f632e6272616e64796275636b406275636b6c616e64
2e6578616d706c65',
    -1: 1,
    -2: h'65eda5a12577c2bae829437fe338701a10aaa375e1bb5b5de108de4
39c08551d',
    -3: h'1e52ed75701163f7f9e40ddf9f341b3dc9ba860af7e0ca7ca7e9eec
d0084d19c',
    -4: h'aff907c99f9ad3aae6c4cdf21122bce2bd68b5283e6907154ad9118
40fa208cf'
  },
  {
```

```
      1: 4,
      2: h'6f75722d736563726574',
      -1: h'849b57219dae48de646d07dbb533566e976686457c1491be3a76dce
   a6c427188'
    },
    {
      1: 2,
      -1: 1,
      2: h'706572656772696e2e746f6f6b407475636b626f726f7567682e6578
   616d706c65',
      -2: h'98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbfbf054e1c7b
   4d91d6280',
      -3: h'f01400b089867804b8e9fc96c3932161f1934f4223069170d924b7e
   03bf822bb',
      -4: h'02d1f7e6f26c43d4868d87ceb2353161740aacf1f7163647984b522
   a848df1c3'
    },
    {
      1: 4,
      2: h'30313863306165352d346439622d343731622d626664362d65656633
   3134626337303337',
      -1: h'849b57219dae48de646d07dbb533566e976686457c1491be3a76dce
   a6c427188'
    },
    {
      1: 2,
      2: h'62696c626f2e62616767696e7340686f626269746f6e2e6578616d70
   6c65',
      -1: 3,
      -2: h'0072992cb3ac08ecf3e5c63dedec0d51a8c1f79ef2f82f94f3c737b
   f5de7986671eac625fe8257bbd0394644caaa3aaf8f27a4585fbbcad0f2457620
   085e5c8f42ad',
      -3: h'01dca6947bce88bc5790485ac97427342bc35f887d86d65a089377e
   247e60baa55e4e8501e2ada5724ac51d6909008033ebc10ac999b9d7f5cc2519f
   3fe1ea1d9475',
      -4: h'00085138ddabf5ca975f5860f91a08e91d6d5f9a76ad4018766a476
   680b55cd339e8ab6c72b5facdb2a2a50ac25bd086647dd3e2e6e99e84ca2c3609
   fdf177feb26d'
    },
    {
      1: 3,
      2: h'62696c626f2e62616767696e7340686f626269746f6e2e6578616d70
   6c65',
      -2: h'9f810fb4038273d02591e4073f31d2b6001b82cedb4d92f050165d4
   7cfcab8a3c41cb778ac7553793f8ef975768d1a2374d8712564c3bcd77b9ea434
   544899407cff0099920a931a24c4414852ab29bdb0a95c0653f36c60e60bf90b6
   258dda56f37047ba5c2d1d029af9c9d40bac7aa41c78a0dd1068add699e808fea
   011ea1441d8a4f7bb4e97be39f55f1ddd44e9c4ba335159703d4d34b603e65147
```

```
   a4f23d6d3c0996c75edee846a82d190ae10783c961cf0387aed2106d2d0555b6f
   d937fad5535387e0ff72ffbe78941402b0b822ea2a74b6058c1dabf9b34a76cb6
   3b87faa2c6847b8e2837fff91186e6b1c14911cf989a89092a81ce601ddacd3f9
   cf',
       -1: h'010001',
       -3: h'6d6502f41f84151228f24a467e1d19bb218fbcc34abd858db41fe29
   221fd936d1e4fe3b5abf23bf1e8999295f15d0d144c4b362ec3514bef2e25bbd0
   f80d62ae4c0c48c90ad49dd74c681dae10a4bbd81195d63bb0d03f00a64687e43
   aeb5ff8dab20d2d109ef16fa7677e2e8bfa8e7e42e72bd4160c3aa9688b00f9b3
   3059648316ed8c5016309074cc1332d81aa39ed389e8a9eab5844c414c704e05d
   90c5e2b85854ab5054ea5f83a84896c6a83cdac5edda1f8b3274f7d38e8039826
   8462a33ef9b525107c60ac8564c19cfe6e0e3775f242a1cafd3b9617d225dacf7
   4ce4f972976d61b057f82ff9870aea056aeee076c3df1cfc718d539c3a906b433
   c1',
       -4: h'dd297183f0f04d725c6fad3de51a17ca0402019e519c0bd9967a35c
   a11ed9d47b1fdfa7b019ffd9d168eec75fff9215f1907aeb5aa364c38c3016538
   56ea64f2bc3d251d00cd9d0dd9fbee2009abfd60ac986a5e36a4277afd53ec8c8
   4b2787c50cb7e9f909a7e1922933844b2b9a7747e8bc4eaef44996c3e9e99bfc6
   d4ab49',
       -5: h'b8a136761f9c4dfe84445e24e1efe3cbbf067cf61421a532a12489b
   81ce9dc2b9b937382aacea0ad3f1b47f72ed039b5319c169ad76a0f223de47ad4
   7aadcc3f5e6f30c38df251d3799bb69662afc2a5bb6a757953384cd6267bcf8c8
   c92e530156a01bf263cf7c117bd10fe85da91c47952a80675f76cc1de9545274b
   3ba457',
       -6: h'07c3d5bd792f26b8f62fe19843bbf7cbdafa2b0e60f526a15c1c2c5
   94ce9d7d4d596023e615f39ab53486f5af142d0fe22c5d7477f936a77afb913d1
   b7938139d88c190a7ca5bb76ea096361f294fc4f719fe4542c7cf4f9e77d13d81
   72ca0f85469e0a73f8f7d0feadbda64e71587a09a74d3d41fd47bc2862c515f9f
   5e8629',
       -7: h'08b0e60c676e87295cf68eebf38ac45159fba7343a3c5f3763e8816
   71e4d4fe4e99ce64a175a44ac031578acc5125e350e51c7aaa04b48cd16d6c385
   6f04f16166439bab08ea88398936f0406202de09c929b8bfee4fef260187c07c6
   03da5f63e7bcffb3c84903111b9ffabcb873f675d42abd02a0b6c9e2fa91d293d
   5c605f',
       -8: h'dcf8aabd740dd33c0c784fac06f6608b6f3d5cff57090177556a8fc
   cc2a7220429eff4ee828ebe35904a090b0c7f71da1060634d526cfe370af3e4d1
   5ef68a7beed931a423f157c175892cb1bbb434a0c386327e1ad8ac79a0d55aded
   d707d1c7f0c601541e9421ec5a02ae3149ea1e99129305eb19ae8ece2a3293f3f
   1a688e'
     }
   ]
```

Appendix D. Document Updates

D.1. Version -04 to -05

- Removed the jku, x5c, x5t, x5t#S256, x5u, and jwk headers.
- Add enveloped data vs encrypted data structures.
- Add counter signature parameter.

D.2. Version -03 to -04

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

D.3. Version -02 to -03

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

D.4. Version -02 to -03

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

D.5. Version -01 to -2

- Add first pass of algorithm information
- Add direct key derivation example.

D.6. Version -00 to -01

- Add note on where the document is being maintained and contributing notes.
- Put in proposal on MTI algorithms.
- Changed to use labels rather than keys when talking about what indexes a map.
- Moved nonce/IV to be a common header item.
- Expand section to discuss the common set of labels used in COSE_Key maps.
- Start marking element 0 in registries as reserved.
- Update examples.

Editorial Comments

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

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

[CREF3] JLS: It would be possible to extend this section to talk about those decisions which an application needs to think about rather than just talking about MTI algoithms.

[CREF4] Hannes: I would remove references to CMS and S/MIME since they are most likely only helpful to a very small audience.

[CREF5] JLS: I have moved msg_type into the individual structures. However, they would not be necessary in the cases where a) the security service is known and b) security libraries can setup to take individual structures. Should they be moved back to just appearing if used in a COSE_MSG rather than on the individual structure? This would make things shorter if one was using just a signed message because the msg_type field can be omitted as well as the COSE_Tagged_MSG tag field. One the other hand, it

will complicated the code if one is doing general purpose library type things.

[CREF6] JLS: Should we create an IANA registries for the values of msg_type?

[CREF7] CB: I would like to make msg_type go away

[CREF8] JLS: A completest version of this grammar would list the options available in the protected and unprotected headers. Do we want to head that direction?

[CREF9] JLS: Is there a reason to assign a CBOR tag to identify keys and/or key sets?

[CREF10] JLS: We can really simplify the grammar for COSE_Key to be just the kty (the one required field) and the generic item. The reason to do this is that it makes things simpler. The reason not to do this says that we really need to add a lot more items so that a grammar check can be done that is more tightly enforced.

[CREF11] Ilari: Look to see if we need to be clearer about how the fields defined in the table are transported and thus why they have labels.

[CREF12] Ilari: Check to see what the curves are renamed to during final publishing. It appears to be X25519 now.

[CREF13] JLS: Do we create a registry for curves? Is is the same registry for both EC1 and EC2?

[CREF14] JLS: Should we use the bignum encoding for x, y and d instead of bstr?

[CREF15] 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.

[CREF16] JLS: Should we register both or just the cose+cbor one?

Authors' Addresses

Jim Schaad
August Cellars

Email: ietf@augustcellars.com

Brian Campbell
Ping Identity

Email: brian.d.campbell@gmail.com