CoRE Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2017

G. Selander
J. Mattsson
F. Palombini
Ericsson AB
L. Seitz
SICS Swedish ICT
March 13, 2017

Object Security of CoAP (OSCOAP)
draft-ietf-core-object-security-02

Abstract

This document defines Object Security of CoAP (OSCOAP), a method for application layer protection of the Constrained Application Protocol (CoAP), using the CBOR Object Signing and Encryption (COSE). OSCOAP provides end-to-end encryption, integrity and replay protection to CoAP payload, options, and header fields, as well as a secure message binding. OSCOAP is designed for constrained nodes and networks and can be used across intermediaries and over any layer. The use of OSCOAP is signaled with the CoAP option Object-Security, also defined in this document.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

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

This document defines Object Security of CoAP (OSCOAP), a data object based security protocol, protecting CoAP message exchanges end-to-end, across intermediary nodes. An analysis of end-to-end security for CoAP messages through intermediary nodes is performed in [I-D.hartke-core-e2e-security-reqs], this specification addresses the forwarding case. In addition to the core features defined in [RFC7252], OSCOAP supports Observe [RFC7641] and Blockwise [RFC7959].

OSCOAP is designed for constrained nodes and networks and provides an in-layer security protocol for CoAP which does not depend on underlying layers. OSCOAP can be used anywhere that CoAP can be used, including unreliable transport [RFC7228], reliable transport [I-D.ietf-core-coap-tcp-tls], and non-IP transport [I-D.bormann-6lo-coap-802-15-ie]. OSCOAP may also be used to protect group communication for CoAP [I-D.tiloca-core-multicast-oscoap]. The use of OSCOAP does not affect the URI scheme and OSCOAP can therefore be used with any URI scheme defined for CoAP. The application decides the conditions for which OSCOAP is required.

OSCOAP builds on CBOR Object Signing and Encryption (COSE) [I-D.ietf-cose-msg], providing end-to-end encryption, integrity, replay protection, and secure message binding. The use of OSCOAP is

signaled with the CoAP option Object-Security, defined in Section 2. OSCOAP provides protection of CoAP payload, certain options, and header fields. The solution transforms an unprotected CoAP message into a protected CoAP message in the following way: the unprotected CoAP message is protected by including payload (if present), certain options, and header fields in a COSE object. The message fields that have been encrypted are removed from the message whereas the Object-Security option and the COSE object are added, see Figure 1.

```
      Client                                            Server
         |  request:                                       |
         |    GET example.com                              |
         |    [Header, Token, Options:{...,                |
         |     Object-Security:COSE object}]               |
         +------------------------------------------------>|
         |  response:                                      |
         |    2.05 (Content)                               |
         |    [Header, Token, Options:{...,                |
         |     Object-Security:-}, Payload:COSE object]    |
         |<------------------------------------------------+
         |                                                 |
```

Figure 1: Sketch of OSCOAP

OSCOAP may be used in extremely constrained settings, where CoAP over DTLS may be prohibitive e.g. due to large code size. Alternatively, OSCOAP can be combined with DTLS, thereby enabling end-to-end security of e.g. CoAP payload and options, in combination with hop-by-hop protection of the entire CoAP message, during transport between end-point and intermediary node. Examples of the use of OSCOAP are given in Appendix C.

The message protection provided by OSCOAP can alternatively be applied only to the payload of individual messages. We call this object security of content (OSCON) and it is defined in Appendix D.

1.1. Terminology

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

Readers are expected to be familiar with the terms and concepts described in CoAP [RFC7252], Observe [RFC7641], Blockwise [RFC7959], COSE [I-D.ietf-cose-msg], CBOR [RFC7049], CDDL

[I-D.greevenbosch-appsawg-cbor-cddl], and constrained environments [RFC7228].

2. The Object-Security Option

The Object-Security option (see Figure 2) indicates that OSCOAP is used to protect the CoAP message exchange. The Object-Security option is critical, safe to forward, part of the cache key, not repeatable, and opaque.

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

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

Figure 2: The Object-Security Option

A successful response to a request with the Object-Security option SHALL contain the Object-Security option. A CoAP endpoint SHOULD NOT cache a response to a request with an Object-Security option, since the response is only applicable to the original client's request. The Object-Security option is included in the cache key for backward compatibility with proxies not recognizing the Object-Security option. The effect is that messages with the Object-Security option will never generate cache hits. For Max-Age processing, see Section 4.3.1.1.

The protection is achieved by means of a COSE object (see Section 5) included in the protected CoAP message. The placement of the COSE object depends on whether the method/response code allows payload (see [RFC7252]):

o If the method/response code allows payload, then the compressed COSE object is the payload of the protected message, and the Object-Security option has length zero. An endpoint receiving a CoAP message with payload, that also contains a non-empty Object-Security option SHALL treat it as malformed and reject it.

o If the method/response code does not allow payload, then the compressed COSE object is the value of the Object-Security option and the length of the Object-Security option is equal to the size of the compressed COSE object. An endpoint receiving a CoAP message without payload, that also contains an empty Object-Security option SHALL treat it as malformed and reject it.

The size of the COSE object depends on whether the method/response code allows payload, if the message is a request or response, on the set of options that are included in the unprotected message, the AEAD algorithm, the length of the information identifying the security context, and the length of the sequence number.

3. The Security Context

OSCOAP uses COSE with an Authenticated Encryption with Additional Data (AEAD) algorithm between a CoAP client and a CoAP server. An implementation supporting this specification MAY only implement the client part or MAY only the server part.

The specification requires that client and server establish a security context to apply to the COSE objects protecting the CoAP messages. In this section we define the security context, and also specify how to derive the initial security contexts in client and server based on common shared secret and a key derivation function (KDF).

3.1. Security Context Definition

The security context is the set of information elements necessary to carry out the cryptographic operations in OSCOAP. For each endpoint, the security context is composed of a "Common Context", a "Sender Context", and a "Recipient Context".

The endpoints protect messages to send using the Sender Context and verify messages received using the Recipient Context, both contexts being derived from the Common Context and other data. Clients need to be able to retrieve the correct security context to use.

An endpoint uses its Sender ID (SID) to derive its Sender Context, and the other endpoint uses the same ID, now called Recipient ID (RID), to derive its Recipient Context. In communication between two endpoints, the Sender Context of one endpoint matches the Recipient Context of the other endpoint, and vice versa. Thus the two security contexts identified by the same IDs in the two endpoints are not the same, but they are partly mirrored. Retrieval and use of the security context are shown in Figure 3.

```
                  .------------.            .------------.
                  |  Common,   |            |  Common,   |
                  |  Sender,   |            |  Recipient,|
                  |  Recipient |            |  Sender    |
                  '------------'            '------------'
                      Client                    Server
                         |                        |
   Retrieve context for  | request:               |
    target resource      | [Token = Token1,       |
   Protect request with  |  kid = SID, ...]       |
     Sender Context      +----------------------->| Retrieve context with
                         |                        |  RID = SID
                         |                        | Verify request with
                         |                        |  Recipient Context
                         | response:              | Protect response with
                         | [Token = Token1, ...]  |  Sender Context
   Retrieve context with |<-----------------------+
    Token = Token1       |                        |
   Verify request with   |                        |
    Recipient Context    |                        |
```

Figure 3: Retrieval and use of the Security Context

The Common Context contains the following parameters:

- o Algorithm (Alg). Value that identifies the COSE AEAD algorithm to use for encryption. Its value is immutable once the security context is established.

- o Master Secret. Variable length, uniformly random byte string containing the key used to derive traffic keys and IVs. Its value is immutable once the security context is established.

- o Master Salt (OPTIONAL). Variable length byte string containing the salt used to derive traffic keys and IVs. Its value is immutable once the security context is established.

The Sender Context contains the following parameters:

- o Sender ID. Variable length byte string identifying the Sender Context. Its value is immutable once the security context is established.

- o Sender Key. Byte string containing the symmetric key to protect messages to send. Derived from Common Context and Sender ID. Length is determined by Algorithm. Its value is immutable once the security context is established.

- Sender IV. Byte string containing the IV to protect messages to send. Derived from Common Context and Sender ID. Length is determined by Algorithm. Its value is immutable once the security context is established.

- Sequence Number. Non-negative integer used to protect requests and observe responses to send. Used as partial IV [I-D.ietf-cose-msg] to generate unique nonces for the AEAD. Maximum value is determined by Algorithm.

The Recipient Context contains the following parameters:

- Recipient ID. Variable length byte string identifying the Recipient Context. Its value is immutable once the security context is established.

- Recipient Key. Byte string containing the symmetric key to verify messages received. Derived from Common Context and Recipient ID. Length is determined by the Algorithm. Its value is immutable once the security context is established.

- Recipient IV. Byte string containing the IV to verify messages received. Derived from Common Context and Recipient ID. Length is determined by Algorithm. Its value is immutable once the security context is established.

- Replay Window. The replay window to verify requests and observe responses received.

An endpoint may free up memory by not storing the Sender Key, Sender IV, Recipient Key, and Recipient IV, deriving them from the Common Context when needed. Alternatively, an endpoint may free up memory by not storing the Master Secret and Master Salt after the other parameters have been derived.

The endpoints MAY interchange the client and server roles while maintaining the same security context. When this happens, the former server still protects messages to send using its Sender Context, and verifies messages received using its Recipient Context. The same is also true for the former client. The endpoints MUST NOT change the Sender/Recipient ID. In other words, changing the roles does not change the set of keys to be used.

3.2. **Derivation of Security Context Parameters**

The parameters in the security context are derived from a small set of input parameters. The following input parameters SHALL be pre-established:

- Master Secret

- Sender ID

- Recipient ID

The following input parameters MAY be pre-established. In case any of these parameters is not pre-established, the default value indicated below is used:

- AEAD Algorithm (Alg)

   * Default is AES-CCM-64-64-128 (value 12)

- Master Salt

   * Default is the empty string

- Key Derivation Function (KDF)

   * Default is HKDF SHA-256

- Replay Window Type and Size

   * Default is DTLS-type replay protection with a window size of 32

How the input parameters are pre-established, is application specific. The EDHOC protocol [I-D.selander-ace-cose-ecdhe] enables the establishment of input parameters with the property of forward secrecy and negotiation of KDF and AEAD, it thus provides all necessary pre-requisite steps for using OSCOAP as defined here.

3.2.1. Derivation of Sender Key/IV, Recipient Key/IV

The KDF MUST be one of the HKDF [RFC5869] algorithms defined in COSE. HKDF SHA-256 is mandatory to implement. The security context parameters Sender Key/IV and Recipient Key/IV SHALL be derived from the input parameters using the HKDF, which consists of the composition of the HKDF-Extract and HKDF-Expand steps ([RFC5869]):

```
   output parameter = HKDF(salt, IKM, info, L)
```

where:

- salt is the Master Salt as defined above

- IKM is the Master Secret is defined above

o info is a CBOR array consisting of:

```
   info = [
       id : bstr,
       alg : int,
       type : tstr,
       L : int
   ]
```

   * id is the Sender ID or Recipient ID

   * type is "Key" or "IV"

o L is the key/IV size of the AEAD algorithm in octets without leading zeroes.

For example, if the algorithm AES-CCM-64-64-128 (see Section 10.2 in [I-D.ietf-cose-msg]) is used, the value for L is 16 for keys and 7 for IVs.

3.2.2. Initial Sequence Numbers and Replay Window

The Sequence Number is initialized to 0. The supported types of replay protection and replay window length is application specific and depends on the lower layers. Default is DTLS-type replay protection with a window size of 32 initiated as described in Section 4.1.2.6 of [RFC6347].

3.3. Requirements on the Security Context Parameters

As collisions may lead to the loss of both confidentiality and integrity, Sender ID SHALL be unique in the set of all security contexts using the same Master Secret. Normally (e.g. when using EDHOC) Sender IDs can be very short. Note that Sender IDs of different lengths can be used with the same Master Secret. E.g. the SID with value 0x00 is different from the SID with the value 0x0000. If Sender ID uniqueness cannot be guaranteed, random Sender IDs MUST be used. Random Sender IDs MUST be long enough so that the probability of collisions is negligible.

To enable retrieval of the right Recipient Context, the Recipient ID SHOULD be unique in the sets of all Recipient Contexts used by an endpoint.

The same Master Salt MAY be used with several Master Secrets.

4. Protected CoAP Message Fields

OSCOAP transforms an unprotected CoAP message into a protected CoAP message, and vice versa. This section defines how the CoAP message fields are protected. OSCOAP protects as much of the unprotected CoAP message as possible, while still allowing forward proxy operations [I-D.hartke-core-e2e-security-reqs]. Message fields may either be

o Class E: encrypted and integrity protected,

o Class I: integrity protected only, or

o Class U: unprotected.

This section also outlines how the message fields are transferred, a detailed description of the processing is provided in Section 7. Message fields of the unprotected CoAP message are either transferred in the header/options part of the protected CoAP message, or in the plaintext of the COSE object. Depending on which, the location of the message field in the protected CoAP message is called "inner" or "outer":

o Inner message field: message field included in the plaintext of the COSE object of the protected CoAP message (see Section 5.1)

o Outer message field: message field included in the header or options part of the protected CoAP message

The inner message fields are by definition encrypted and integrity protected by the COSE object (Class E). The outer message fields are not encrypted and thus visible to an intermediary, but may be integrity protected by including the message field values in the AAD of the COSE object (see Section 5.2). I.e. outer message fields may be Class I or Class U.

Note that, even though the message formats are slightly different, OSCOAP complies with CoAP over unreliable transport [RFC7252] as well as CoAP over reliable transport [I-D.ietf-core-coap-tcp-tls].

4.1. CoAP Payload

The CoAP Payload SHALL be encrypted and integrity protected (Class E), and thus is an inner message field.

The sending endpoint writes the payload of the unprotected CoAP message into the plaintext of the COSE object.

The receiving endpoint verifies and decrypts the COSE object, and recreates the payload of the unprotected CoAP message.

4.2. CoAP Header

Many CoAP header fields are required to be read and changed during a normal message exchange or when traversing a proxy and thus cannot be protected between the endpoints, e.g. CoAP message layer fields such as Message ID.

The CoAP header field Code MUST be sent in plaintext to support RESTful processing, but MUST be integrity protected to prevent an intermediary from changing, e.g. from GET to DELETE (Class I). The CoAP version number MUST be integrity protected to prevent potential future version-based attacks (Class I). Note that while the version number is not sent in each CoAP message over reliable transport [I-D.ietf-core-coap-tcp-tls], its value is known to client and server.

Other CoAP header fields SHALL neither be integrity protected nor encrypted (Class U). The CoAP header fields are thus outer message fields.

The sending endpoint SHALL copy the header fields from the unprotected CoAP message to the protected CoAP message. The receiving endpoint SHALL copy the header fields from the protected CoAP message to the unprotected CoAP message. Both sender and receiver insert the CoAP version number and header field Code in the AAD of the COSE object (see section Section 5.2).

4.3. CoAP Options

Most options are encrypted and integrity protected (Class E), and thus inner message fields. But to allow certain proxy operations, some options have outer values, i.e. are present in the protected CoAP message. Certain options may have both an inner value and a potentially different outer value, where the inner value is intended for the destination endpoint and the outer value is intended for the proxy.

A summary of how options are protected and processed is shown in Figure 4. Options within each class are protected and processed in a similar way, but certain options which require special processing as described in the subsections and indicated by a * in Figure 4.

No.	Name	E	I	U
1	If-Match	x		
3	Uri-Host			x
4	ETag	x		
5	If-None-Match	x		
6	Observe		*	
7	Uri-Port			x
8	Location-Path	x		
11	Uri-Path	x		
12	Content-Format	x		
14	Max-Age	*		
15	Uri-Query	x		
17	Accept	x		
20	Location-Query	x		
23	Block2	*		
27	Block1	*		
28	Size2	*		
35	Proxy-Uri			*
39	Proxy-Scheme			x
60	Size1	*		

E=Encrypt and Integrity Protect, I=Integrity Protect only,
U=Unprotected, *=Special

Figure 4: Protection of CoAP Options

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

Specifications of new CoAP options SHOULD define how they are processed with OSCOAP. New COAP options SHOULD be of class E and SHOULD NOT have outer options unless a forwarding proxy needs to read that option value. If a certain option is both inner and outer, the two values SHOULD NOT be the same, unless a proxy is required by specification to be able to read the end-to-end value.

4.3.1. Class E Options

For options in class E (see Figure 4) the option value in the unprotected CoAP message, if present, SHALL be encrypted and integrity protected between the endpoints. Hence the actions resulting from the use of such options is analogous to communicating in a protected manner with the endpoint. For example, a client using an ETag option will not be served by a proxy.

The sending endpoint SHALL write the class E option from the unprotected CoAP message into the plaintext of the COSE object.

Except for the special options described in the subsections, the sending endpoint SHALL NOT use the outer options of class E. However, note that an intermediary may, legitimately or not, add, change or remove the value of an outer option.

Except for the Block options Section 4.3.1.2, the receiving endpoint SHALL discard any outer options of class E from the protected CoAP message and SHALL replace it in the unprotected CoAP messages with the value from the COSE object when present.

4.3.1.1. Max-Age

An inner Max-Age option, like other class E options, is used as defined in [RFC7252] taking into account that it is not accessible to proxies.

Since OSCOAP binds CoAP responses to requests, a cached response would not be possible to use for any other request. To avoid unnecessary caching, a server MAY add an outer Max-Age option with value zero to protected CoAP responses (see Section 5.6.1 of [RFC7252]).

The outer Max-Age option is not integrity protected.

4.3.1.2. The Block Options

Blockwise [RFC7959] is an optional feature. An implementation MAY comply with [RFC7252] and the Object-Security option without implementing [RFC7959].

The Block options (Block1, Block2, Size1 and Size2) MAY be either only inner options, only outer options or both inner and outer options. The inner and outer options are processed independently.

The inner block options are used for endpoint-to-endpoint secure fragmentation of payload into blocks and protection of information about the fragmentation (block number, block size, last block). In this case, the CoAP client fragments the CoAP message as defined in [RFC7959] before the message is processed by OSCOAP. The CoAP server first processes the OSCOAP message before processing blockwise as defined in [RFC7959].

There SHALL be a security policy defining a maximum unfragmented message size for inner Block options such that messages exceeding this size SHALL be fragmented by the sending endpoint.

Additionally, a proxy may arbitrarily do block fragmentation on any CoAP message, in particular an OSCOAP message, as defined in [RFC7959] and thereby add outer Block options to a block and send on the next hop. The outer block options are thus neither encrypted nor integrity protected.

An endpoint receiving a message with an outer Block option SHALL first process this option according to [RFC7959], until all blocks of the protected CoAP message has been received, or the cumulated message size of the exceeds the maximum unfragmented message size. In the latter case the message SHALL be discarded. In the former case, the processing of the protected CoAP message continues as defined in this document.

If the unprotected CoAP message in turn contains Block options, the receiving endpoint processes this according to [RFC7959].

TODO: Update processing to support multiple concurrently proceeding requests

4.3.2. Class I Options

Except for the special options described in the subsections, for options in Class I (see Figure 4) the option value SHALL only be integrity protected between the endpoints. Options in Class I have outer values. Unless otherwise specified, the sending endpoint SHALL encode the Class I options in the protected CoAP message as described in Section 4.3.4.

Class I options are included in the external_aad (Section 5.2).

4.3.2.1. Observe

Observe [RFC7641] is an optional feature. An implementation MAY support [RFC7252] and the Object-Security option without supporting [RFC7641]. The Observe option as used here targets the requirements on forwarding of [I-D.hartke-core-e2e-security-reqs] (Section 2.2.1.2).

In order for a proxy to support forwarding of Observe, there MUST be an outer Observe option in the message.

o The Observe Registration (see Section 1.2 of [RFC7641]) of the unprotected CoAP request SHALL be encoded in the protected CoAP request as described in Section 4.3.4.

o The Observe Notification (see Section 1.2 of [RFC7641]) of the unprotected CoAP response SHALL be encoded in the protected CoAP response as described in Section 4.3.4.

To secure the Observe Registration and the order of the Notifications, Observe SHALL be integrity protected as described in this section:

o The Observe option in the unprotected CoAP request SHALL be included in the external_aad of the request (see Section 5.2).

o The Observe option SHALL be included in the external_aad of the response (see Section 5.2), with value set to the 3 least significant bytes of the Sequence Number of the response

4.3.3. Class U Options

Options in Class U have outer values and are used to support forward proxy operations. Unless otherwise specified, the sending endpoint SHALL encode the Class U options in the protected CoAP message as described in Section 4.3.4.

4.3.3.1. Uri-Host, Uri-Port, and Proxy-Scheme

The sending endpoint SHALL copy Uri-Host, Uri-Port, and Proxy-Scheme from the unprotected CoAP message to the protected CoAP message. When Uri-Host, Uri-Port, Proxy-Scheme options are present, Proxy-Uri is not used [RFC7252].

4.3.3.2. Proxy-Uri

Proxy-Uri, when present, is split by OSCOAP into class U options and privacy sensitive class E options, which are processed accordingly. When Proxy-Uri is used in the unprotected CoAP message, Uri-* are not present [RFC7252].

The sending endpoint SHALL first decompose the Proxy-Uri value of the unprotected CoAP message into the Proxy-Scheme, Uri-Host, Uri-Port, Uri-Path and Uri-Query options (if present) according to section 6.4 of [RFC7252].

Uri-Path and Uri-Query are class E options and MUST be protected and processed as if obtained from the unprotected CoAP message, see Section 4.3.1.

The value of the Proxy-Uri option of the protected CoAP message MUST be replaced with Proxy-Scheme, Uri-Host and Uri-Port options (if

present) composed according to section 6.5 of [RFC7252] and MUST be processed as a class U option, see Section 4.3.3.

An example of how Proxy-Uri is processed is given below.

An unprotected CoAP message contains:

o Proxy-Uri = "coap://example.com/resource?q=1"

During OSCOAP processing, Proxy-Uri is split into:

o Proxy-Scheme = "coap"

o Uri-Host = "example.com"

o Uri-Port = "5863"

o Uri-Path = "resource"

o Uri-Query = "q=1"

Uri-Path and Uri-Query follow the processing defined in Section 4.3.1. Proxy-Uri is added to the OSCOAP protected message with value:

o Proxy-Uri = "coap://example.com"

4.3.4. Outer Options in the Protected CoAP Message

All options with outer values present in the protected CoAP message, including the Object-Security option, SHALL be encoded as described in Section 3.1 of [RFC7252], where the delta is the difference to the previously included outer option.

5. The COSE Object

This section defines how to use COSE [I-D.ietf-cose-msg] to wrap and protect data in the unprotected CoAP message. OSCOAP uses the untagged COSE_Encrypt0 structure with an Authenticated Encryption with Additional Data (AEAD) algorithm. The key lengths, IV lengths, and maximum sequence number are algorithm dependent.

The AEAD algorithm AES-CCM-64-64-128 defined in Section 10.2 of [I-D.ietf-cose-msg] is mandatory to implement. For AES-CCM-64-64-128 the length of Sender Key and Recipient Key is 128 bits, the length of nonce, Sender IV, and Recipient IV is 7 bytes, and the maximum Sequence Number is 2^55 - 1.

The nonce is constructed as described in Section 3.1 of [I-D.ietf-cose-msg], i.e. by padding the partial IV (Sequence Number in network byte order) with zeroes and XORing it with the context IV (Sender IV or Recipient IV). The first bit in the Sender IV or Recipient IV SHALL be flipped in responses.

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

The COSE Object SHALL be a COSE_Encrypt0 object with fields defined as follows

- The "protected" field includes:

  * The "Partial IV" parameter. The value is set to the Sequence Number. The Partial IV SHALL be of minimum length needed to encode the sequence number. This parameter SHALL be present in requests, and MAY be present in responses. In case of Observe (Section 4.3.2.1}) the Partial IV SHALL be present in the response.

  * The "kid" parameter. The value is set to the Sender ID (see Section 3). This parameter SHALL be present in requests and SHALL NOT be present in responses.

- The "unprotected" field is empty.

- The "ciphertext" field is computed from the Plaintext (see Section 5.1) and the Additional Authenticated Data (AAD) (see Section 5.2) following Section 5.2 of [I-D.ietf-cose-msg].

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

5.1. Plaintext

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

- all Class E options Section 4.3.1 present in the unprotected CoAP message (see Section 4). The options are encoded as described in Section 3.1 of [RFC7252], where the delta is the difference to the previously included Class E option; and

- the Payload of unprotected CoAP message, if present, and in that case prefixed by the one-byte Payload Marker (0xFF).

```
      0                   1                   2                   3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |    Options to Encrypt (if any) ...
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |1 1 1 1 1 1 1 1|    Payload (if any) ...
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      (only if there
        is payload)
```

Figure 5: Plaintext

5.2. Additional Authenticated Data

The external_aad SHALL be a CBOR array as defined below:

```
   external_aad = [
      ver : uint,
      code : uint,
      options : bstr,
      alg : int,
      request_kid : bstr,
      request_seq : bstr
   ]
```

where:

- ver: contains the CoAP version number, as defined in Section 3 of [RFC7252].

- code: contains is the CoAP Code of the unprotected CoAP message, as defined in Section 3 of [RFC7252].

- options: contains the Class I options Section 4.3.2 present in the unprotected CoAP message encoded as described in Section 3.1 of [RFC7252], where the delta is the difference to the previously included class I option

- alg: contains the Algorithm from the security context used for the exchange (see Section 3.1).

- request_kid: contains the value of the 'kid' in the COSE object of the request (see Section 5).

- request_seq: contains the value of the 'Partial IV' in the COSE object of the request (see Section 5).

6. Sequence Numbers, Replay, Message Binding, and Freshness

6.1. AEAD Nonce Uniqueness

An AEAD nonce MUST NOT be used more than once per AEAD key. In order to assure unique nonces, each Sender Context contains a Sequence Number used to protect requests, and - in case of Observe - responses. The maximum sequence number is algorithm dependent and SHALL be 2^(min(nonce length in bits, 56) - 1) - 1. If the Sequence Number exceeds the maximum sequence number, the endpoint MUST NOT process any more messages with the given Sender Context. The endpoint SHOULD acquire a new security context (and consequently inform the other endpoint) before this happens. The latter is out of scope of this document.

6.2. Replay Protection

In order to protect from replay of messages, each Recipient Context contains a Replay Window used to verify request, and - in case of Observe - responses. A receiving endpoint SHALL verify that a Sequence Number (Partial IV) received in the COSE object has not been received before in the Recipient Context. The size and type of the Replay Window depends on the use case and lower protocol layers. In case of reliable and ordered transport from endpoint to endpoint, the recipient MAY just store the last received sequence number and require that newly received Sequence Numbers equals the last received Sequence Number + 1.

6.3. Sequence Number and Replay Window State

6.3.1. The Basic Case

To prevent reuse of the Nonce/Sequence Number with the same key, or from accepting replayed messages, a node needs to handle the situation of suddenly losing sequence number and replay window state in RAM, e.g. as a result of a reboot.

After boot, a node MAY reject to use existing security contexts from before it booted and MAY establish a new security context with each party it communicates, e.g. using EDHOC [I-D.selander-ace-cose-ecdhe]. However, establishing a fresh security context may have a non-negligible cost in terms of e.g. power consumption.

If a stored security context is to be used after reboot, then the node MUST NOT reuse a previous Sequence Number and MUST NOT accept previously accepted messages. The node MAY perform the following procedure:

- o Before sending a message, the client stores in persistent memory a sequence number associated to the stored security context higher than any sequence number which has been or are being sent using this security context. After boot, the client does not use any lower sequence number in a request than what was persistently stored with that security context.

  * Storing to persistent memory can be costly. Instead of storing a sequence number for each request, the client may store Seq + K to persistent memory every K requests, where Seq is the current sequence number and K > 1. This is a trade-off between the number of storage operations and efficient use of sequence numbers.

- o After boot, before accepting a message from a stored security context, the server synchronizes the replay window so that no old messages are being accepted. The server uses the Repeat option [I-D.mattsson-core-coap-actuators] for synchronizing the replay window: For each stored security context, the first time after boot the server receives an OSCOAP request, it generates a pseudo-random nonce and responds with the Repeat option set to the nonce as described in [I-D.mattsson-core-coap-actuators]. If the server receives a repeated OSCOAP request containing the Repeat option and the same nonce, and if the server can verify the request, then the sequence number obtained in the repeated message is set as the lower limit of the replay window.

6.4. Freshness

For responses without Observe, OSCOAP provides absolute freshness. For requests, and responses with Observe, OSCOAP provides relative freshness in the sense that the sequence numbers allows a recipient to determine the relative order of messages. For applications having stronger demands on freshness (e.g. control of actuators), OSCOAP needs to be augmented with mechanisms providing absolute freshness [I-D.mattsson-core-coap-actuators].

6.5. Delay and Mismatch Attacks

In order to prevent response delay and mismatch attacks [I-D.mattsson-core-coap-actuators] from on-path attackers and compromised proxies, OSCOAP binds responses to the request by including the request's ID (Sender ID or Recipient ID) and sequence number in the AAD of the response. The server therefore needs to store the request's ID (Sender ID or Recipient ID) and sequence number until all responses have been sent.

7. Processing

7.1. Protecting the Request

Given an unprotected request, the client SHALL perform the following steps to create a protected request:

1. Retrieve the Sender Context associated with the target resource.

2. Compose the Additional Authenticated Data, as described in Section 5.

3. Compose the AEAD nonce by XORing the context IV (Sender IV) with the partial IV (Sequence Number in network byte order). Increment the Sequence Number by one.

4. Encrypt the COSE object using the Sender Key. Compress the COSE Object as specified in Appendix A.

5. Format the protected CoAP message according to Section 4. The Object-Security option is added, see Section 4.3.4.

6. Store the association Token - Security Context. The client SHALL be able to find the Recipient Context from the Token in the response.

7.2. Verifying the Request

A server receiving a request containing the Object-Security option SHALL perform the following steps:

1. Process outer Block options according to [RFC7959], until all blocks of the request have been received, see Section 4.3.1.2.

2. Retrieve the Recipient Context associated with the Recipient ID in the 'kid' parameter of the COSE object.

3. Verify the Sequence Number in the 'Partial IV' parameter, as described in Section 6.

4. Compose the Additional Authenticated Data, as described in Section 5.

5. Compose the AEAD nonce by XORing the context IV (Recipient IV) with the padded 'Partial IV' parameter, received in the COSE Object.

6. Decrypt the COSE object using the Recipient Key.

   * If decryption fails, the server MUST stop processing the request and SHOULD send an 4.01 error message.

   * If decryption succeeds, update the Recipient Replay Window, as described in Section 6.

7. Add decrypted options or payload to the unprotected request, overwriting any outer E options (see Section 4). The Object-Security option is removed.

8. The unprotected CoAP request is processed according to [RFC7252]

7.3. Protecting the Response

Given an unprotected response, the server SHALL perform the following steps to create a protected response:

1. Retrieve the Sender Context in the Security Context used to verify the request.

2. Compose the Additional Authenticated Data, as described in Section 5.

3. Compose the AEAD nonce

   * If Observe is not used, compose the AEAD nonce by XORing the context IV (Recipient IV with the first bit flipped) with the padded Partial IV parameter from the request.

   * If Observe is used, compose the AEAD nonce by XORing the context IV (Recipient IV with the first bit flipped) with the partial IV (Sequence Number in network byte order). Increment the Sequence Number by one.

4. Encrypt the COSE object using the Sender Key. Compress the COSE Object as specified in Appendix A.

5. Format the protected CoAP message according to Section 4. The Object-Security option is added, see Section 4.3.4.

7.4. Verifying the Response

A client receiving a response containing the Object-Security option SHALL perform the following steps:

1. Process outer Block options according to [RFC7959], until all blocks of the protected CoAP message have been received, see Section 4.3.1.2.

2. Retrieve the Recipient Context associated with the Token.

3. If Observe is used, verify the Sequence Number in the 'Partial IV' parameter as described in Section 6.

4. Compose the Additional Authenticated Data, as described in Section 5.

5. Compose the AEAD nonce

   * If Observe is not used, compose the AEAD nonce by XORing the context IV (Recipient IV with the first bit flipped) with the padded Partial IV parameter from the request.

   * If Observe is used, compose the AEAD nonce by XORing the context IV (Recipient IV with the first bit flipped) with the padded Partial IV parameter from the response.

6. Decrypt the COSE object using the Recipient Key.

   * If decryption fails, the client MUST stop processing the response and SHOULD send an 4.01 error message.

   * If decryption succeeds and Observe is used, update the Recipient Replay Window, as described in Section 6.

7. Add decrypted options or payload to the unprotected response overwriting any outer E options (see Section 4). The Object-Security option is removed.

   * If Observe is used, replace the Observe value with the 3 least significant bytes in the sequence number.

8. The unprotected CoAP response is processed according to [RFC7252]

8. Web Linking

The use of OSCOAP MAY be indicated by a target attribute "osc" in a web link [RFC5988] to a CoAP resource. This attribute is a hint indicating that the destination of that link is to be accessed using OSCOAP. Note that this is simply a hint, it does not include any security context material or any other information required to run OSCOAP.

A value MUST NOT be given for the "osc" attribute; any present value MUST be ignored by parsers. The "osc" attribute MUST NOT appear more than once in a given link-value; occurrences after the first MUST be ignored by parsers.

9. Security Considerations

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

DTLS protects hop-by-hop the entire CoAP message, including header, options, and payload. OSCOAP protects end-to-end the payload, and all information in the options and header, that is not required for forwarding (see Section 4). DTLS and OSCOAP can be combined, thereby enabling end-to-end security of CoAP payload, in combination with hop-by-hop protection of the entire CoAP message, during transport between end-point and intermediary node.

The CoAP message layer, however, cannot be protected end-to-end through intermediary devices since the parameters Type and Message ID, as well as Token and Token Length may be changed by a proxy. Moreover, messages that are not possible to verify should for security reasons not always be acknowledged but in some cases be silently dropped. This would not comply with CoAP message layer, but does not have an impact on the application layer security solution, since message layer is excluded from that.

The use of COSE to protect CoAP messages as specified in this document requires an established security context. The method to establish the security context described in Section 3.2 is based on a common shared secret material in client and server, which may be obtained e.g. by using EDHOC [I-D.selander-ace-cose-ecdhe] or the ACE framework [I-D.ietf-ace-oauth-authz]. An OSCOAP profile of ACE is described in [I-D.seitz-ace-oscoap-profile].

The formula 2^(min(nonce length in bits, 56) - 1) - 1 (Section 6.1) guarantees unique nonces during the required use the algorithm, considering the same partial IV and flipped first bit of IV (Section 5) is used in request and response (which is the reason for -1 in the exponent). The compression algorithm (Appendix A) assumes that the partial IV is 56 bits or less (which is the reason for min(,) in the exponent).

The mandatory-to-implement AEAD algorithm AES-CCM-64-64-128 is selected for broad applicability in terms of message size (2^64 blocks) and maximum number of messages (2^56). Compatibility with CCM* is achieved by using the algorithm AES-CCM-16-64-128 [I-D.ietf-cose-msg].

Most AEAD algorithms require a unique nonce for each message, for which the sequence numbers in the COSE message field "Partial IV" is used. If the recipient accepts any sequence number larger than the one previously received, then the problem of sequence number synchronization is avoided. With reliable transport it may be defined that only messages with sequence number which are equal to previous sequence number + 1 are accepted. The alternatives to sequence numbers have their issues: very constrained devices may not be able to support accurate time, or to generate and store large numbers of random nonces. The requirement to change key at counter wrap is a complication, but it also forces the user of this specification to think about implementing key renewal.

The inner block options enable the sender to split large messages into protected blocks such that the receiving node can verify blocks before having received the complete message. The outer block options allow for arbitrary proxy fragmentation operations that cannot be verified by the endpoints, but can by policy be restricted in size since the encrypted options allow for secure fragmentation of very large messages. A maximum message size (above which the sending endpoint fragments the message and the receiving endpoint discards the message, if complying to the policy) may be obtained as part of normal resource discovery.

Applications need to use a padding scheme if the content of a message can be determined solely from the length of the payload. As an example, the strings "YES" and "NO" even if encrypted can be distinguished from each other as there is no padding supplied by the current set of encryption algorithms. Some information can be determined even from looking at boundary conditions. An example of this would be returning an integer between 0 and 100 where lengths of 1, 2 and 3 will provide information about where in the range things are. Three different methods to deal with this are: 1) ensure that all messages are the same length. For example using 0 and 1 instead of 'yes' and 'no'. 2) Use a character which is not part of the responses to pad to a fixed length. For example, pad with a space to three characters. 3) Use the PKCS #7 style padding scheme where m bytes are appended each having the value of m. For example, appending a 0 to "YES" and two 1's to "NO". This style of padding means that all values need to be padded.

10. Privacy Considerations

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

The unprotected options (Figure 4) may reveal privacy sensitive information. In particular Uri-Host SHOULD NOT contain privacy sensitive information.

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

11. IANA Considerations

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

11.1. CoAP Option Numbers Registry

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

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

11.2. Media Type Registrations

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

Type name: application

Subtype name: oscon

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: See Appendix C of this document.

Interoperability considerations: N/A

Published specification: [[this document]] (this document)

Applications that use this media type: To be identified

Fragment identifier considerations: N/A

Additional information:

* Magic number(s): N/A

* File extension(s): N/A

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

Person & email address to contact for further information:
   Goeran Selander <goran.selander@ericsson.com>

Intended usage: COMMON

Restrictions on usage: N/A

Author: Goeran Selander, goran.selander@ericsson.com

11.3. CoAP Content Format Registration

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

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

12. Acknowledgments

The following individuals provided input to this document: Christian Amsuess, Carsten Bormann, Joakim Brorsson, Martin Gunnarsson, Klaus Hartke, Jim Schaad, Marco Tiloca, and Malisa Vu&#269;ini&#263;.

Ludwig Seitz and Goeran Selander worked on this document as part of the CelticPlus project CyberWI, with funding from Vinnova.

13. References

13.1. Normative References

[I-D.ietf-cose-msg]
Schaad, J., "CBOR Object Signing and Encryption (COSE)", draft-ietf-cose-msg-24 (work in progress), November 2016.

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

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

[RFC5988] Nottingham, M., "Web Linking", RFC 5988, DOI 10.17487/RFC5988, October 2010, <http://www.rfc-editor.org/info/rfc5988>.

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

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

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

[RFC7641] Hartke, K., "Observing Resources in the Constrained Application Protocol (CoAP)", RFC 7641, DOI 10.17487/RFC7641, September 2015, <http://www.rfc-editor.org/info/rfc7641>.

[RFC7959] Bormann, C. and Z. Shelby, Ed., "Block-Wise Transfers in the Constrained Application Protocol (CoAP)", RFC 7959, DOI 10.17487/RFC7959, August 2016, <http://www.rfc-editor.org/info/rfc7959>.

13.2. Informative References

[I-D.bormann-6lo-coap-802-15-ie]
Bormann, C., "Constrained Application Protocol (CoAP) over IEEE 802.15.4 Information Element for IETF", draft-bormann-6lo-coap-802-15-ie-00 (work in progress), April 2016.

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

[I-D.hartke-core-e2e-security-reqs]
Selander, G., Palombini, F., and K. Hartke, "Requirements for CoAP End-To-End Security", draft-hartke-core-e2e-security-reqs-02 (work in progress), January 2017.

[I-D.ietf-ace-oauth-authz]
Seitz, L., Selander, G., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "Authentication and Authorization for Constrained Environments (ACE)", draft-ietf-ace-oauth-authz-05 (work in progress), February 2017.

[I-D.ietf-core-coap-tcp-tls]
Bormann, C., Lemay, S., Tschofenig, H., Hartke, K., Silverajan, B., and B. Raymor, "CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets", draft-ietf-core-coap-tcp-tls-07 (work in progress), March 2017.

[I-D.mattsson-core-coap-actuators]
Mattsson, J., Fornehed, J., Selander, G., and F. Palombini, "Controlling Actuators with CoAP", draft-mattsson-core-coap-actuators-02 (work in progress), November 2016.

[I-D.seitz-ace-oscoap-profile]
Seitz, L. and F. Palombini, "OSCOAP profile of ACE", draft-seitz-ace-oscoap-profile-01 (work in progress), October 2016.

[I-D.selander-ace-cose-ecdhe]
Selander, G., Mattsson, J., and F. Palombini, "Ephemeral Diffie-Hellman Over COSE (EDHOC)", draft-selander-ace-cose-ecdhe-04 (work in progress), October 2016.

[I-D.tiloca-core-multicast-oscoap]
Tiloca, M., Selander, G., and F. Palombini, "Secure group communication for CoAP", draft-tiloca-core-multicast-oscoap-00 (work in progress), October 2016.

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

Appendix A. OSCOAP Compression

The Concise Binary Object Representation (CBOR) combines very small message sizes with extensibility. CBOR Object Signing and Encryption (COSE) uses CBOR to achieve smaller message sizes than JOSE. COSE is however constructed to support a large number of different stateless use cases, and is not fully optimized for use as a stateful security protocol, leading to a larger than necessary message expansion. In this section we define a simple stateless compression mechanism for OSCOAP, which significantly reduces the per-packet overhead.

The value of the Object-Security option SHALL in general be encoded as:

```
[
  Partial IV,
  ? kid,
  ciphertext
]
```

Furthermore, the type and length for the ciphertext is redundant and 10 bits in the first two bytes are static. The type and length for the ciphertext SHALL be excluded, and the first sixteen bits in the above COSE array SHALL be encoded as a single byte:

```
10000abc 01000def -> 00abcdef
```

The exception is Responses without Observe that SHALL be encoded as:

```
ciphertext
```

A.1. Examples

A.1.1. Example Request

```
   COSE Object Before Compression (24 bytes)
   83 a2 04 41 25 06 41 05 a0 4e ae a0 15 56 67 92
   4d ff 8a 24 e4 cb 35 b9

   [
   {
     4:h'25',
     6:h'05'
   },
   {},
   h'aea0155667924dff8a24e4cb35b9'
   ]

   After Compression (18 bytes)
   19 05 41 25 ae a0 15 56 67 92 4d ff 8a 24 e4 cb
   35 b9
```

A.1.2. Example Response

```
   COSE Object Before Compression (18 bytes)
   83 a0 a0 4e ae a0 15 56 67 92 4d ff 8a 24 e4 cb
   35 b9

   [
   {},
   {},
   h'aea0155667924dff8a24e4cb35b9'
   ]

   After Compression (14 bytes)
   ae a0 15 56 67 92 4d ff 8a 24 e4 cb 35 b9
```

A.1.3. Example Response (with Observe)

```
COSE Object Before Compression (21 bytes)
83 a1 06 41 07 a0 4e ae a0 15 56 67 92 4d ff 8a
24 e4 cb 35 b9

[
{
  6:h'07'
},
{},
h'aea0155667924dff8a24e4cb35b9'
]

After Compression (16 bytes)
11 07 ae a0 15 56 67 92 4d ff 8a 24 e4 cb 35 b9
```

Appendix B. Test Vectors

TODO: This section needs to be updated.

Appendix C. Examples

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

C.1. Secure Access to Sensor

This example targets the scenario in Section 3.1 of [I-D.hartke-core-e2e-security-reqs] and illustrates a client requesting the alarm status from a server.

```
      Client  Proxy  Server
         |      |      |
         +----->|      |            Code: 0.01 (GET)
         | GET  |      |           Token: 0x8c
         |      |      | Object-Security: [kid:5f, seq:42,
         |      |      |                   {Uri-Path:"alarm_status"}]
         |      |      |         Payload: -
         |      |      |
         |      +----->|            Code: 0.01 (GET)
         |      | GET  |           Token: 0x7b
         |      |      | Object-Security: [kid:5f, seq:42,
         |      |      |                   {Uri-Path:"alarm_status"}]
         |      |      |         Payload: -
         |      |      |
         |      |<-----+            Code: 2.05 (Content)
         |      | 2.05 |           Token: 0x7b
         |      |      | Object-Security: -
         |      |      |         Payload: [{"OFF"}]
         |      |      |
         |<-----+      |            Code: 2.05 (Content)
         | 2.05 |      |           Token: 0x8c
         |      |      | Object-Security: -
         |      |      |         Payload: [{"OFF"}]
         |      |      |
```

Figure 6: Secure Access to Sensor. Square brackets [...] indicate a COSE object. Curly brackets { ... } indicate encrypted data.

Since the method (GET) doesn't allow payload, the Object-Security option carries the COSE object as its value. Since the response code (Content) allows payload, the COSE object is carried as the CoAP payload.

The COSE header of the request contains an identifier (5f), indicating which security context was used to protect the message and a sequence number (42). The option Uri-Path ("alarm_status") and payload ("OFF") are encrypted.

The server verifies that the sequence number has not been received before. The client verifies that the response is bound to the request.

C.2. Secure Subscribe to Sensor

This example targets the scenario in Section 3.2 of [I-D.hartke-core-e2e-security-reqs] and illustrates a client requesting subscription to a blood sugar measurement resource (GET

/glucose), first receiving the value 220 mg/dl and then a second
value 180 mg/dl.

```
 Client  Proxy  Server
    |      |      |
    +----->|      |            Code: 0.01 (GET)
    | GET  |      |           Token: 0x83
    |      |      |         Observe: 0
    |      |      | Object-Security: [kid:ca, seq:15,
    |      |      |                   {Uri-Path:"glucose"}]
    |      |      |         Payload: -
    |      |      |
    |      +----->|            Code: 0.01 (GET)
    |      | GET  |           Token: 0xbe
    |      |      |         Observe: 0
    |      |      | Object-Security: [kid:ca, seq:15,
    |      |      |                   {Uri-Path:"glucose"}]
    |      |      |         Payload: -
    |      |      |
    |      |<-----+            Code: 2.05 (Content)
    |      | 2.05 |           Token: 0xbe
    |      |      |         Observe: 000032
    |      |      | Object-Security: -
    |      |      |         Payload: [seq:32, {Content-Format:0, "220"}]
    |      |      |
    |<-----+      |            Code: 2.05 (Content)
    | 2.05 |      |           Token: 0x83
    |      |      |         Observe: 000032
    |      |      | Object-Security: -
    |      |      |         Payload: [seq:32, {Content-Format:0, "220"}]
   ...    ...    ...
    |      |      |
    |      |<-----+            Code: 2.05 (Content)
    |      | 2.05 |           Token: 0xbe
    |      |      |         Observe: 000036
    |      |      | Object-Security: -
    |      |      |         Payload: [seq:36, {Content-Format:0, "180"}]
    |      |      |
    |<-----+      |            Code: 2.05 (Content)
    | 2.05 |      |           Token: 0x83
    |      |      |         Observe: 000036
    |      |      | Object-Security: -
    |      |      |         Payload: [seq:36, {Content-Format:0, "180"}]
    |      |      |
```

Figure 7: Secure Subscribe to Sensor. Square brackets [...] indicate a COSE object. Curly brackets { ... } indicate encrypted data.

Since the method (GET) doesn't allow payload, the Object-Security option carries the COSE object as its value. Since the response code (Content) allows payload, the COSE object is carried as the CoAP payload.

The COSE header of the request contains an identifier (ca), indicating the security context used to protect the message and a Sequence Number (15). The COSE header of the responses contains sequence numbers (32 and 36). The options Content-Format (0) and the payload ("220" and "180"), are encrypted. The Observe option is integrity protected. The shown Observe values (000032 and 000036) are the ones that the client will see after OSCOAP processing.

The server verifies that the sequence number has not been received before. The client verifies that the sequence number has not been received before and that the responses are bound to the request.

Appendix D. Object Security of Content (OSCON)

TODO: This section needs to be updated.

OSCOAP protects message exchanges end-to-end between a certain client and a certain server, targeting the security requirements for forward proxy of [I-D.hartke-core-e2e-security-reqs]. In contrast, many use cases require one and the same message to be protected for, and verified by, multiple endpoints, see caching proxy section of [I-D.hartke-core-e2e-security-reqs]. Those security requirements can be addressed by protecting essentially the payload/content of individual messages using the COSE format ([I-D.ietf-cose-msg]), rather than the entire request/response message exchange. This is referred to as Object Security of Content (OSCON).

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

The unprotected payload shall be the plaintext/payload of the COSE object. The 'protected' field of the COSE object 'Headers' shall include the context identifier, both for requests and responses. If the unprotected CoAP message includes a Content-Format option, then the COSE object shall include a protected 'content type' field, whose value is set to the unprotected message Content-Format value. The Content-Format option of the protected CoAP message shall be replaced with "application/oscon" (Section 11)

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

Most AEAD algorithms require a unique nonce for each message. Sequence numbers for partial IV as specified for OSCOAP may be used for replay protection as described in Section 6. The use of time stamps in the COSE header parameter 'operation time' [I-D.ietf-cose-msg] for freshness may be used.

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

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

D.1. Overhead OSCON

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

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

The sizes of COSE messages for selected algorithms are detailed in this section.

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

o Cid: 0xa1534e3c

o Seq: 0xa3

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

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

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

D.2. MAC Only

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

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

The object in COSE encoding gives:

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

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

Figure 8 summarizes these results.

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

Figure 8: Message overhead for a 5-byte Tid using HMAC 256/64

D.3. Signature Only

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

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

The object in COSE encoding gives:

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

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

Figure 9 summarizes these results.

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

Figure 9: Message overhead for a 5-byte Tid using 64 byte ECDSA signature.

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

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

Since the key is implicitly known by the recipient, the COSE_Encrypt0_Tagged structure is used (Section 5.2 of [I-D.ietf-cose-msg]).

The object in COSE encoding gives:

```
  993(                         # COSE_Encrypt0_Tagged
    [
      h'a20444a1534e3c0641a3', # protected:
                                 {04:h'a1534e3c',
                                  06:h'a3'}
      {},                      # unprotected
      TAG                      # ciphertext + truncated 8-byte TAG
    ]
  )
```

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

Figure 10 summarizes these results.

Structure	Tid	TAG	COSE OH	Message OH
COSE_Encrypt0_Tagged	5 B	8 B	12 B	25 bytes

Figure 10: Message overhead for a 5-byte Tid using AES_128_CCM_8.

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

This example is based on AES-CCM and ECDSA with 64 bytes signature. The same assumption on the security context as in Appendix D.4. COSE defines the field 'counter signature w/o headers' that is used here to sign a COSE_Encrypt0_Tagged message (see Section 3 of [I-D.ietf-cose-msg]).

The object in COSE encoding gives:

```
993(                         # COSE_Encrypt0_Tagged
  [
    h'a20444a1534e3c0641a3', # protected:
                               {04:h'a1534e3c',
                                06:h'a3'}
    {9:SIG},                 # unprotected:
                                09: 64 bytes signature
    TAG                      # ciphertext + truncated 8-byte TAG
  ]
)
```

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

Figure 11 summarizes these results.

Structure	Tid	TAG	SIG	COSE OH	Message OH
COSE_Encrypt0_Tagged	5 B	8 B	64 B	15 B	92 B

Figure 11: Message overhead for a 5-byte Tid using AES-CCM countersigned with ECDSA.

Authors' Addresses

Goeran Selander
Ericsson AB

Email: goran.selander@ericsson.com

John Mattsson
Ericsson AB

Email: john.mattsson@ericsson.com

Francesca Palombini
Ericsson AB

Email: francesca.palombini@ericsson.com

Ludwig Seitz
SICS Swedish ICT

Email: ludwig@sics.se