CoRE Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 2, 2018

G. Selander
J. Mattsson
F. Palombini
Ericsson AB
L. Seitz
SICS Swedish ICT

September 29, 2017

Object Security for Constrained RESTful Environments (OSCORE)
draft-ietf-core-object-security-05

Abstract

This document defines Object Security for Constrained RESTful Environments (OSCORE), a method for application-layer protection of the Constrained Application Protocol (CoAP), using CBOR Object Signing and Encryption (COSE). OSCORE provides end-to-end encryption, integrity and replay protection, as well as a secure message binding. OSCORE is designed for constrained nodes and networks and can be used over any layer and across intermediaries, and also with HTTP. OSCORE may be used to protect group communications as is specified in a separate draft.

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, and a mapping to HTTP is also specified [RFC8075]. CoAP [RFC7252] references DTLS [RFC6347] for security. CoAP and HTTP proxies require (D)TLS 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 message payload and metadata, in transit between the endpoints. The proxy can also inject, delete, or reorder packets since they are no longer protected by (D)TLS.

This document defines the Object Security for Constrained RESTful Environments (OSCORE) security protocol, protecting CoAP and CoAP-mappable HTTP requests and responses end-to-end across intermediary nodes such as CoAP forward proxies and cross-protocol translators including HTTP-to-CoAP proxies [RFC8075]. In addition to the core CoAP features defined in [RFC7252], OSCORE supports Observe [RFC7641] and Blockwise [RFC7959]. An analysis of end-to-end security for CoAP messages through some types of intermediary nodes is performed in [I-D.hartke-core-e2e-security-reqs]. OSCORE protects the Request/Response layer only, and not the CoAP Messaging Layer (Section 2 of [RFC7252]). Therefore, all the CoAP messages mentioned in this document refer to non-empty CON, NON, and ACK messages. Additionally, since the message formats for CoAP over unreliable transport [RFC7252] and for CoAP over reliable transport [I-D.ietf-core-coap-tcp-tls] differ only in terms of Messaging Layer, OSCORE can be applied to both unreliable and reliable transports.

OSCORE is designed for constrained nodes and networks and provides an in-layer security protocol that does not depend on underlying layers. OSCORE can be used anywhere where CoAP or HTTP can be used, including non-IP transports (e.g., [I-D.bormann-6lo-coap-802-15-ie]). An

extension of OSCORE may also be used to protect group communication for CoAP [I-D.tiloca-core-multicast-oscoap]. The use of OSCORE does not affect the URI scheme and OSCORE can therefore be used with any URI scheme defined for CoAP or HTTP. The application decides the conditions for which OSCORE is required.

OSCORE builds on CBOR Object Signing and Encryption (COSE) [RFC8152], providing end-to-end encryption, integrity, replay protection, and secure message binding. A compressed version of COSE is used, as discussed in Section 8. The use of OSCORE is signaled with the Object-Security CoAP option or HTTP header, defined in Section 2 and Section 10.2. OSCORE is designed to protect as much information as possible, while still allowing proxy operations (Section 10). OSCORE provides protection of message payload, almost all CoAP options, and the RESTful method. The solution transforms a message into an "OSCORE message" before sending, and vice versa after receiving. The OSCORE message is related to the original message in the following way: the original message is translated to CoAP (if not already in CoAP) and the resulting message payload (if present), options not processed by a proxy, and the request/response method (CoAP Code) are protected in a COSE object. The message fields of the original messages that are encrypted are not present in the OSCORE message, and instead the Object-Security option/header and the compressed COSE object are included, see Figure 1.

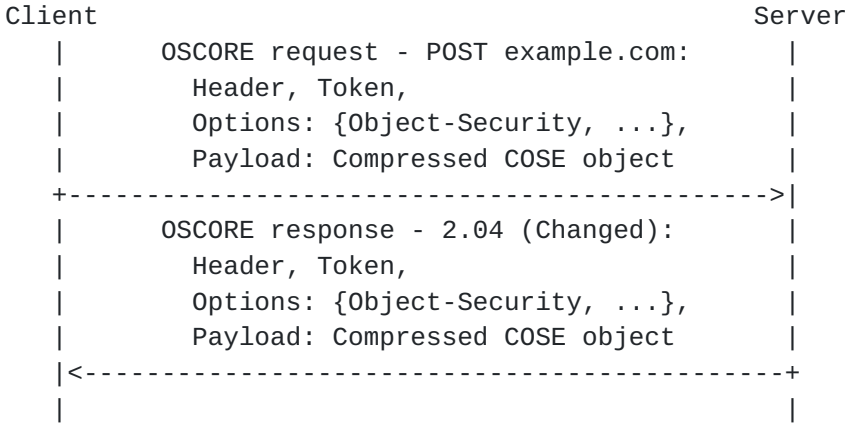


Figure 1: Sketch of OSCORE with CoAP

OSCORE may be used in very constrained settings, thanks to its small message size and the restricted code and memory requirements in addition to what is required by CoAP. OSCORE can be combined with transport layer security such as DTLS or TLS, thereby enabling end-to-end security of e.g. CoAP Payload, Options and Code, in combination with hop-by-hop protection of the Messaging Layer, during transport between end-point and intermediary node. Examples of the use of OSCORE are given in Appendix B.

An implementation supporting this specification MAY only implement the client part, MAY only implement the server part, or MAY only implement one of the proxy parts. OSCORE is designed to work with legacy CoAP-to-CoAP forward proxies [RFC7252], but an OSCORE-aware proxy will be more efficient. HTTP-to-CoAP proxies [RFC8075] and CoAP-to-HTTP proxies need to implement respective parts of this specification to work with OSCORE (see Section 10).

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 [RFC8152], CBOR [RFC7049], CDDL [I-D.greevenbosch-appsawg-cbor-cddl], and constrained environments [RFC7228].

The terms Common/Sender/Recipient Context, Master Secret/Salt, Sender ID/Key, Recipient ID/Key, and Common IV are defined in Section 3.1.

2. The CoAP Object-Security Option

The CoAP Object-Security option (see Figure 2) indicates that the CoAP message is an OSCORE message and that it contains a compressed COSE object (see Section 5 and Section 8). The Object-Security option is critical, safe to forward, part of the cache key, and not repeatable.

No.	C	U	N	R	Name	Format	Length	Default
TBD	x				Object-Security	see below	0-255	(none)

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

Figure 2: The Object-Security Option

The Object-Security option contains the OSCORE flag byte and for requests, the Sender ID (see Section 8). If the flag byte is all zero (0x00) the Option value SHALL be empty (Option Length = 0). An endpoint receiving a CoAP message without payload, that also contains an Object-Security option SHALL treat it as malformed and reject it.

A successful response to a request with the Object-Security option SHALL contain the Object-Security option. Whether error responses contain the Object-Security option depends on the error type (see Section 7).

Since the payload and most options are encrypted Section 4, and the corresponding plain text message fields of the original are not included in the OSCORE message, the processing of these fields does not expand the total message size.

A CoAP proxy SHOULD NOT cache a response to a request with an Object-Security option, since the response is only applicable to the original client's request, see Section 10.1. As the compressed COSE Object is included in the cache key, messages with the Object-Security option will never generate cache hits. For Max-Age processing, see Section 4.2.3.1.

3. The Security Context

OSCORE requires that client and server establish a shared security context used to process the COSE objects. OSCORE uses COSE with an Authenticated Encryption with Additional Data (AEAD) algorithm for protecting message data between a client and a server. In this section, we define the security context and how it is derived in client and server based on a common shared master 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 OSCORE. 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 and Servers 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.

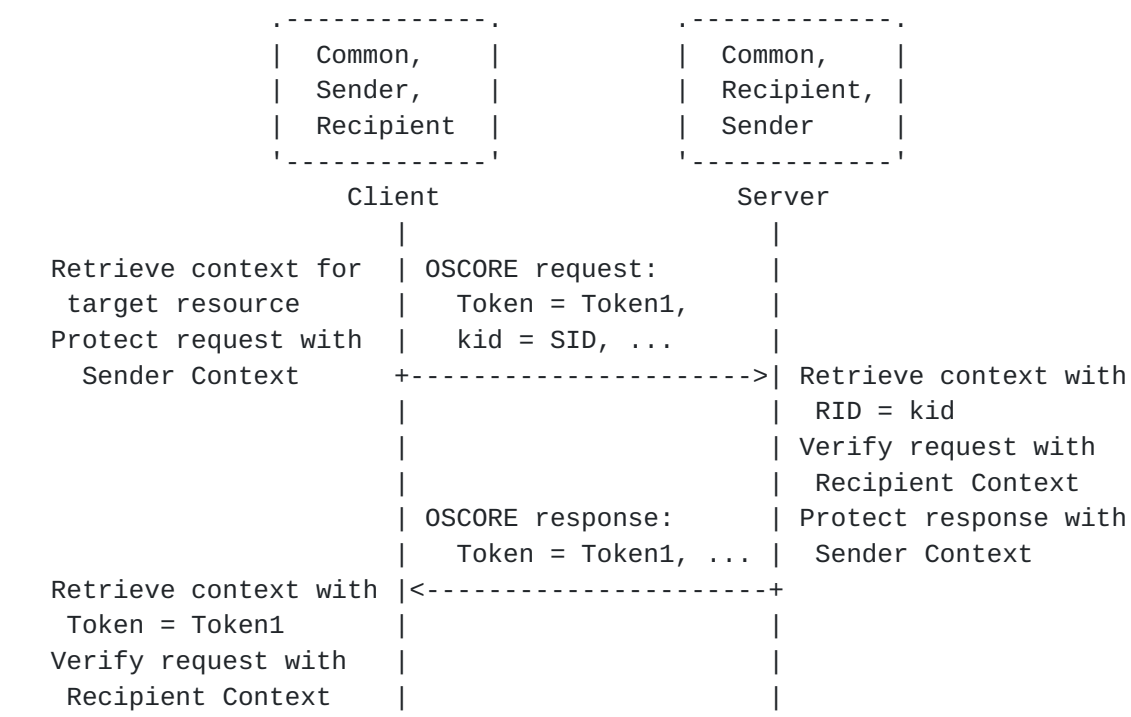


Figure 3: Retrieval and use of the Security Context

The Common Context contains the following parameters:

- AEAD Algorithm (alg). The COSE AEAD algorithm to use for encryption. Its value is immutable once the security context is established.

- Key Derivation Function. The HMAC based HKDF [RFC5869] used to derive Sender Key, Recipient Key, and Common IV.

- 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.

- 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.

- Common IV. Byte string derived from Master Secret and Master Salt. Length is determined by the AEAD Algorithm. Its value is immutable once the security context is established.

The Sender Context contains the following parameters:

- Sender ID. Non-negative integer used to identify the Sender Context and to assure unique nonces. Length is determined by the

AEAD Algorithm. Its value is immutable once the security context is established.

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

- Sender Sequence Number. Non-negative integer used by the sender to protect requests and Observe notifications. Used as "Partial IV" [RFC8152] to generate unique nonces for the AEAD. Maximum value is determined by the AEAD Algorithm.

The Recipient Context contains the following parameters:

- Recipient ID. Non-negative integer used to identify the Recipient Context and to assure unique nonces. Length is determined by the AEAD Algorithm. 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 AEAD Algorithm. Its value is immutable once the security context is established.

- Replay Window (Server only). The replay window to verify requests received.

An endpoint may free up memory by not storing the Common IV, Sender Key, and Recipient Key, deriving them from the Master Key and Master Salt 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 when changing roles. In other words, changing the roles does not change the set of keys to be used.

3.2. Establishment 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-16-64-128 (COSE algorithm encoding: 10)
- 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 ([RFC6347])

All input parameters need to be known to and agreed on by both endpoints, but the replay window may be different in the two endpoints. The replay window type and size is used by the client in the processing of the Request-Tag [I-D.amsuess-core-repeat-request-tag]. How the input parameters are pre-established, is application specific. The ACE framework may be used to establish the necessary input parameters [I-D.ietf-ace-oauth-authz].

3.2.1. Derivation of Sender Key, Recipient Key, and Common IV

The KDF MUST be one of the HMAC based HKDF [RFC5869] algorithms defined in COSE. HKDF SHA-256 is mandatory to implement. The security context parameters Sender Key, Recipient Key, and Common 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

- info is a CBOR array consisting of:

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

where:

- id is the Sender ID or Recipient ID when deriving keys and nil when deriving the Common IV. Sender ID and Recipient ID are encoded as described in Section 5

- type is "Key" or "IV"

- L is the size of the key/IV for the AEAD algorithm used, in octets

For example, if the algorithm AES-CCM-16-64-128 (see Section 10.2 in [RFC8152]) is used, the value for L is 16 for keys and 13 for the Common IV.

3.2.2. Initial Sequence Numbers and Replay Window

The Sender 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. The 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 and Master Salt. When a trusted third party assigns identifiers (e.g., using [I-D.ietf-ace-oauth-authz]) or by using a protocol that allows the parties to negotiate locally unique identifiers in each endpoint, the Sender IDs can be very short. The maximum Sender ID is 2^(nonce length in bits - 40) - 1, For AES-CCM-16-64-128 the maximum Sender ID is 2^64 - 1. 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 Client MAY provide a Context Hint Section 8.3 to help the Server find the right context.

While the triple (Master Secret, Master Salt, Sender ID) MUST be unique, the same Master Salt MAY be used with several Master Secrets and the same Master Secret MAY be used with several Master Salts.

4. Protected Message Fields

OSCORE transforms a CoAP message (which may have been generated from an HTTP message) into an OSCORE message, and vice versa. OSCORE protects as much of the original message as possible while still allowing certain proxy operations (see Section 10). This section defines how OSCORE protects the message fields and transfers them end-to-end between client and server (in any direction).

The remainder of this section and later sections discuss the behavior in terms of CoAP messages. If HTTP is used for a particular leg in the end-to-end path, then this section applies to the conceptual CoAP message that is mappable to/from the original HTTP message as discussed in Section 10. That is, an HTTP message is conceptually transformed to a CoAP message and then to an OSCORE message, and similarly in the reverse direction. An actual implementation might translate directly from HTTP to OSCORE without the intervening CoAP representation.

Message fields of the CoAP message may be protected end-to-end between CoAP client and CoAP server in different ways:

- Class E: encrypted and integrity protected,
- Class I: integrity protected only, or
- Class U: unprotected.

The sending endpoint SHALL transfer Class E message fields in the ciphertext of the COSE object in the OSCORE message. The sending endpoint SHALL include Class I message fields in the Additional Authenticated Data (AAD) of the AEAD algorithm, allowing the receiving endpoint to detect if the value has changed in transfer. Class U message fields SHALL NOT be protected in transfer. Class I and Class U message field values are transferred in the header or options part of the OSCORE message which is visible to proxies.

Message fields not visible to proxies, i.e., transported in the ciphertext of the COSE object, are called "Inner" (Class E). Message

fields transferred in the header or options part of the OSCORE message, which is visible to proxies, are called "Outer" (Class I or U).

CoAP message fields are either Inner or Outer: Inner if the value is intended for the destination endpoint, Outer if the value is intended for a proxy. An OSCORE message may contain both an Inner and an Outer message field of certain CoAP message fields. Inner and Outer message fields are processed independently.

4.1. CoAP Payload

The CoAP Payload, if present in the original CoAP message, SHALL be encrypted and integrity protected and is thus an Inner message field. The sending endpoint writes the payload of the original CoAP message into the plaintext (Section 5.2) input to the COSE object. The receiving endpoint verifies and decrypts the COSE object, and recreates the payload of the original CoAP message.

4.2. CoAP Options

A summary of how options are protected is shown in Figure 4. Options which require special processing, in particular those which may have both Inner and Outer message fields, are labelled with asterisks.

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 (Inner)
I = Integrity Protect only (Outer)
U = Unprotected (Outer)
* = Special

Figure 4: Protection of CoAP Options

Unknown CoAP options SHALL be processed as class E (and no special processing). Specifications of new CoAP options SHOULD define how they are processed with OSCORE. A new COAP option SHOULD be of class E unless it requires proxy processing.

4.2.1. Inner Options

When using OSCORE, Inner option message fields (marked in column E of Figure 4) are sent in a way analogous to communicating in a protected manner directly with the other endpoint.

The sending endpoint SHALL write the Inner option message fields present in the original CoAP message into the plaintext of the COSE object Section 5.2, and then remove the Inner option message fields from the OSCORE message.

The processing of Inner option message fields by the receiving endpoint is specified in Section 7.2 and Section 7.4.

4.2.2. Outer Options

Outer option message fields (marked in column U or I of Figure 4) are used to support proxy operations.

The sending endpoint SHALL include the Outer option message field present in the original message in the options part of the OSCORE message. All Outer option message fields, including Object-Security, SHALL be encoded as described in Section 3.1 of [RFC7252], where the delta is the difference to the previously included Outer option message field.

The processing of Outer options by the receiving endpoint is specified in Section 7.2 and Section 7.4.

A procedure for integrity-protection-only of Class I option message fields is specified in Section 5.3.

Note: There are currently no Class I option message fields defined.

4.2.3. Special Options

Some options require special processing, marked with an asterisk '*' in Figure 4. An asterisk in the columns E and U indicate that the option may be added as an Inner and/or Outer message by the sending endpoint; the processing is specified in this section.

4.2.3.1. Max-Age

The Inner Max-Age option is used to specify the freshness (as defined in [RFC7252]) of the resource, end-to-end from the server to the client, taking into account that the option is not accessible to proxies. The Inner Max-Age SHALL be processed by OSCORE as specified in Section 4.2.1.

The Outer Max-Age option is used to avoid unnecessary caching of OSCORE responses at OSCORE unaware intermediary nodes. A server MAY set a Class U Max-Age option with value zero to Observe responses (see Section 5.6.1 of [RFC7252]) which is then processed according to Section 4.2.2. The Outer Max-Age option value SHALL be discarded by the OSCORE client.

Non-Observe OSCORE responses do not need to include a Max-Age option since the responses are non-cacheable by construction (see Section 4.3).

4.2.3.2. The Block Options

Blockwise [RFC7959] is an optional feature. An implementation MAY support [RFC7252] and the Object-Security option without supporting [RFC7959]. The Block options are used to secure message fragmentation end-to-end (Inner options) or for proxies to fragment the OSCORE message for the next hop (Outer options). Inner and Outer block processing may have different performance properties depending on the underlying transport. The integrity of the message can be verified end-to-end both in case of Inner and Outer Blockwise provided all blocks are received (see Section 4.2.3.2.2).

4.2.3.2.1. Inner Block Options

The sending CoAP endpoint MAY fragment a CoAP message as defined in [RFC7959] before the message is processed by OSCORE. In this case the Block options SHALL be processed by OSCORE as Inner options (Section 4.2.1). The receiving CoAP endpoint SHALL process the OSCORE message according to Section 4.2.1 before processing blockwise as defined in [RFC7959].

For blockwise request operations using Block1, an endpoint MUST comply with the Request-Tag processing defined in Section 3 of [I-D.amsuess-core-repeat-request-tag]. In particular, the rules in section 3.3.1 of [I-D.amsuess-core-repeat-request-tag] MUST be followed, which guarantee that a specific request body is assembled only from the corresponding request blocks.

For blockwise response operations using Block2, an endpoint MUST comply with the ETag processing defined in Section 4 of [I-D.amsuess-core-repeat-request-tag].

4.2.3.2.2. Outer Block Options

Proxies MAY fragment an OSCORE message using [RFC7959], which then introduces Outer Block options not generated by the sending endpoint. Note that the Outer Block options are neither encrypted nor integrity protected. As a consequence, a proxy can maliciously inject block fragments indefinitely, since the receiving endpoint needs to receive the last block (see [RFC7959]) to be able to compose the OSCORE message and verify its integrity. Therefore, applications supporting OSCORE and [RFC7959] MUST specify a security policy defining a maximum unfragmented message size (MAX_UNFRAGMENTED_SIZE) considering the maximum size of message which can be handled by the endpoints. Messages exceeding this size SHOULD be fragmented by the sending endpoint using Inner Block options (Section 4.2.3.2.1).

An endpoint receiving an OSCORE message with an Outer Block option SHALL first process this option according to [RFC7959], until all blocks of the OSCORE message have been received, or the cumulated message size of the blocks exceeds MAX_UNFRAGMENTED_SIZE. In the former case, the processing of the OSCORE message continues as defined in this document. In the latter case the message SHALL be discarded.

To allow multiple concurrent request operations to the same server (not only same resource), a CoAP proxy SHOULD follow the Request-Tag processing specified in section 3.3.2 of [I-D.amsuess-core-repeat-request-tag].

4.2.3.3. Proxy-Uri

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

The sending endpoint SHALL first decompose the Proxy-Uri value of the original 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 SHALL be protected and processed as Inner options (Section 4.2.1).

The Proxy-Uri option of the OSCORE message SHALL be set to the composition of Proxy-Scheme, Uri-Host and Uri-Port options (if present) as specified in section 6.5 of [RFC7252], and processed as an Outer option of Class U (Section 4.2.2).

Note that replacing the Proxy-Uri value with the Proxy-Scheme and Uri-* options works by design for all CoAP URIs (see Section 6 of [RFC7252]. OSCORE-aware HTTP servers should not use the userinfo component of the HTTP URI (as defined in section 3.2.1. of [RFC3986]), so that this type of replacement is possible in the presence of CoAP-to-HTTP proxies. In other documents specifying cross-protocol proxying behavior using different URI structures, it is expected that the authors will create Uri-* options that allow decomposing the Proxy-Uri, and specify in which OSCORE class they belong.

An example of how Proxy-Uri is processed is given here. Assume that the original CoAP message contains:

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

During OSCORE processing, Proxy-Uri is split into:

- Proxy-Scheme = "coap"
- Uri-Host = "example.com"
- Uri-Port = "5683"
- Uri-Path = "resource"
- Uri-Query = "q=1"

Uri-Path and Uri-Query follow the processing defined in Section 4.2.1, and are thus encrypted and transported in the COSE object. The remaining options are composed into the Proxy-Uri included in the options part of the OSCORE message, which has value:

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

See Section 6.1 and 12.6 of [RFC7252] for more information.

4.2.3.4. 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 an OSCORE-unaware proxy to support forwarding of Observe messages ([RFC7641]), there SHALL be an Outer Observe option, i.e., present in the options part of the OSCORE message. The processing of the CoAP Code for Observe messages is described in Section 4.3.

To secure the order of notifications, the client SHALL maintain a Notification Number for each Observation it registers. The Notification Number is a non-negative integer containing the largest Partial IV of the successfully received notifications for the associated Observe registration, see Section 6.4. The Notification Number is initialized to the Partial IV of the first successfully received notification response to the registration request. In contrast to [RFC7641], the received Partial IV MUST always be compared with the Notification Number, which thus MUST NOT be forgotten after 128 seconds.

If the verification fails, the client SHALL stop processing the response, and in the case of CON respond with an empty ACK. The client MAY ignore the Observe option value.

The Observe option in the CoAP request may be legitimately removed by a proxy. If the Observe option is removed from a CoAP request by a proxy, then the server can still verify the request (as a non-Observe request), and produce a non-Observe response. If the OSCORE client receives a response to an Observe request without an outer Observe value, then it MUST verify the response as a non-Observe response. (The reverse case is covered in the verification of the response, see Section 7.)

4.3. CoAP Header

Most CoAP header fields are required to be read and/or changed by CoAP proxies and thus cannot in general be protected end-to-end between the endpoints. As mentioned in Section 1, OSCORE protects the CoAP Request/Response layer only, and not the Messaging Layer (Section 2 of [RFC7252]), so fields such as Type and Message ID are not protected with OSCORE.

The CoAP header field Code is protected by OSCORE. Code SHALL be encrypted and integrity protected (Class E) to prevent an intermediary from eavesdropping or manipulating the Code (e.g., changing from GET to DELETE).

The sending endpoint SHALL write the Code of the original CoAP message into the plaintext of the COSE object Section 5.2. After that, the Outer Code of the OSCORE message SHALL be set to 0.02 (POST) for requests and to 2.04 (Changed) for responses, except for Observe messages. For Observe messages, the Outer Code of the OSCORE message SHALL be set to 0.05 (FETCH) for requests and to 2.05 (Content) for responses. This exception allows OSCORE to be compliant with the Observe processing in OSCORE-unaware proxies. The choice of POST and FETCH ([RFC8132]) allows all OSCORE messages to have payload.

The receiving endpoint SHALL discard the Code in the OSCORE message and write the Code of the Plaintext in the COSE object (Section 5.2) into the decrypted CoAP message.

The other CoAP header fields are Unprotected (Class U). The sending endpoint SHALL write all other header fields of the original message into the header of the OSCORE message. The receiving endpoint SHALL write the header fields from the received OSCORE message into the header of the decrypted CoAP message.

5. The COSE Object

This section defines how to use COSE [RFC8152] to wrap and protect data in the original message. OSCORE uses the untagged COSE_Encrypt0 structure with an Authenticated Encryption with Additional Data (AEAD) algorithm. The key lengths, IV length, nonce length, and maximum Sender Sequence Number are algorithm dependent.

The AEAD algorithm AES-CCM-16-64-128 defined in Section 10.2 of [RFC8152] is mandatory to implement. For AES-CCM-16-64-128 the length of Sender Key and Recipient Key is 128 bits, the length of nonce and Common IV is 13 bytes. The maximum Sender Sequence Number is specified in Section 11.

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 is empty.

- The "unprotected" field includes:

  * The "Partial IV" parameter. The value is set to the Sender Sequence Number. All leading zeroes SHALL be removed when encoding the Partial IV, i.e. the first byte (if any) SHALL never be zero. This parameter SHALL be present in requests. In case of Observe (Section 4.2.3.4) the Partial IV SHALL be present in responses, and otherwise the Partial IV SHALL NOT be present in responses.

  * The "kid" parameter. The value is set to the Sender ID (see Section 3). All leading zeroes SHALL be removed when encoding the Partial IV, i.e. the first byte (if any) SHALL never be zero. This parameter SHALL be present in requests and SHALL NOT be present in responses.

- The "ciphertext" field is computed from the secret key (Sender Key or Recipient Key), Nonce (see Section 5.1), Plaintext (see Section 5.2), and the Additional Authenticated Data (AAD) (see Section 5.3) following Section 5.2 of [RFC8152].

The encryption process is described in Section 5.3 of [RFC8152].

5.1. Nonce

The nonce is constructed by left-padding the Partial IV (in network byte order) with zeroes to exactly 5 bytes, left-padding the Sender ID of the endpoint that generated the Partial IV (in network byte order) with zeroes to exactly nonce length - 5 bytes, concatenating the padded Partial IV with the padded ID, and then XORing with the Common IV.

When observe is not used, the request and the response uses the same nonce. In this way, the Partial IV does not have to be sent in responses, which reduces the size. For processing instructions, see Section 7.

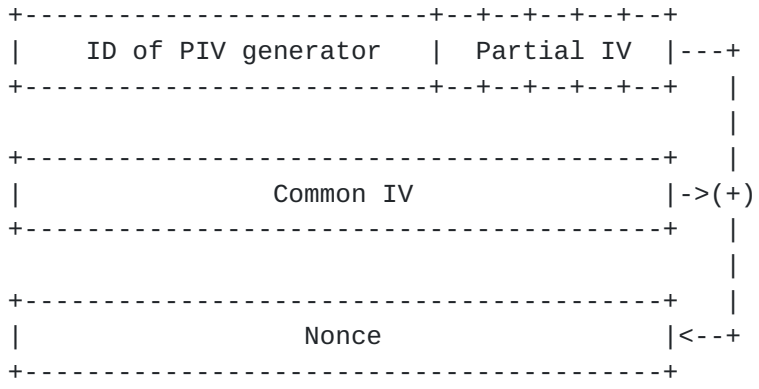


Figure 5: AEAD Nonce Formation

5.2. Plaintext

The Plaintext is formatted as a CoAP message without Header (see Figure 6) consisting of:

- the Code of the original CoAP message as defined in Section 3 of [RFC7252]; and

- all Inner option message fields (see Section 4.2.1) present in the original CoAP message (see Section 4.2). 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 original CoAP message, if present, and in that case prefixed by the one-byte Payload Marker (0xFF).

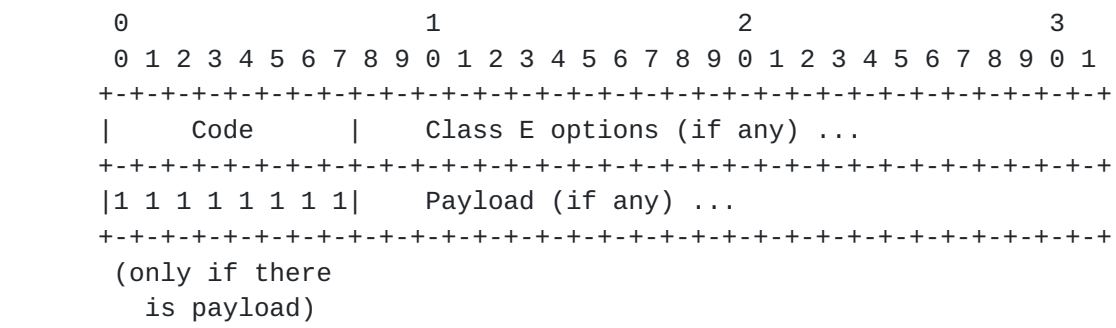


Figure 6: Plaintext

5.3. Additional Authenticated Data

The external_aad SHALL be a CBOR array as defined below:

```
external_aad = [
   version : uint,
   alg : int,
   request_kid : bstr,
   request_piv : bstr,
   options : bstr
]
```

where:

- version: contains the OSCORE version number. Implementations of this specification MUST set this field to 1. Other values are reserved for future versions.

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

- options: contains the (non-special) Class I options (see Section 4.2.2) present in the original CoAP message encoded as described in Section 3.1 of [RFC7252], where the delta is the difference to the previously included class I option.

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

6.1. Message Binding

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

6.2. 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 Sender Sequence Number used to protect requests, and - in case of Observe - responses. If messages are processed concurrently, the operation of reading and increasing the Sender Sequence Number MUST be atomic.

The maximum Sender Sequence Number is algorithm dependent, see Section 11. If the Sender Sequence Number exceeds the maximum, 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.

6.3. Freshness

For requests, OSCORE provides weak absolute freshness as the only guarantee is that the request is not older than the security context. For applications having stronger demands on request freshness (e.g., control of actuators), OSCORE needs to be augmented with mechanisms providing freshness [I-D.amsuess-core-repeat-request-tag].

For responses, the message binding guarantees that a response is not older than its request. For responses without Observe, this gives strong absolute freshness. For responses with Observe, the absolute freshness gets weaker with time, and it is RECOMMENDED that the client regularly restart the observation.

For requests, and responses with Observe, OSCORE also provides relative freshness in the sense that the received Partial IV allows a recipient to determine the relative order of responses.

6.4. Replay Protection

In order to protect from replay of requests, the server's Recipient Context includes a Replay Window. A server SHALL verify that a Partial IV received in the COSE object has not been received before. If this verification fails and the message received is a CON message, the server SHALL respond with a 5.03 Service Unavailable error message with the inner Max-Age option set to 0. The diagnostic payload MAY contain the "Replay protection failed" string. 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 server MAY just store the last received Partial IV and require that newly received Partial IVs equals the last received Partial IV + 1.

Responses to non-Observe requests are protected against replay as they are cryptographically bound to the request.

In the case of Observe, a client receiving a notification SHALL verify that the Partial IV of a received notification is greater than the Notification Number bound to that Observe registration. If the verification fails, the client SHALL stop processing the response, and in the case of CON respond with an empty ACK. If the verification succeeds, the client SHALL overwrite the corresponding Notification Number with the received Partial IV.

If messages are processed concurrently, the Partial IV needs to be validated a second time after decryption and before updating the replay protection data. The operation of validating the Partial IV and updating the replay protection data MUST be atomic.

6.5. Losing Part of the Context State

To prevent reuse of the Nonce with the same key, or from accepting replayed messages, a node needs to handle the situation of losing rapidly changing parts of the context, such as the request Token, Sender Sequence Number, Replay Window, and Nofitifcation Numbers. These are typically stored in RAM and therefore lost in the case of an unplanned 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. However, establishing a fresh security context may have a non-negligible cost in terms of, e.g., power consumption.

After boot, a node MAY use a partly persistently stored security context, but then the node MUST NOT reuse a previous Sender Sequence

Number and MUST NOT accept previously accepted messages. Some ways to achieve this is described below:

6.5.1. Sequence Number

To prevent reuse of Sender Sequence Numbers, a node MAY perform the following procedure during normal operations:

- Each time the Sender Sequence Number is evenly divisible by K, where K is a positive integer, store the Sender Sequence Number in persistent memory. After boot, the node initiates the Sender Sequence Number to the value stored in persistent memory + K - 1. Storing to persistent memory can be costly. The value K gives a trade-off between the number of storage operations and efficient use of Sender Sequence Numbers.

6.5.2. Replay Window

To prevent accepting replay of previously received requests, the server MAY perform the following procedure after boot:

- For each stored security context, the first time after boot the server receives an OSCORE request, the server uses the Repeat option [I-D.amsuess-core-repeat-request-tag] to get a request with verifiable freshness and uses that to synchronize the replay window. If the server can verify the fresh request, the Partial IV in the fresh request is set as the lower limit of the replay window.

6.5.3. Replay Protection of Observe Notifications

To prevent accepting replay of previously received notification responses, the client MAY perform the following procedure after boot:

- The client rejects notifications bound to the earlier registration, removes all Notification Numbers and re-register using Observe.

7. Processing

This section describes the OSCORE message processing.

7.1. Protecting the Request

Given a CoAP request, the client SHALL perform the following steps to create an OSCORE request:

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

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

3. Compute the AEAD nonce from the Sender ID, Common IV, and Partial IV (Sender Sequence Number in network byte order). Then (in one atomic operation, see Section 6.2) increment the Sender Sequence Number by one.

4. Encrypt the COSE object using the Sender Key. Compress the COSE Object as specified in Section 8.

5. Format the OSCORE message according to Section 4. The Object-Security option is added, see Section 4.2.2.

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 4.2.3.2.

2. Discard the message Code and all non-special Inner option message fields (marked with 'x' in column E of Figure 4) present in the received message. For example, an If-Match Outer option is discarded, but an Uri-Host Outer option is not discarded.

3. Decompress the COSE Object (Section 8) and retrieve the Recipient Context associated with the Recipient ID in the 'kid' parameter. If the request is a NON message and either the decompression or the COSE message fails to decode, or the server fails to retrieve a Recipient Context with Recipient ID corresponding to the 'kid' parameter received, then the server SHALL stop processing the request. If the request is a CON message, and:

   * either the decompression or the COSE message fails to decode, the server SHALL respond with a 4.02 Bad Option error message. The diagnostic payload SHOULD contain the string "Failed to decode COSE".

   * the server fails to retrieve a Recipient Context with Recipient ID corresponding to the 'kid' parameter received, the server SHALL respond with a 4.01 Unauthorized error

message. The diagnostic payload MAY contain the string "Security context not found".

4. Verify the 'Partial IV' parameter using the Replay Window, as described in Section 6.

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

6. Compute the AEAD nonce from the Recipient ID, Common IV, and the 'Partial IV' parameter, received in the COSE Object.

7. Decrypt the COSE object using the Recipient Key.

   * If decryption fails, the server MUST stop processing the request and, if the request is a CON message, the server MUST respond with a 4.00 Bad Request error message. The diagnostic payload MAY contain the "Decryption failed" string.

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

8. For each decrypted option, check if the option is also present as an Outer option: if it is, discard the Outer. For example: the message contains a Max-Age Inner and a Max-Age Outer option. The Outer Max-Age is discarded.

9. Add decrypted code, options and payload to the decrypted request. The Object-Security option is removed.

10. The decrypted CoAP request is processed according to [RFC7252]

7.3. Protecting the Response

Given a CoAP response, the server SHALL perform the following steps to create an OSCORE response. Note that CoAP error responses derived from CoAP processing (point 10. in Section 7.2) are protected, as well as successful CoAP responses, while the OSCORE errors (point 3., 4., 7. in Section 7.2) do not follow the processing below, but are sent as simple CoAP responses, without OSCORE processing.

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. Compute the AEAD nonce

   * If Observe is not used, the nonce from the request is used.

   * If Observe is used, Compute the AEAD nonce from the Sender ID, Common IV, and Partial IV (Sender Sequence Number in network byte order). Then (in one atomic operation, see Section 6.2) increment the Sender Sequence Number by one.

4. Encrypt the COSE object using the Sender Key. Compress the COSE Object as specified in Section 8.

5. Format the OSCORE message according to Section 4. The Object-Security option is added, see Section 4.2.2.

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 OSCORE message have been received, see Section 4.2.3.2.

2. Discard the message Code and all non-special Class E options from the message. For example, ETag Outer option is discarded, Max-Age Outer option is not discarded.

3. Retrieve the Recipient Context associated with the Token. Decompress the COSE Object (Section 8). If either the decompression or the COSE message fails to decode, then go to 11.

4. For Observe notifications, verify the received 'Partial IV' parameter against the corresponding Notification Number as described in Section 6. If the client receives a notification for which no Observe request was sent, then go to 11.

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

6. Compute the AEAD nonce

   * If the Observe option is not present in the response, the nonce from the request is used.

* If the Observe option is present in the response, compute the AEAD nonce from the Recipient ID, Common IV, and the 'Partial IV' parameter, received in the COSE Object.

7. Decrypt the COSE object using the Recipient Key.

   * If decryption fails, then go to 11.

   * If decryption succeeds and Observe is used, update the corresponding Notification Number, as described in Section 6.

8. For each decrypted option, check if the option is also present as an Outer option: if it is, discard the Outer. For example: the message contains a Max-Age Inner and a Max-Age Outer option. The Outer Max-Age is discarded.

9. Add decrypted code, options and payload to the decrypted request. The Object-Security option is removed.

10. The decrypted CoAP response is processed according to [RFC7252]

11. (Optional) In case any of the previous erroneous conditions apply: if the response is a CON message, then the client SHALL send an empty ACK back and stop processing the response; if the response is a ACK or a NON message, then the client SHALL simply stop processing the response.

8. OSCORE Compression

The Concise Binary Object Representation (CBOR) [RFC7049] combines very small message sizes with extensibility. The CBOR Object Signing and Encryption (COSE) [RFC8152] uses CBOR to create compact encoding of signed and encrypted data. 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 OSCORE called the "compressed COSE object", which significantly reduces the per-packet overhead.

8.1. Encoding of the Object-Security Value

The value of the Object-Security option SHALL contain the OSCORE flag byte and the kid parameter as follows:

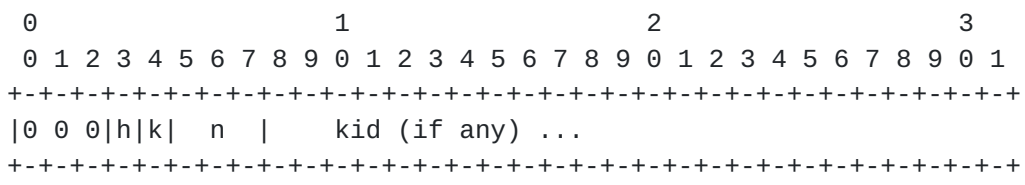


Figure 7: Object-Security Value

- The first byte (= the OSCORE flag byte) encodes a set of flags and the length of the Partial IV parameter.
  - The three least significant bits, n, encode the Partial IV length + 1. If n = 0 then the Partial IV is not present in the compressed COSE object. The value n = 7 is reserved.
  - The fourth least significant bit is the kid flag, k: it is set to 1 if the kid is present in the compressed COSE object.
  - The fifth least significant bit is the Context Hint flag, h: it is set to 1 if the compressed COSE object contains a Context Hint, see Section 8.3.
  - The sixth-eighth least significant bits are reserved and SHALL be set to zero when not in use.
- The remaining bytes encode the value of the kid, if the kid is present (k = 1)

The presence of Partial IV and kid in requests and responses is specified in Section 5, and summarized in Figure 8.

	k	n
Request	1	> 0
Response without Observe	0	0
Response with Observe	0	> 0

Figure 8: Presence of data fields in OSCORE flag byte

8.2. Encoding of the OSCORE Payload

The payload of the OSCORE message SHALL be encoded as follows:

- The first n - 1 bytes encode the value of the Partial IV, if the Partial IV is present (n > 0).

- The following 1 byte encode the length of the Context Hint (Section 8.3), s, if the Context Hint flag is set (h = 1).

- The following s bytes encode the Context Hint, if the Context Hint flag is set (h = 1).

- The remaining bytes encode the ciphertext.

8.3. Context Hint

For certain use cases, e.g. deployments where the same Recipient ID is used with multiple contexts, it is necessary or favorable for the sending endpoint to provide a Context Hint in order for the receiving endpoint to retrieve the recipient context. The Context Hint is implicitly integrity protected, as a manipulation leads to the wrong or no context being retrieved resulting in a verification error.

Examples:

- If the sending endpoint has an identifier in some other namespace which can be used by the recipient endpoint to retrieve or establish the security context, then that identifier can be used as Context Hint.

- In case of a group communication scenario [I-D.tiloca-core-multicast-oscoap], if the recipient endpoint belongs to multiple groups, involving the same endpoints, then a group identifier can be used as Context Hint to enable the receiving endpoint to find the right group security context.

8.4. Compression Examples

This section provides examples of COSE Objects before and after OSCORE compression.

8.4.1. Example: Request

Before compression:

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

```
0x83 40 a2 04 41 25 06 41 05 4e ae a0 15 56 67 92
4d ff 8a 24 e4 cb 35 b9 (24 bytes)
```

After compression:

Flag byte: 0b00001010 = 0x0a

Option Value: 0a 25 (2 bytes)

Payload: 05 ae a0 15 56 67 92 4d ff 8a 24 e4 cb 35 b9 (15 bytes)

8.4.2. Example: Request 2

Before compression:

```
[
h'',
{ 4:h'00', 6:h'00' },
h'aea0155667924dff8a24e4cb35b9'
]
```

0x83 40 a2 04 41 00 06 41 00 4e ae a0 15 56 67 92
4d ff 8a 24 e4 cb 35 b9 (24 bytes)

After compression:

Flag byte: 0b00001001 = 0x09

Option Value: 09 (1 bytes)

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

8.4.3. Example: Response (without Observe)

Before compression:

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

0x83 40 a0 4e ae a0 15 56 67 92 4d ff 8a 24 e4 cb
35 b9 (18 bytes)

After compression:

Flag byte: 0b00000000 = 0x00

Option Value: (0 bytes)

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

8.4.4. Example: Response (with Observe)

Before compression:

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

0x83 40 a1 06 41 07 4e ae a0 15 56 67 92 4d ff
8a 24 e4 cb 35 b9 (21 bytes)

After compression:

Flag byte: 0b00000010 = 0x02

Option Value: 02 (1 bytes)

Payload: 07 ae a0 15 56 67 92 4d ff 8a 24 e4 cb 35 b9 (15 bytes)

9. Web Linking

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

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.

10. Proxy Operations

RFC 7252 defines operations for a CoAP-to-CoAP proxy (see Section 5.7 of [RFC7252]) and for proxying between CoAP and HTTP (Section 10 of [RFC7252]). A more detailed description of the HTTP-to-CoAP mapping is provided by [RFC8075]. This section describes the operations of OSCORE-aware proxies.

10.1. CoAP-to-CoAP Forwarding Proxy

OSCORE is designed to work with legacy CoAP-to-CoAP forward proxies [RFC7252], but OSCORE-aware proxies provide certain simplifications as specified in this section.

The targeted proxy operations are specified in Section 2.2.1 of [I-D.hartke-core-e2e-security-reqs]. In particular caching is disabled since the CoAP response is only applicable to the original client's CoAP request. An OSCORE-aware proxy SHALL NOT cache a response to a request with an Object-Security option. As a consequence, the search for cache hits and CoAP freshness/Max-Age processing can be omitted.

Proxy processing of the (Outer) Proxy-Uri option is as defined in [RFC7252].

Proxy processing of the (Outer) Block options is as defined in [RFC7959] and [I-D.amsuess-core-repeat-request-tag].

Proxy processing of the (Outer) Observe option is as defined in [RFC7641]. OSCORE-aware proxies MAY look at the Partial IV value instead of the Outer Observe option.

10.2. HTTP-to-CoAP Translation Proxy

Section 10.2 of [RFC7252] and [RFC8075] specify the behavior of an HTTP-to-CoAP proxy. As requested in Section 1 of [RFC8075], this section describes the HTTP mapping for the OSCORE protocol extension of CoAP.

The presence of the Object-Security option, both in requests and responses, is expressed in an HTTP header field named Object-Security in the mapped request or response. The value of the field is the value of the Object-Security option Section 8.1 in base64url encoding (Section 5 of [RFC4648]) without padding (see [RFC7515] Appendix C for implementation notes for this encoding). The value of the payload is the OSCORE payload Section 8.2, also base64url-encoded without padding.

Example:

Mapping and notation here is based on "Simple Form" (Section 5.4.1.1 of [RFC8075]).

```
 [HTTP request -- Before object security processing]

   GET http://proxy.url/hc/?target_uri=coap://device.url/orders HTTP/1.1

 [HTTP request -- HTTP Client to Proxy]

   POST http://proxy.url/hc/?target_uri=coap://device.url/ HTTP/1.1
   Object-Security: 0b 25
   Body: 09 07 01 13 61 f7 0f d2 97 b1 [binary]

 [CoAP request -- Proxy to CoAP Server]

   POST coap://device.url/
   Object-Security: 0b 25
   Payload: 09 07 01 13 61 f7 0f d2 97 b1 [binary]

 [CoAP response -- CoAP Server to Proxy]

   2.04 Changed
   Object-Security: [empty]
   Payload: 00 31 d1 fc f6 70 fb 0c 1d d5 ... [binary]

 [HTTP response -- Proxy to HTTP Client]

   HTTP/1.1 200 OK
   Object-Security: [empty]
   Body: 00 31 d1 fc f6 70 fb 0c 1d d5 ... [binary]

 [HTTP response -- After object security processing]

   HTTP/1.1 200 OK
   Body: Exterminate! Exterminate!
```

Note that the HTTP Status Code 200 in the next-to-last message is the mapping of CoAP Code 2.04 (Changed), whereas the HTTP Status Code 200 in the last message is the mapping of the CoAP Code 2.05 (Content), which was encrypted within the compressed COSE object carried in the Body of the HTTP response.

10.3. CoAP-to-HTTP Translation Proxy

Section 10.1 of [RFC7252] describes the behavior of a CoAP-to-HTTP proxy. RFC 8075 [RFC8075] does not cover this direction in any more detail and so an example instantiation of Section 10.1 of [RFC7252] is used below.

Example:

```
   [CoAP request -- Before object security processing]

     GET coap://proxy.url/
     Proxy-Uri=http://device.url/orders

   [CoAP request -- CoAP Client to Proxy]

     POST coap://proxy.url/
     Proxy-Uri=http://device.url/
     Object-Security: 0b 25
     Payload: 09 07 01 13 61 f7 0f d2 97 b1 [binary]

   [HTTP request -- Proxy to HTTP Server]

     POST http://device.url/ HTTP/1.1
     Object-Security: 0b 25
     Body: 09 07 01 13 61 f7 0f d2 97 b1 [binary]

   [HTTP response -- HTTP Server to Proxy]

     HTTP/1.1 200 OK
     Object-Security: [empty]
     Body: 00 31 d1 fc f6 70 fb 0c 1d d5 ... [binary]

   [CoAP response -- CoAP Server to Proxy]

     2.04 Changed
     Object-Security: [empty]
     Payload: 00 31 d1 fc f6 70 fb 0c 1d d5 ... [binary]

   [CoAP response -- After object security processing]

     2.05 Content
     Payload: Exterminate! Exterminate!
```

Note that the HTTP Code 2.04 (Changed) in the next-to-last message is the mapping of HTTP Status Code 200, whereas the CoAP Code 2.05 (Content) in the last message is the value that was encrypted within the compressed COSE object carried in the Body of the HTTP response.

11. Security Considerations

In scenarios with intermediary nodes such as proxies or brokers, transport layer security such as (D)TLS 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.

(D)TLS protects hop-by-hop the entire message, including header, options, and payload. OSCORE protects end-to-end the payload, and all information in the options and header, that is not required for proxy operations (see Section 4). (D)TLS and OSCORE can be combined, thereby enabling end-to-end security of the message payload, in combination with hop-by-hop protection of the entire message, during transport between end-point and intermediary node. The message layer, however, cannot be protected end-to-end through intermediary devices since, even if the protocol itself isn't translated, the parameters Type, Message ID, Token, and Token Length may be changed by a proxy.

The use of COSE to protect 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 the ACE framework [I-D.ietf-ace-oauth-authz]. An OSCORE profile of ACE is described in [I-D.seitz-ace-oscoap-profile].

Most AEAD algorithms require a unique nonce for each message, for which the sender 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 maximum sender sequence number is dependent on the AEAD algorithm. The maximum sender sequence number SHALL be 2^40 - 1, or any algorithm specific lower limit. The compression mechanism (Section 8) assumes that the Partial IV is 40 bits or less. The mandatory-to-implement AEAD algorithm AES-CCM-16-64-128 is selected for compatibility with CCM*.

The inner block options enable the sender to split large messages into OSCORE-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.

12. Privacy Considerations

Privacy threats executed through intermediate nodes are considerably reduced by means of OSCORE. End-to-end integrity protection and encryption of the message payload and all options that are not used for proxy operations, 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.

Using the mechanisms described in Section 6.5 may reveal when a device goes through a reboot. This can be mitigated by the device storing the precise state of sender sequence number and replay window on a clean shutdown.

The length of message fields can reveal information about the message. Applications may use a padding scheme to protect against traffic analysis. 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. Similar arguments apply to other message fields such as resource names.

13. IANA Considerations

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

13.1. CoAP Option Numbers Registry

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

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

13.2. Header Field Registrations

The HTTP header field Object-Security is added to the Message Headers registry:

Header Field Name	Protocol	Status	Reference
Object-Security	http	standard	[[this document]]

14. Acknowledgments

The following individuals provided input to this document: Christian Amsuess, Tobias Andersson, Carsten Bormann, Joakim Brorsson, Thomas Fossati, Martin Gunnarsson, Klaus Hartke, Jim Schaad, Dave Thaler, Marco Tiloca, and Malisa Vučinić.

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

15. References

15.1. Normative References

[I-D.amsuess-core-repeat-request-tag]
Amsuess, C., Mattsson, J., and G. Selander, "Repeat And Request-Tag", draft-amsuess-core-repeat-request-tag-00 (work in progress), July 2017.

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

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5988] Nottingham, M., "Web Linking", RFC 5988, DOI 10.17487/RFC5988, October 2010, <https://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, <https://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, <https://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, <https://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, <https://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, <https://www.rfc-editor.org/info/rfc7959>.

[RFC8075] Castellani, A., Loreto, S., Rahman, A., Fossati, T., and E. Dijk, "Guidelines for Mapping Implementations: HTTP to the Constrained Application Protocol (CoAP)", RFC 8075, DOI 10.17487/RFC8075, February 2017, <https://www.rfc-editor.org/info/rfc8075>.

[RFC8132] van der Stok, P., Bormann, C., and A. Sehgal, "PATCH and FETCH Methods for the Constrained Application Protocol (CoAP)", RFC 8132, DOI 10.17487/RFC8132, April 2017, <https://www.rfc-editor.org/info/rfc8132>.

[RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <https://www.rfc-editor.org/info/rfc8152>.

15.2. 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]
Birkholz, H., Vigano, C., and C. Bormann, "Concise data definition language (CDDL): a notational convention to express CBOR data structures", draft-greevenbosch-appsawg-cbor-cddl-11 (work in progress), July 2017.

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

[I-D.ietf-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-07 (work in progress), August 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-09 (work in progress), May 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., Palombini, F., and M. Gunnarsson, "OSCOAP profile of the Authentication and Authorization for Constrained Environments Framework", draft-seitz-ace-oscoap-profile-04 (work in progress), July 2017.

[I-D.tiloca-core-multicast-oscoap]
Tiloca, M., Selander, G., and F. Palombini, "Secure group communication for CoAP", draft-tiloca-core-multicast-oscoap-03 (work in progress), July 2017.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

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

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

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

Appendix A. Test Vectors

TODO: This section needs to be updated.

Appendix B. Examples

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

B.1. Secure Access to 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.

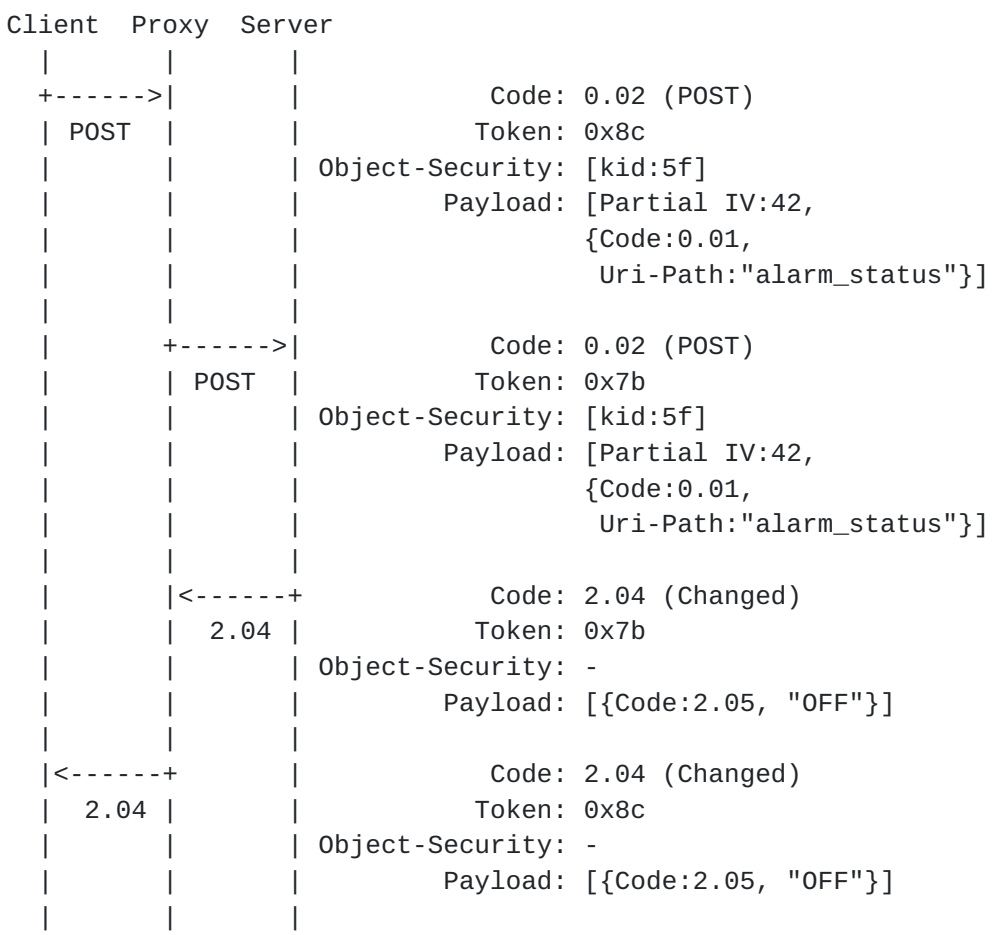


Figure 9: Secure Access to Sensor. Square brackets [...] indicate a COSE object. Curly brackets { ... } indicate encrypted data.

The request/response Codes are encrypted by OSCORE and only dummy Codes (POST/Changed) are visible in the header of the OSCORE message. The option Uri-Path ("alarm_status") and payload ("OFF") are encrypted.

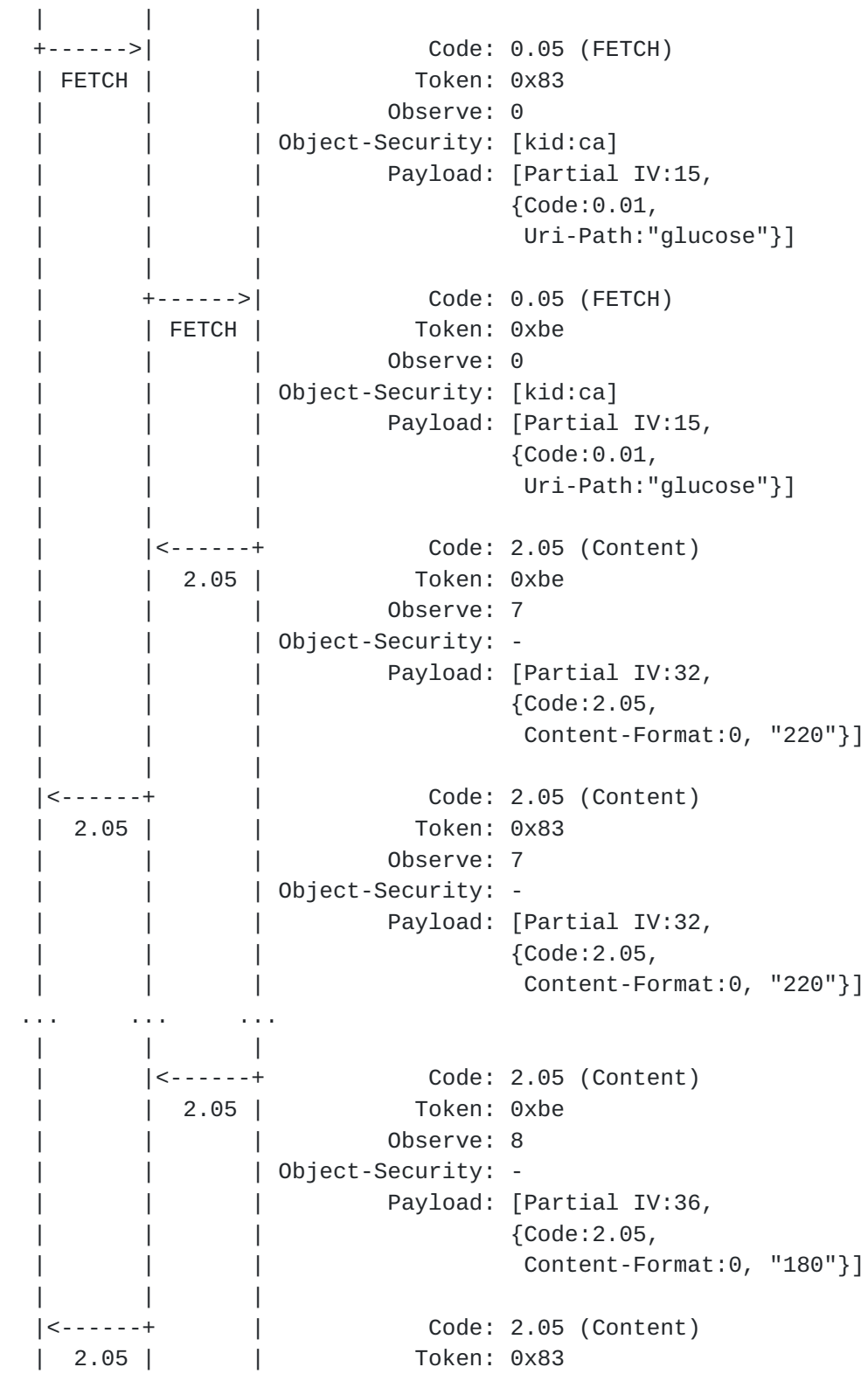
The COSE header of the request contains an identifier (5f), indicating which security context was used to protect the message and a Partial IV (42).

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

B.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.



```
      |       |       |         Observe: 8
      |       |       | Object-Security: -
      |       |       |         Payload: [Partial IV:36,
      |       |       |                  {Code:2.05,
      |       |       |                   Content-Format:0, "180"}]
      |       |       |
```

Figure 10: Secure Subscribe to Sensor. Square brackets [...] indicate a COSE object. Curly brackets { ... } indicate encrypted data.

The request/response Codes are encrypted by OSCORE and only dummy Codes (FETCH/Content) are visible in the header of the OSCORE message. The options Content-Format (0) and the payload ("220" and "180"), are encrypted.

The COSE header of the request contains an identifier (ca), indicating the security context used to protect the message and a Partial IV (15). The COSE headers of the responses contains Partial IVs (32 and 36).

The server verifies that the Partial IV has not been received before. The client verifies that the responses are bound to the request and that the Partial IVs are greater than any Partial IV previously received in a response bound to the request.

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