LAKE Working Group
Internet-Draft
Intended status: Standards Track
Expires: 5 September 2024

G. Selander
J. Preuß Mattsson
Ericsson AB
M. Vuini
G. Fedrecheski
INRIA
M. Richardson
Sandelman Software Works

4 March 2024

Lightweight Authorization using Ephemeral Diffie-Hellman Over COSE
draft-ietf-lake-authz-01

Abstract

This document describes a procedure for authorizing enrollment of new devices using the lightweight authenticated key exchange protocol Ephemeral Diffie-Hellman Over COSE (EDHOC). The procedure is applicable to zero-touch onboarding of new devices to a constrained network leveraging trust anchors installed at manufacture time.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://ericssonresearch.github.io/ace-ake-authz/draft-ietf-lake-authz.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-lake-authz/.

Discussion of this document takes place on the Lightweight Authenticated Key Exchange Working Group mailing list (mailto:lake@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/lake/. Subscribe at https://www.ietf.org/mailman/listinfo/lake/.

Source for this draft and an issue tracker can be found at https://github.com/EricssonResearch/ace-ake-authz.

Status of This Memo

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

This Internet-Draft will expire on 5 September 2024.

Table of Contents

1. Introduction

For constrained IoT deployments [RFC7228] the overhead and processing contributed by security protocols may be significant which motivates the specification of lightweight protocols that are optimizing, in particular, message overhead (see [I-D.ietf-lake-reqs]). This document describes a procedure for augmenting the lightweight authenticated Diffie-Hellman key exchange EDHOC [I-D.ietf-lake-edhoc] with third party-assisted authorization.

The procedure involves a device, a domain authenticator, and an enrollment server. The device and domain authenticator perform mutual authentication and authorization, assisted by the enrollment server which provides relevant authorization information to the device (a "voucher") and to the authenticator. The high-level model is similiar to BRSKI [RFC8995].

In this document we consider the target interaction for which authorization is needed to be "enrollment", for example joining a network for the first time (e.g., [RFC9031]), but it can be applied to authorize other target interactions.

The enrollment server may represent the manufacturer of the device, or some other party with information about the device from which a trust anchor has been pre-provisioned into the device. The (domain) authenticator may represent the service provider or some other party controlling access to the network in which the device is enrolling.

The protocol assumes that authentication between device and authenticator is performed with EDHOC [I-D.ietf-lake-edhoc], and defines the integration of a lightweight authorization procedure using the External Authorization Data (EAD) fields defined in EDHOC.

The protocol enables a low message count by performing authorization and enrollment in parallel with authentication, instead of in sequence which is common for network access. It further reuses protocol elements from EDHOC leading to reduced message sizes on constrained links.

This protocol is applicable to a wide variety of settings, and can be mapped to different authorization architectures.

1.1. Terminology

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

Readers are expected to have an understanding of CBOR [RFC8949] and EDHOC [I-D.ietf-lake-edhoc]. Appendix C.1 of [I-D.ietf-lake-edhoc] contains some basic info about CBOR.

2. Problem Description

The (potentially constrained) device (U) wants to enroll into a domain over a constrained link. The device authenticates and enforces authorization of the (non-constrained) domain authenticator (V) with the help of a voucher conveying authorization information. The voucher has a similar role as in [RFC8366] but should be considerably more compact. The domain authenticator, in turn, authenticates the device and authorizes its enrollment into the domain.

The procedure is assisted by a (non-constrained) enrollment server (W) located in a non-constrained network behind the domain authenticator, e.g. on the Internet, providing information to the device (the voucher) and to the domain authenticator as part of the protocol.

The objective of this document is to specify such a protocol which is lightweight over the constrained link by reusing elements of EDHOC [I-D.ietf-lake-edhoc] and by shifting message overhead to the non-constrained side of the network. See illustration in Figure 1.

Note the cardinality of the involved parties. It is expected that the authenticator needs to handle a large unspecified number of devices, but for a given device type or manufacturer it is expected that one or a few nodes host enrollment servers.

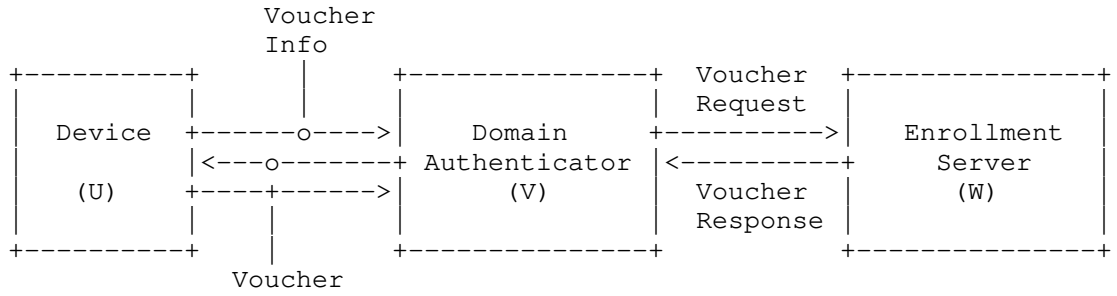


Figure 1: Overview of message flow. EDHOC is used on the constrained link between U and V. Voucher Info and Voucher are sent in EDHOC External Authorization Data (EAD). The link between V and W is not constrained.

3. Assumptions

The protocol is based on the following pre-existing relations between the device (U), the domain authenticator (V) and the enrollment server (W), see Figure 2.

* U and W have an explicit relation: U is configured with a public key of W, see Section 3.1.

* V and W have an implicit relation, e.g., based on web PKI with trusted CA certificates, see Section 3.2.

* U and V need not have any previous relation, this protocol establishes a relation between U and V.

Each of the three parties can gain protected communication with the other two during the protocol.

V may be able to access credentials over non-nonstrained networks, but U may be limited to constrained networks. Implementations wishing to leverage the zero-touch capabilities of this protocol are expected to support transmission of credentials from V to U by value during the EDHOC exchange, which will impact the message size depending on type of credential used.

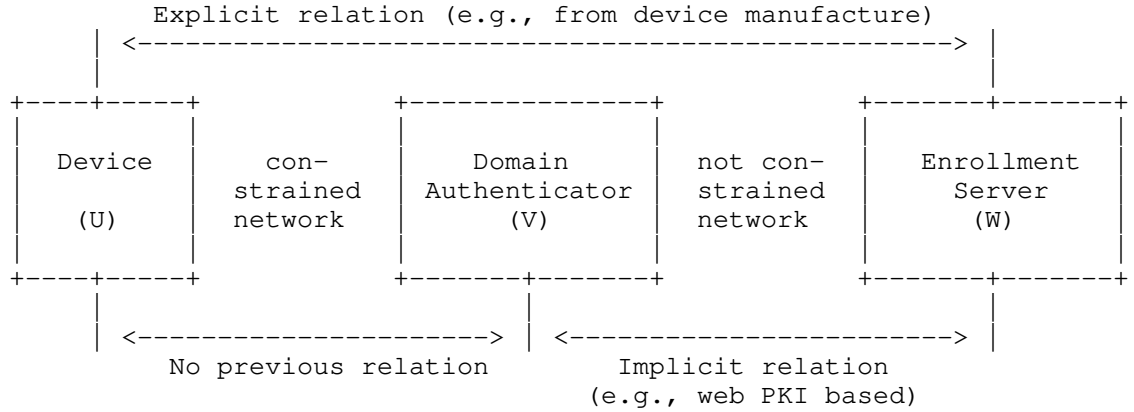


Figure 2: Overview of pre-existing relations.

3.1. Device (U)

To authenticate to V, the device (U) runs EDHOC in the role of Initiator with authentication credential CRED_U, for example, an X.509 certificate or a CBOR Web Token (CWT, [RFC8392]). CRED_U may, for example, be carried in ID_CRED_I of EDHOC message_3 or be provisioned to V over a non-constrained network, see bottom of Figure 3.

U also needs to identify itself to W, this device identifier is denoted by ID_U. The purpose of ID_U is for W to be able to determine if the device with this identifier is authorized to enroll with V. ID_U may be a reference to CRED_U, like ID_CRED_I in EDHOC (see Section 3.5.2 of [I-D.ietf-lake-edhoc]), or a device identifier from a different name space, such as EUI-64 identifiers.

U is also provisioned with information about W:

* A static public DH key of W (G_W) used to establish secure communication with the enrollment server (see Section 4.4).

* Location information about the enrollment server (LOC_W) that can be used by V to reach W. This is typically a URI but may alternatively be only the domain name.

3.2. Domain Authenticator (V)

To authenticate to U, the domain authenticator (V) runs EDHOC in the role of Responder with an authentication credential CRED_V, which is a CWT Claims Set [RFC8392] containing a public key of V, see Section 4.5.2.1. This proves to U the possession of the private key corresponding to the public key of CRED_V. CRED_V typically needs to be transported to U in EDHOC (using ID_CRED_R = CRED_V, see Section 3.5.2 of [I-D.ietf-lake-edhoc]) since there is no previous relation between U and V.

V and W need to establish a secure (confidentiality and integrity protected) connection for the Voucher Request/Response protocol. Furthermore, W needs access the same credential CRED_V as V used with U, and V needs to prove to W the possession of the private key corresponding to the public key of CRED_V. It is RECOMMENDED that V authenticates to W using the same credential CRED_V as with U.

* V and W may protect the Voucher Request/Response protocol using TLS 1.3 with client authentication [RFC8446] if CRED_V is an X.509 certificate of a signature public key. However, note that CRED_V may not be a valid credential to use with TLS 1.3, e.g., when U and V run EDHOC with method 1 or 3, where the public key of CRED_V is a static Diffie-Hellman key.

* V may run EDHOC with W using ID_CRED_I = CRED_V. In this case the secure connection between V and W may be based on OSCORE [RFC8613].

Note that both TLS 1.3 and EDHOC may be run between V and W during this setup procedure. For example, W may authenticate to V using TLS 1.3 with server certificates signed by a CA trusted by V, and then V may run EDHOC using CRED_V over the secure TLS connection to W, see Figure 3.

Note also that the secure connection between V and W may be long lived and reused for multiple voucher requests/responses.

Other details of proof-of-possession related to CRED_V and transport of CRED_V are out of scope of this document.

3.3. Enrollment Server (W)

The enrollment server (W) is assumed to have the private DH key corresponding to G_W, which is used to establish secure communication with the device (see Section 4.4). W provides to U the authorization decision for enrollment with V in the form of a voucher (see Section 4.4.2). Authorization policies are out of scope for this document.

Authentication credentials and communication security with V is described in Section 3.2. To calculate the voucher, W needs access to message_1 and CRED_V as used in the EDHOC session between U and V, see Section 4.4.2.

* W MUST verify that CRED_V is bound to the secure connection between W and V

* W MUST verify that V is in possession of the private key corresponding to the public key of CRED_V

W needs to be available during the execution of the protocol between U and V.

4. The Protocol

4.1. Overview

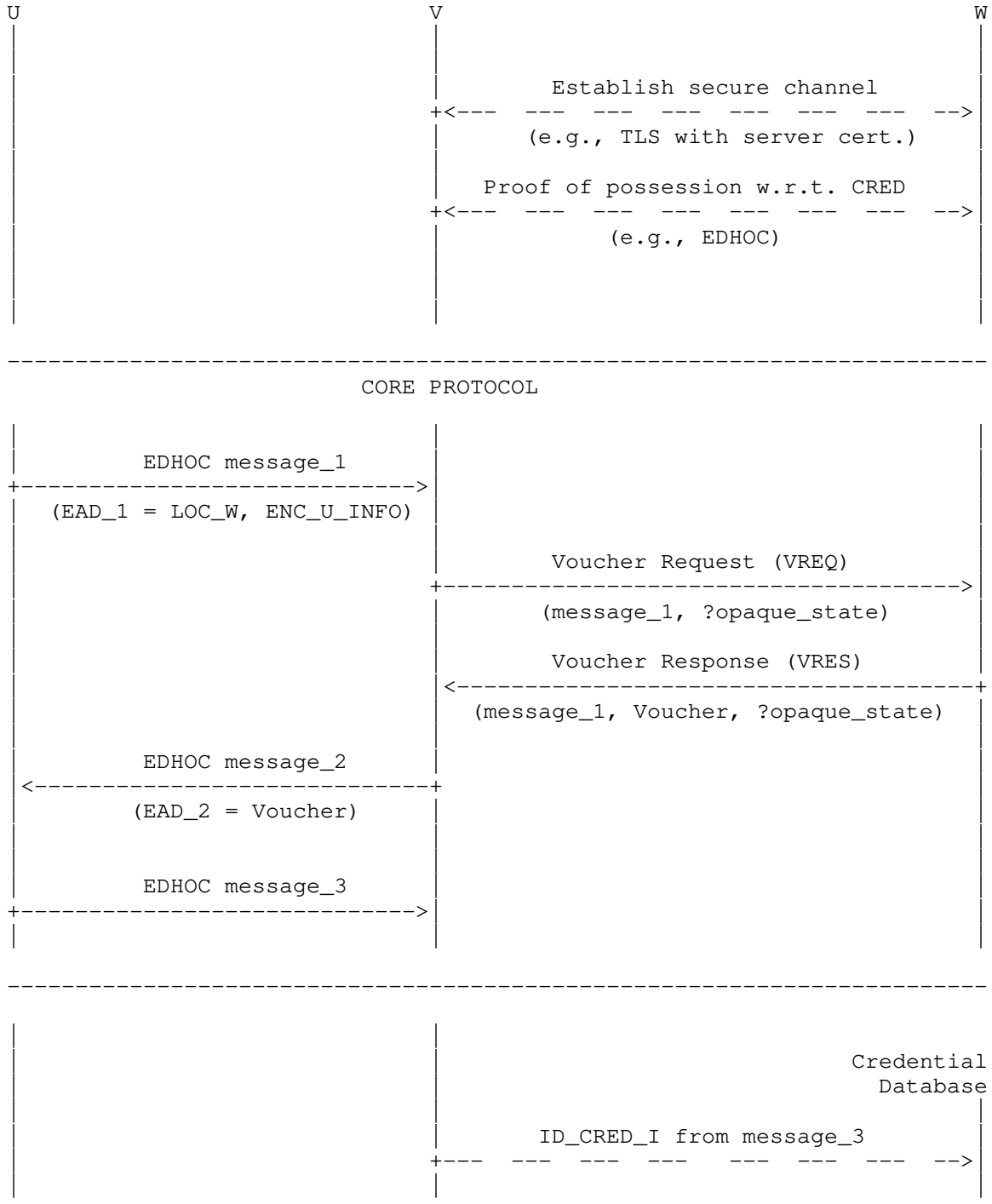
The protocol consist of three security sessions going on in parallel:

1. The EDHOC session between device (U) and (domain) authenticator (V)

2. Voucher Request/Response between authenticator (V) and enrollment server (W)

3. An exchange of voucher-related information, including the voucher itself, between device (U) and enrollment server (W), mediated by the authenticator (V).

Figure 3 provides an overview of the message flow detailed in this section. An outline of EDHOC is given in Section 3 of [I-D.ietf-lake-edhoc].


U
V
W
Establish secure channel
(e.g., TLS with server cert.)
Proof of possession w.r.t. CRED
(e.g., EDHOC)
CORE PROTOCOL
EDHOC message_1
(EAD_1 = LOC_W, ENC_U_INFO)
Voucher Request (VREQ)
(message_1, ?opaque_state)
Voucher Response (VRES)
(message_1, Voucher, ?opaque_state)
EDHOC message_2
(EAD_2 = Voucher)
EDHOC message_3
Credential
Database
ID_CRED_I from message_3

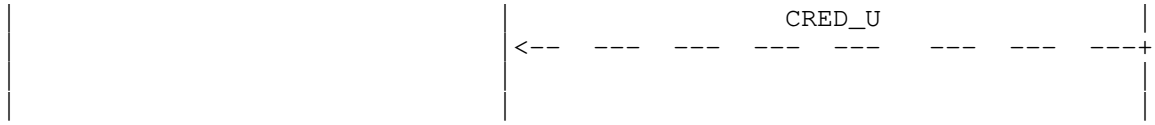


Figure 3: Overview of the protocol: W-assisted authorization of U and V to each other: EDHOC between U and V, and Voucher Request/ Response between V and W. Before the protocol, V and W are assumed to have established a secure channel and performed proof-of-possession of relevant keys. Credential lookup of CRED_U may involve W or other credential database.

4.2. Reuse of EDHOC

The protocol illustrated in Figure 3 reuses several components of EDHOC:

* G_X, the ephemeral public Diffie-Hellman key of U, is also used in the protocol between U and W.

* SUITES_I includes the cipher suite for EDHOC selected by U, and also defines the algorithms used between U and W (see Section 3.6 of [I-D.ietf-lake-edhoc]):

  - EDHOC AEAD algorithm: used to encrypt ID_U

  - EDHOC hash algorithm: used for key derivation and to calculate the voucher

  - EDHOC MAC length in bytes: length of the voucher

  - EDHOC key exchange algorithm: used to calculate the shared secret between U and W

* EAD_1, EAD_2 are the External Authorization Data message fields of message_1 and message_2, respectively, see Section 3.8 of [I-D.ietf-lake-edhoc]. This document specifies the EAD items with ead_label = TBD1, see Section 7.1).

* ID_CRED_I and ID_CRED_R are used to identify the authentication credentials CRED_U and CRED_V, respectively. As shown at the bottom of Figure 3, V may use W to obtain CRED_U. CRED_V is transported in ID_CRED_R in message_2, see Section 4.5.2.1.

The protocol also reuses the EDHOC-Extract and EDHOC-Expand key derivation from EDHOC (see Section 4 of [I-D.ietf-lake-edhoc]).

* The intermediate pseudo-random key PRK is derived using EDHOC-Extract():

  - PRK = EDHOC-Extract(salt, IKM)

    o where salt = 0x (the zero-length byte string)

    o IKM is computed as an ECDH cofactor Diffie-Hellman shared secret from the public key of W, G_W, and the private key corresponding to G_X (or v.v.), see Section 5.7.1.2 of [NIST-800-56A].

The output keying material OKM is derived from PRK using EDHOC-Expand(), which is defined in terms of the EDHOC hash algorithm of the selected cipher suite, see Section 4.2 of [I-D.ietf-lake-edhoc]:

* OKM = EDHOC-Expand(PRK, info, length)

  where

```
info = (
   info_label : int,
   context : bstr,
   length : uint,
)
```

4.3. Stateless Operation of V

V may act statelessly with respect to U: the state of the EDHOC session started by U may be dropped at V until authorization from W is received. Once V has received EDHOC message_1 from U and extracted LOC_W from EAD_1, message_1 is forwarded unmodified to W in the form of a Voucher Request. V encapsulates the internal state that it needs to later respond to U, and sends that to W together with EDHOC message_1. This state typically contains U's IP address and port number, together with any other implementation-specific parameter needed by V to respond to U. At this point, V can drop the EDHOC session that was initiated by U.

V MUST encrypt and integrity protect the encapsulated state using a uniformly-distributed (pseudo-)random key, known only to itself. How V serializes and encrypts its internal state is out of scope of this specification. For example, V may use the existing CBOR and COSE libraries.

Editor's note: Consider to include an example of serialized internal state.

W sends to V the voucher together with echoed message_1, as received from U, and V's internal state. This allows V to act as a simple message relay until it has obtained the authorization from W to enroll U. The reception of a successful Voucher Response at V from W implies the authorization for V to enroll U. At this point, V can initialize a new EDHOC session with U, based on the message and the state retrieved from the Voucher Response from W.

4.4. Device <-> Enrollment Server (U <-> W)

The protocol between U and W is carried between U and V in message_1 and message_2 (Section 4.5), and between V and W in the Voucher Request/Response (Section 4.6). The data is protected between the endpoints using secret keys derived from a Diffie-Hellman shared secret (see Section 4.2) as further detailed in this section.

4.4.1. Voucher Info

The external authorization data EAD_1 contains an EAD item with ead_label = TBD1 and ead_value = Voucher_Info, which is a CBOR byte string:

```
Voucher_Info = bstr .cbor Voucher_Info_Seq

Voucher_Info_Seq = (
    LOC_W:      tstr,
    ENC_U_INFO:     bstr
)
```

where

* LOC_W is a text string used by V to locate W, e.g., a URI or a domain name.

* ENC_U_INFO is a byte string containing an encrypted identifier of U and, optionally, opaque application data prepared by U. It is calculated as follows:

ENC_U_INFO is 'ciphertext' of COSE_Encrypt0 (Section 5.2 of [RFC9052]) computed from the following:

* The encryption key K_1 and nonce IV_1 are derived as specified below.

* 'protected' is a byte string of size 0

* 'plaintext' and 'external_aad' as below:

```
plaintext = (
    ID_U:            bstr,
    ?OPAQUE_INFO:    bstr,
)

external_aad = (
    SS:              int,
)
```

where

* ID_U is an identifier of the device, see Section 3.1.

* OPAQUE_INFO is an opaque field provided by the application. If present, it will contain application data that U may want to convey to W, e.g., enrollment hints, see Appendix B. Note that OPAQUE_INFO is opaque when viewed as an information element in EDHOC. It is opaque to V, while the application in U and W can read its contents. The same applies to other references of OPAQUE_INFO throughout this document.

* SS is the selected cipher suite in SUITES_I of EDHOC message_1, see Section 4.5.

The external_aad is wrapped in an enc_structure as defined in Section 5.3 of [RFC9052].

Editor's note: Add more context to external_aad.

The derivation of K_1 = EDHOC-Expand(PRK, info, length) uses the following input to the info struct (see OKM in Section 4.2):

* info_label = 0

* context = h'' (the empty CBOR string)

* length is length of key of the EDHOC AEAD algorithm in bytes (which is the length of K_1)

The derivation of IV_1 = EDHOC-Expand(PRK, info, length) uses the following input to the info struct (see OKM in Section 4.2):

* info_label = 1

* context = h'' (the empty CBOR string)

* length is length of nonce of the EDHOC AEAD algorithm in bytes (which is the length of IV_1)

4.4.2. Voucher

The voucher is an assertion to U that W has authorized V. The voucher consists of the 'ciphertext' field of a COSE_Encrypt0 object:

Voucher = COSE_Encrypt0.ciphertext

Its corresponding plaintext value consists of an opaque field that can be used by W to convey information to U, such as a voucher scope. The authentication tag present in the ciphertext is also bound to message_1 and the credential of V.

The external authorization data EAD_2 contains an EAD item with ead_label = TBD1 and ead_value = Voucher, which is computed from the following:

* The encryption key K_2 and nonce IV_2 are derived as specified below.

* 'protected' is a byte string of size 0

* 'plaintext' and 'external_aad' as below:

```
plaintext = (
    ?OPAQUE_INFO: bstr
)

external_aad = (
    H(message_1):  bstr,
    CRED_V:        bstr,
)
```

where

* OPAQUE_INFO is an opaque field provided by the application.

* H(message_1) is the hash of EDHOC message_1, calculated from the associated voucher request, see Section 4.6.1.

* CRED_V is the CWT Claims Set [RFC8392] containing the public authentication key of V, see Section 4.5.2.1

The derivation of K_2 = EDHOC-Expand(PRK, info, length) uses the following input to the info struct (see Section 4.2):

* info_label = 2

* context = h'' (the empty CBOR string)

* length is length of key of the EDHOC AEAD algorithm in bytes

The derivation of IV_2 = EDHOC-Expand(PRK, info, length) uses the following input to the info struct (see Section 4.2):

* info_label = 3

* context = h'' (the empty CBOR string)

* length is length of nonce of the EDHOC AEAD algorithm in bytes

4.5. Device <-> Authenticator (U <-> V)

This section describes the processing in U and V, which include the EDHOC protocol, see Figure 3. Normal EDHOC processing is omitted here.

4.5.1. Message 1

4.5.1.1. Processing in U

U composes EDHOC message_1 using authentication method, identifiers, etc. according to an agreed application profile, see Section 3.9 of [I-D.ietf-lake-edhoc]. The selected cipher suite, in this document denoted SS, applies also to the interaction with W as detailed in Section 4.2, in particular, with respect to the Diffie Hellman key agreement algorithm used between U and W. As part of the normal EDHOC processing, U generates the ephemeral public key G_X which is reused in the interaction with W, see Section 4.4.

The device sends EDHOC message_1 with EAD item (-TBD1, Voucher_Info) included in EAD_1, where Voucher_Info is specified in Section 4.4. The negative sign indicates that the EAD item is critical, see Section 3.8 of [I-D.ietf-lake-edhoc].

4.5.1.2. Processing in V

V receives EDHOC message_1 from U and processes it as specified in Section 5.2.3 of [I-D.ietf-lake-edhoc], with the additional step of processing the EAD item in EAD_1. Since the EAD item is critical, if V does not recognize it or it contains information that V cannot process, then V MUST abort the EDHOC session, see Section 3.8 of [I-D.ietf-lake-edhoc]. Otherwise, the ead_label = TBD1, triggers the voucher request to W as described in Section 4.6. The exchange between V and W needs to be completed successfully for the EDHOC session to be continued.

4.5.2. Message 2

4.5.2.1. Processing in V

V receives the voucher response from W as described in Section 4.6.

V sends EDHOC message_2 to U with the critical EAD item (-TBD1, Voucher) included in EAD_2, where the Voucher is specified in Section 4.4.

CRED_V is a CWT Claims Set [RFC8392] containing the public authentication key of V encoded as a COSE_Key in the 'cnf' claim, see Section 3.5.2 of [I-D.ietf-lake-edhoc].

ID_CRED_R contains the CWT Claims Set with 'kccs' as COSE header_map, see Section 9.6 of [I-D.ietf-lake-edhoc].

4.5.2.2. Processing in U

U receives EDHOC message_2 from V and processes it as specified in Section 5.3.2 of [I-D.ietf-lake-edhoc], with the additional step of processing the EAD item in EAD_2.

If U does not recognize the EAD item or the EAD item contains information that U cannot process, then U MUST abort the EDHOC session, see Section 3.8 of [I-D.ietf-lake-edhoc]. Otherwise U MUST verify the Voucher by performing the same calculation as in Section 4.4.2 using H(message_1) and CRED_V received in ID_CRED_R of message_2. If the voucher calculated in this way is not identical to what was received in message_2, then U MUST abort the EDHOC session.

4.5.3. Message 3

4.5.3.1. Processing in U

If all verifications are passed, then U sends EDHOC message_3.

EDHOC message_3 may be combined with an OSCORE request, see [I-D.ietf-core-oscore-edhoc].

4.5.3.2. Processing in V

V performs the normal EDHOC verifications of message_3. V may retrieve CRED_U from a Credential Database, after having learnt ID_CRED_I from U.

4.6. Authenticator <-> Enrollment Server (V <-> W)

It is assumed that V and W have set up a secure connection, W has accessed the authentication credential CRED_V to be used in the EDHOC session between V and with U, and that W has verified that V is in possession of the private key corresponding to CRED_V, see Section 3.2 and Section 3.3. V and W run the Voucher Request/ Response protocol over the secure connection.

4.6.1. Voucher Request

4.6.1.1. Processing in V

V sends the voucher request to W. The Voucher Request SHALL be a CBOR array as defined below:

```
Voucher_Request = [
    message_1:      bstr,
    ? opaque_state: bstr
]
```

where

* message_1 is the EDHOC message_1 as it was received from U.

* opaque_state is OPTIONAL and represents the serialized and encrypted opaque state needed by V to statelessly respond to U after the reception of Voucher_Response.

4.6.1.2. Processing in W

W receives and parses the voucher request received over the secure connection with V. The voucher request essentially contains EDHOC message_1 as sent by U to V. W SHALL NOT process message_1 as if it was an EDHOC message intended for W.

W extracts from message_1:

* SS - the selected cipher suite, which is the (last) integer of SUITES_I.

* G_X - the ephemeral public key of U

* ENC_U_INFO - the encryption of (1) the device identifier ID_U and (2) the optional OPAQUE_INFO field, contained in the Voucher_Info field of the EAD item with ead_label = TBD1 (with minus sign indicating criticality)

W verifies and decrypts ENC_U_INFO using the relevant algorithms of the selected cipher suite SS (see Section 4.2), and obtains ID_U.

In case OPAQUE_INFO is present, it is made available to the application.

W calculates the hash of message_1 H(message_1), and associates this session identifier to the device identifier ID_U. If H(message_1) is not unique among session identifiers associated to this device identifier of U, the EDHOC session SHALL be aborted.

W uses ID_U to look up the associated authorization policies for U and enforces them. This is out of scope for the specification.

4.6.2. Voucher Response

4.6.2.1. Processing in W

W retrieves CRED_V associated to the secure connection with V, and constructs the the Voucher for the device with identifier ID_U (see Section 4.4.2).

W generates the voucher response and sends it to V over the secure connection. The Voucher_Response SHALL be a CBOR array as defined below:

```
Voucher_Response = [
    message_1:      bstr,
    Voucher:        bstr,
    ? opaque_state: bstr
]
```

where

* message_1 is the EDHOC message_1 as it was received from V.

* The Voucher is defined in Section 4.4.2.

* opaque_state is the echoed byte string opaque_state from Voucher_Request, if present.

4.6.2.2. Processing in V

V receives the voucher response from W over the secure connection. If present, V decrypts and verifies opaque_state as received from W. If that verification fails then EDHOC is aborted. If the voucher response is successfully received from W, then V responds to U with EDHOC message_2 as described in Section 4.5.2.1.

4.7. Error Handling

This section specifies a new EDHOC error code and how it is used in the proposed protocol.

4.7.1. EDHOC Error "Access denied"

This section specifies the new EDHOC error "Access denied", see Figure 4.

ERR_CODE	ERR_INFO Type	Description
TBD3	error_content	Access denied

Figure 4: EDHOC error code and error information for Access denied.

Error code TBD3 is used to indicate to the receiver that access control has been applied and the sender has aborted the EDHOC session. The ERR_INFO field contains error_content which is a CBOR Sequence consisting of an integer and an optional byte string.

```
error_content = (
  REJECT_TYPE : int,
  ? REJECT_INFO : bstr,
)
```

The purpose of REJECT_INFO is for the sender to provide verifiable and actionable information to the receiver about the error, so that an automated action may be taken to enable access.

REJECT_TYPE	REJECT_INFO	Description
0	-	No REJECT_INFO
1	bstr	REJECT_INFO from trusted third party

Figure 5: REJECT_TYPE and REJECT_INFO for Access denied.

4.7.2. Error handling in W, V, and U

This protocol uses the EDHOC Error "Access denied" in the following way:

* W generates error_content and transfers it to V via the secure connection. If REJECT_TYPE is 1, then REJECT_INFO is encrypted from W to U using the EDHOC AEAD algorithm.

* V receives error_content, prepares an EDHOC "Access denied" error, and sends it to U.

* U receives the error message and extracts the error_content. If REJECT_TYPE is 1, then U decrypts REJECT_INFO, based on which it may retry to gain access.

The encryption of REJECT_INFO follows a procedure analogous to the one defined in Section 4.4.2, with the following differences:

```
plaintext = (
    OPAQUE_INFO:     bstr,
 )

external_aad = (
    H(message_1):    bstr,
 )
```

where

* OPAQUE_INFO is an opaque field that contains actionable information about the error. It may contain, for example, a list of suggested Vs through which U should join instead.

* H(message_1) is the hash of EDHOC message_1, calculated from the associated voucher request, see Section 4.6.1.

5. REST Interface at W

The interaction between V and W is enabled through a RESTful interface exposed by W. This RESTful interface MAY be implemented using either HTTP or CoAP. V SHOULD access the resources exposed by W through the protocol indicated by the scheme in LOC_W URI.

5.1. Scheme "https"

In case the scheme indicates "https", V MUST perform a TLS handshake with W and use HTTP. If the authentication credential CRED_V can be used in a TLS handshake, e.g. an X.509 certificate of a signature public key, then V SHOULD use it to authenticate to W as a client. If the authentication credential CRED_V cannot be used in a TLS handshake, e.g. if the public key is a static Diffie-Hellman key, then V SHOULD first perform a TLS handshake with W using available compatible keys. V MUST then perform an EDHOC session over the TLS

connection proving to W the possession of the private key corresponding to CRED_V. Performing the EDHOC session is only necessary if V did not authenticate with CRED_V in the TLS handshake with W.

Editor's note: Clarify that performing TLS handshake is not necessary for each device request; if there already is a TLS connection between V and W that should be reused. Similar considerations for 5.2 and 5.3.

5.2. Scheme "coaps"

In case the scheme indicates "coaps", V SHOULD perform a DTLS handshake with W and access the resources defined in Section 5.4 using CoAP. The normative requirements in Section 5.1 on performing the DTLS handshake and EDHOC session remain the same, except that TLS is replaced with DTLS.

5.3. Scheme "coap"

In case the scheme indicates "coap", V SHOULD perform an EDHOC session with W, as specified in Appendix A of [I-D.ietf-lake-edhoc] and access the resources defined in Section 5.4 using OSCORE and CoAP. The authentication credential in this EDHOC session MUST be CRED_V.

5.4. URIs

The URIs defined below are valid for both HTTP and CoAP. W MUST support the use of the path-prefix "/.well-known/", as defined in [RFC8615], and the registered name "lake-authz". A valid URI in case of HTTP thus begins with

* "https://www.example.com/.well-known/lake-authz"

In case of CoAP with DTLS:

* "coaps://example.com/.well-known/lake-authz"

In case of EDHOC and OSCORE:

* "coap://example.com/.well-known/lake-authz"

Each operation specified in the following is indicated by a path-suffix.

5.4.1. Voucher Request (/voucherrequest)

To request a voucher, V MUST issue a request:

* Method is POST
* Payload is the serialization of the Voucher Request object, as specified in Section 4.6.1.
* Content-Format (Content-Type) is set to "application/lake-authz-voucherrequest+cbor"

In case of successful processing at W, W MUST issue a 200 OK response with payload containing the serialized Voucher Response object, as specified in Section 4.6.2.

5.4.2. Certificate Request (/certrequest)

V requests the public key certificate of U from W through the "/certrequest" path-suffix. To request U's authentication credential, V MUST issue a request:

* Method is POST
* Payload is the serialization of the ID_CRED_I object, as received in EDHOC message_3.

In case of a successful lookup of the authentication credential at W, W MUST issue 200 OK response with payload containing the serialized CRED_U.

6. Security Considerations

This specification builds on and reuses many of the security constructions of EDHOC, e.g., shared secret calculation and key derivation. The security considerations of EDHOC [I-D.ietf-lake-edhoc] apply with modifications discussed here.

EDHOC provides identity protection of the Initiator, here the device. The encryption of the device identifier ID_U in the first message should consider potential information leaking from the length of ID_U, either by making all identifiers having the same length or the use of a padding scheme.

Although W learns about the identity of U after receiving VREQ, this information must not be disclosed to V, until U has revealed its identity to V with ID_CRED_I in message_3. W may be used for lookup of CRED_U from ID_CRED_I, or this credential lookup function may be

separate from the authorization function of W, see Figure 3. The trust model used here is that U decides to which V it reveals its identity. In an alternative trust model where U trusts W to decide to which V it reveals U's identity, CRED_U could be sent in Voucher Response.

As noted in Section 8.2 of [I-D.ietf-lake-edhoc] an ephemeral key may be used to calculate several ECDH shared secrets. In this specification the ephemeral key G_X is also used to calculate G_XW, the shared secret with the enrollment server.

The private ephemeral key is thus used in the device for calculations of key material relating to both the authenticator and the enrollment server. There are different options for where to implement these calculations, one option is as an addition to EDHOC, i.e., to extend the EDHOC API in the device with input of public key of W (G_W) and device identifier of U (ID_U), and produce the encryption of ID_U which is included in Voucher_Info in EAD_1.

7. IANA Considerations

7.1. EDHOC External Authorization Data Registry

IANA has registered the following entry in the "EDHOC External Authorization Data" registry under the group name "Ephemeral Diffie-Hellman Over COSE (EDHOC)". The ead_label = TBD1 corresponds to the ead_value Voucher_Info in EAD_1, and Voucher in EAD_2 with processing specified in Section 4.5.1 and Section 4.5.2, respectively, of this document.

Label	Value Type	Description
TBD1	bstr	Voucher related information

Table 1: Addition to the EDHOC EAD registry

7.2. The Well-Known URI Registry

IANA has registered the following entry in "The Well-Known URI Registry", using the template from [RFC8615]:

* URI suffix: lake-authz

* Change controller: IETF

* Specification document: [[this document]]

* Related information: None

7.3. Well-Known Name Under ".arpa" Name Space

This document allocates a well-known name under the .arpa name space according to the rules given in [RFC3172] and [RFC6761]. The name "lake-authz.arpa" is requested. No subdomains are expected, and addition of any such subdomains requires the publication of an IETF Standards Track RFC. No A, AAAA, or PTR record is requested.

7.4. Media Types Registry

IANA has added the media types "application/lake-authz-voucherrequest+cbor" to the "Media Types" registry.

7.4.1. application/lake-authz-voucherrequest+cbor Media Type Registration

* Type name: application
* Subtype name: lake-authz-voucherrequest+cbor
* Required parameters: N/A
* Optional paramaters: N/A
* Encoding considerations: binary
* Security cosniderations: See Section 6 of this document.
* Interoperability considerations: N/A
* Published specification: [[this document]] (this document)
* Application 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: See "Authors' Addresses" section.

* Intended usage: COMMON

* Restrictions on usage: N/A

* Author: See "Authors' Addresses" section.

* Change Controller: IESG

7.5. CoAP Content-Formats Registry

IANA has added the media type "application/lake-authz-voucherrequest+cbor" to the "CoAP Content-Formats" registry under the registry group "Constrained RESTful Environments (CoRE) Parameters".

Media Type	Encoding	ID	Reference
application/lake-authz-voucherrequest+cbor	-	TBD2	[[this document]]

Table 2: Addition to the CoAP Content-Formats registry

8. References

8.1. Normative References

[I-D.ietf-lake-edhoc]
Selander, G., Mattsson, J. P., and F. Palombini, "Ephemeral Diffie-Hellman Over COSE (EDHOC)", Work in Progress, Internet-Draft, draft-ietf-lake-edhoc-23, 22 January 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-lake-edhoc-23>.

[NIST-800-56A]
Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography - NIST Special Publication 800-56A, Revision 3", April 2018, <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar3.pdf>.

[RFC8366] Watsen, K., Richardson, M., Pritikin, M., and T. Eckert, "A Voucher Artifact for Bootstrapping Protocols", RFC 8366, DOI 10.17487/RFC8366, May 2018, <https://www.rfc-editor.org/info/rfc8366>.

[RFC8392] Jones, M., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "CBOR Web Token (CWT)", RFC 8392, DOI 10.17487/RFC8392, May 2018, <https://www.rfc-editor.org/info/rfc8392>.

[RFC8613] Selander, G., Mattsson, J., Palombini, F., and L. Seitz, "Object Security for Constrained RESTful Environments (OSCORE)", RFC 8613, DOI 10.17487/RFC8613, July 2019, <https://www.rfc-editor.org/info/rfc8613>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/info/rfc8949>.

[RFC9052] Schaad, J., "CBOR Object Signing and Encryption (COSE): Structures and Process", STD 96, RFC 9052, DOI 10.17487/RFC9052, August 2022, <https://www.rfc-editor.org/info/rfc9052>.

8.2. Informative References

[I-D.ietf-core-oscore-edhoc]
Palombini, F., Tiloca, M., Höglund, R., Hristozov, S., and G. Selander, "Using Ephemeral Diffie-Hellman Over COSE (EDHOC) with the Constrained Application Protocol (CoAP) and Object Security for Constrained RESTful Environments (OSCORE)", Work in Progress, Internet-Draft, draft-ietf-core-oscore-edhoc-10, 29 November 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-core-oscore-edhoc-10>.

[I-D.ietf-lake-reqs]
Vuini, M., Selander, G., Mattsson, J. P., and D. Garcia-Carillo, "Requirements for a Lightweight AKE for OSCORE", Work in Progress, Internet-Draft, draft-ietf-lake-reqs-04, 8 June 2020, <https://datatracker.ietf.org/doc/html/draft-ietf-lake-reqs-04>.

[IEEE802.15.4]
IEEE standard for Information Technology, "IEEE Std 802.15.4 Standard for Low-Rate Wireless Networks", n.d..

[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>.

[RFC3172] Huston, G., Ed., "Management Guidelines & Operational Requirements for the Address and Routing Parameter Area Domain ("arpa")", BCP 52, RFC 3172, DOI 10.17487/RFC3172, September 2001, <https://www.rfc-editor.org/info/rfc3172>.

[RFC6761] Cheshire, S. and M. Krochmal, "Special-Use Domain Names", RFC 6761, DOI 10.17487/RFC6761, February 2013, <https://www.rfc-editor.org/info/rfc6761>.

[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>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC8615] Nottingham, M., "Well-Known Uniform Resource Identifiers (URIs)", RFC 8615, DOI 10.17487/RFC8615, May 2019, <https://www.rfc-editor.org/info/rfc8615>.

[RFC8995] Pritikin, M., Richardson, M., Eckert, T., Behringer, M., and K. Watsen, "Bootstrapping Remote Secure Key Infrastructure (BRSKI)", RFC 8995, DOI 10.17487/RFC8995, May 2021, <https://www.rfc-editor.org/info/rfc8995>.

[RFC9031] Vuini, M., Ed., Simon, J., Pister, K., and M. Richardson, "Constrained Join Protocol (CoJP) for 6TiSCH", RFC 9031, DOI 10.17487/RFC9031, May 2021, <https://www.rfc-editor.org/info/rfc9031>.

Appendix A. Use with Constrained Join Protocol (CoJP)

This section outlines how the protocol is used for network enrollment and parameter provisioning. An IEEE 802.15.4 network is used as an example of how a new device (U) can be enrolled into the domain managed by the domain authenticator (V).

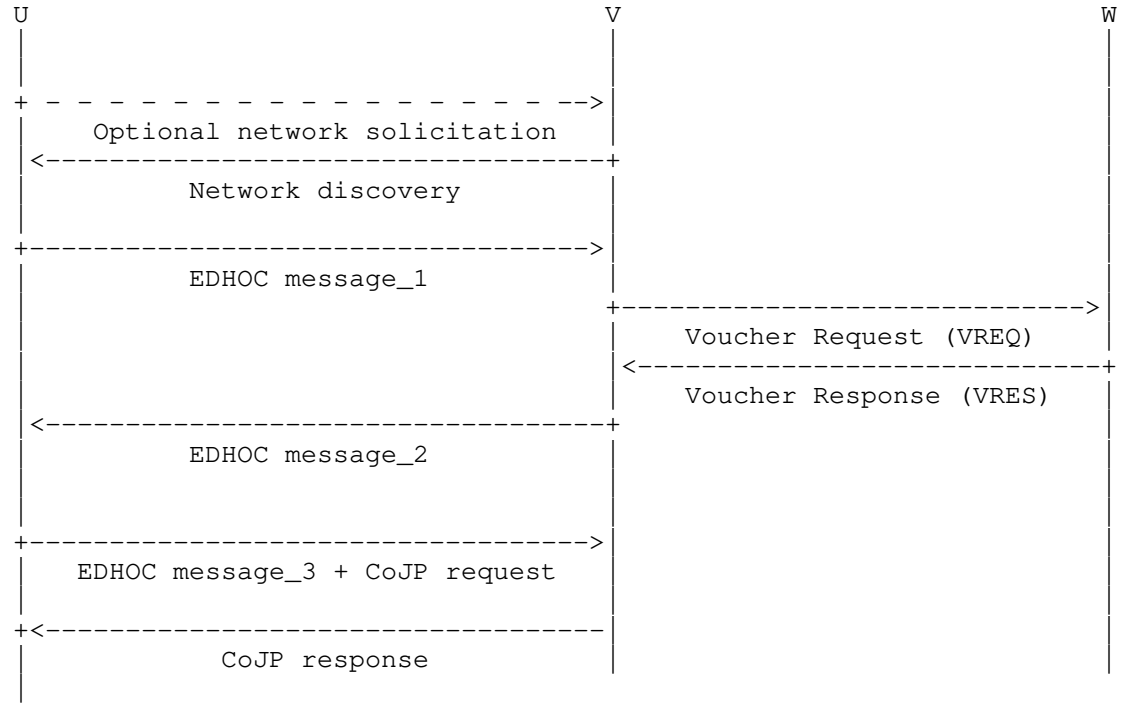


Figure 6: Use of draft-ietf-lake-authz with CoJP.

A.1. Network Discovery

When a device first boots, it needs to discover the network it attempts to join. The network discovery procedure is defined by the link-layer technology in use. In case of Time-slotted Channel Hopping (TSCH) networks, a mode of [IEEE802.15.4], the device scans the radio channels for Enhanced Beacon (EB) frames, a procedure known as passive scan. EBs carry the information about the network, and particularly the network identifier. Based on the EB, the network identifier, the information pre-configured into the device, the device makes the decision on whether it should join the network advertised by the received EB frame. This process is described in Section 4.1 of [RFC9031]. In case of other, non-TSCH modes of IEEE 802.15.4 it is possible to use the active scan procedure and send solicitation frames. These solicitation frames trigger the nearest network coordinator to respond by emitting a beacon frame. The network coordinator emitting beacons may be multiple link-layer hops away from the domain authenticator (V), in which case it plays the role of a Join Proxy (see [RFC9031]). Join Proxy does not participate in the protocol and acts as a transparent router between the device and the domain authenticator. For simplicity, Figure 6

illustrates the case when the device and the domain authenticator are a single hop away and can communicate directly.

A.2. The Enrollment Protocol with Parameter Provisioning

A.2.1. Flight 1

Once the device has discovered the network it wants to join, it constructs the EDHOC message_1, as described in Section 4.5. The device SHALL map the message to a CoAP request:

* The request method is POST.

* The type is Confirmable (CON).

* The Proxy-Scheme option is set to "coap".

* The Uri-Host option is set to "lake-authz.arpa". This is an anycast type of identifier of the domain authenticator (V) that is resolved to its IPv6 address by the Join Proxy.

* The Uri-Path option is set to ".well-known/edhoc".

* The payload is the (true, EDHOC message_1) CBOR sequence, where EDHOC message_1 is constructed as defined in Section 4.5.

A.2.2. Flight 2

The domain authenticator receives message_1 and processes it as described in Section 4.5. The message triggers the exchange with the enrollment server, as described in Section 4.6. If the exchange between V and W completes successfully, the domain authenticator prepares EDHOC message_2, as described in Section 4.5. The authenticator SHALL map the message to a CoAP response:

* The response code is 2.04 Changed.

* The payload is the EDHOC message_2, as defined in Section 4.5.

A.2.3. Flight 3

The device receives EDHOC message_2 and processes it as described in Section 4.5}. Upon successful processing of message_2, the device prepares flight 3, which is an OSCORE-protected CoJP request containing an EDHOC message_3, as described in [I-D.ietf-core-oscore-edhoc]. EDHOC message_3 is prepared as described in Section 4.5. The OSCORE-protected payload is the CoJP Join Request object specified in Section 8.4.1 of [RFC9031]. OSCORE

protection leverages the OSCORE Security Context derived from the EDHOC session, as specified in Appendix A of [I-D.ietf-lake-edhoc]. To that end, [I-D.ietf-core-oscore-edhoc] specifies that the Sender ID of the client (device) must be set to the connection identifier selected by the domain authenticator, C_R. OSCORE includes the Sender ID as the kid in the OSCORE option. The network identifier in the CoJP Join Request object is set to the network identifier obtained from the network discovery phase. In case of IEEE 802.15.4 networks, this is the PAN ID.

The device SHALL map the message to a CoAP request:

* The request method is POST.

* The type is Confirmable (CON).

* The Proxy-Scheme option is set to "coap".

* The Uri-Host option is set to "lake-authz.arpa".

* The Uri-Path option is set to ".well-known/edhoc".

* The EDHOC option [I-D.ietf-core-oscore-edhoc] is set and is empty.

* The payload is prepared as described in Section 3.2 of [I-D.ietf-core-oscore-edhoc], with EDHOC message_3 and the CoJP Join Request object as the OSCORE-protected payload.

Note that the OSCORE Sender IDs are derived from the connection identifiers of the EDHOC session. This is in contrast with [RFC9031] where ID Context of the OSCORE Security Context is set to the device identifier (pledge identifier). Since the device identity is exchanged during the EDHOC session, and the certificate of the device is communicated to the authenticator as part of the Voucher Response message, there is no need to transport the device identity in OSCORE messages. The authenticator playing the role of the [RFC9031] JRC obtains the device identity from the execution of the authorization protocol.

A.2.4. Flight 4

Flight 4 is the OSCORE response carrying CoJP response message. The message is processed as specified in Section 8.4.2 of [RFC9031].

Appendix B. Enrollment Hints

This section defines items that can be used in the OPAQUE_INFO field of either EAD_1 or the Access Denied error response. The purpose of the proposed items is to improve protocol scalability, aiming to reduce battery usage and enrollment delay. The main use case is when several potential gateways (V) are detected by U's radio, which can lead to U trying to enroll (and failing) several times until it finds a suitable V.

B.1. Domain Authenticator hints

In case W denies the enrollment of U to a given V, a list of Domain Authenticator hints (v_hint) can be sent from W to U. The hint is optional and is included in the REJECT_INFO item in the Access Denied error message. It consists of a list of application-defined identifiers of V (e.g. MAC addresses, SSIDs, PAN IDs, etc.), as defined below:

```
v_hint = [ 1* bstr ]
```

B.2. Device Hints

U may send a Device hint (u_hint) so that it can help W to select which Vs to include in v_hint. This can be useful in large scale scenarios with many gateways (V). The hint is an optional field included in the OPAQUE_INFO field within EAD_1, and it must be encrypted. The hint itself is application dependent, and can contain GPS coordinates, application-specific tags, the list of Vs detected by U, or other relevant information. It is defined as follows:

```
u_hint: [ 1* bstr ]
```

Appendix C. Examples

This section presents high level examples of the protocol execution.

Note: the examples below include samples of access policies used by W. These are provided for the sake of completeness only, since the authorization mechanism used by W is out of scope in this document.

C.1. Minimal

This is a simple example that demonstrates successful execution of the protocol.

Premises:

* device u1 has ID_U = key id = 14

* the access policy in W specifies, via a list of ID_U, that device u1 can enroll via any domain authenticator, i.e., the list contains ID_U = 14. In this case, the policy only specifies a restriction in terms of U, effectively allowing enrollment via any V.

Execution:

1. device u1 discovers a gateway (v1) and tries to enroll

2. gateway v1 identifies the zero-touch join attempt by checking that the label of EAD_1 = TBD1, and prepares a Voucher Request using the information contained in the value of EAD_1

3. upon receiving the request, W obtains ID_U = 14, authorizes the access, and replies with Voucher Response

C.2. Wrong gateway

In this example, a device u1 tries to enroll a domain via gateway v1, but W denies the request because the pairing (u1, v1) is not configured in its access policies.

This example also illustrates how the REJECT_INFO field of the EDHOC error Access Denied could be used, in this case to suggest that the device should select another gateway for the join procedure.

Premises:

* devices and gateways communicate via Bluetooth Low Energy (BLE), therefore their network identifers are MAC addresses (EUI-48)

* device u1 has ID_U = key id = 14

* there are 3 gateways in the radio range of u1:

  - v1 with MAC address = A2-A1-88-EE-97-75

  - v2 with MAC address = 28-0F-70-84-51-E4

  - v3 with MAC address = 39-63-C9-D0-5C-62

* the access policy in W specifies, via a mapping of shape (ID_U; MAC1, MAC2, ...) that device u1 can only join via gateway v3, i.e., the mapping is: (14; 39-63-C9-D0-5C-62)

* W is able to map the PoP key of the gateways to their respective MAC addresses

Execution:

1. device u1 tries to join via gateway v1, which forwards the request to W

2. W verifies that MAC address A2-A1-88-EE-97-75 is not in the access policy mapping, and replies with an error. The error_content has REJECT_TYPE = 1, and the plaintext of REJECT_INFO contains a list of suggested gateways = [h'3963C9D05C62']. The single element in the list is the 6-byte MAC address of v3, serialized as a bstr.

3. gateway v1 assembles an EDHOC error "Access Denied" with error_content, and sends it to u1

4. device u1 processes the error, decrypts REJECT_INFO, and retries the protocol via gateway v3

Acknowledgments

The authors sincerely thank Aurelio Schellenbaum for his contribution in the initial phase of this work.

Authors' Addresses

Göran Selander
Ericsson AB
Sweden
Email: goran.selander@ericsson.com

John Preuß Mattsson
Ericsson AB
Sweden
Email: john.mattsson@ericsson.com

Malia Vuini
INRIA
France
Email: malisa.vucinic@inria.fr

Geovane Fedrecheski
INRIA
France
Email: geovane.fedrecheski@inria.fr

Michael Richardson
Sandelman Software Works
Canada
Email: mcr+ietf@sandelman.ca

LAKE Working Group
Internet-Draft
Intended status: Standards Track
Expires: 25 July 2024

G. Selander
J. Preuß Mattsson
F. Palombini
Ericsson

22 January 2024

Ephemeral Diffie-Hellman Over COSE (EDHOC)
draft-ietf-lake-edhoc-23

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 25 July 2024.

Table of Contents

1. Introduction

1.1. Motivation

Many Internet of Things (IoT) deployments require technologies which are highly performant in constrained environments [RFC7228]. IoT devices may be constrained in various ways, including memory, storage, processing capacity, and power. The connectivity for these settings may also exhibit constraints such as unreliable and lossy channels, highly restricted bandwidth, and dynamic topology. The IETF has acknowledged this problem by standardizing a range of lightweight protocols and enablers designed for the IoT, including the Constrained Application Protocol (CoAP, [RFC7252]), Concise Binary Object Representation (CBOR, [RFC8949]), and Static Context Header Compression (SCHC, [RFC8724]).

The need for special protocols targeting constrained IoT deployments extends also to the security domain [I-D.ietf-lake-reqs]. Important characteristics in constrained environments are the number of round trips and protocol message sizes, which if kept low can contribute to good performance by enabling transport over a small number of radio frames, reducing latency due to fragmentation or duty cycles, etc. Another important criterion is code size, which may be prohibitively large for certain deployments due to device capabilities or network load during firmware update. Some IoT deployments also need to support a variety of underlying transport technologies, potentially even with a single connection.

Some security solutions for such settings exist already. CBOR Object Signing and Encryption (COSE, [RFC9052]) specifies basic application-layer security services efficiently encoded in CBOR. Another example is Object Security for Constrained RESTful Environments (OSCORE, [RFC8613]) which is a lightweight communication security extension to CoAP using CBOR and COSE. In order to establish good quality cryptographic keys for security protocols such as COSE and OSCORE, the two endpoints may run an authenticated Diffie-Hellman key exchange protocol, from which shared secret keying material can be derived. Such a key exchange protocol should also be lightweight; to prevent bad performance in case of repeated use, e.g., due to device rebooting or frequent rekeying for security reasons; or to avoid latencies in a network formation setting with many devices authenticating at the same time.

This document specifies Ephemeral Diffie-Hellman Over COSE (EDHOC), a lightweight authenticated key exchange protocol providing good security properties including forward secrecy, identity protection, and cipher suite negotiation. Authentication can be based on raw public keys (RPK) or public key certificates and requires the application to provide input on how to verify that endpoints are trusted. This specification supports the referencing of credentials

in order to reduce message overhead, but credentials may alternatively be embedded in the messages. EDHOC does not currently support pre-shared key (PSK) authentication as authentication with static Diffie-Hellman public keys by reference produces equally small message sizes but with much simpler key distribution and identity protection.

EDHOC makes use of known protocol constructions, such as SIGMA [SIGMA], the Noise XX pattern [Noise], and Extract-and-Expand [RFC5869]. EDHOC uses COSE for cryptography and identification of credentials (including COSE_Key, CBOR Web Token (CWT), CWT Claims Set (CCS), X.509, and CBOR encoded X.509 (C509) certificates, see Section 3.5.2). COSE provides crypto agility and enables the use of future algorithms and credential types targeting IoT.

EDHOC is designed for highly constrained settings making it especially suitable for low-power networks [RFC8376] such as Cellular IoT, 6TiSCH, and LoRaWAN. A main objective for EDHOC is to be a lightweight authenticated key exchange for OSCORE, i.e., to provide authentication and session key establishment for IoT use cases such as those built on CoAP [RFC7252] involving 'things' with embedded microcontrollers, sensors, and actuators. By reusing the same lightweight primitives as OSCORE (CBOR, COSE, CoAP) the additional code size can be kept very low. Note that while CBOR and COSE primitives are built into the protocol messages, EDHOC is not bound to a particular transport.

A typical setting is when one of the endpoints is constrained or in a constrained network, and the other endpoint is a node on the Internet (such as a mobile phone). Thing-to-thing interactions over constrained networks are also relevant since both endpoints would then benefit from the lightweight properties of the protocol. EDHOC could, e.g., be run when a device connects for the first time, or to establish fresh keys which are not revealed by a later compromise of the long-term keys.

1.2. Message Size Examples

Examples of EDHOC message sizes are shown in Figure 1, using different kinds of authentication keys and COSE header parameters for identification: static Diffie-Hellman keys or signature keys, either in CBOR Web Token (CWT) / CWT Claims Set (CCS) [RFC8392] identified by a key identifier using 'kid' [RFC9052], or in X.509 certificates identified by a hash value using 'x5t' [RFC9360]. As a comparison, in the case of RPK authentication, the EDHOC message size when transferred in CoAP can be less than 1/7 of the DTLS 1.3 handshake [RFC9147] with ECDHE and connection ID, see Section 2 of [I-D.ietf-iotops-security-protocol-comparison].

	Static DH Keys		Signature Keys	
	kid	x5t	kid	x5t
message_1	37	37	37	37
message_2	45	58	102	115
message_3	19	33	77	90
Total	101	128	216	242

Figure 1: Examples of EDHOC message sizes in bytes.

1.3. Document Structure

The remainder of the document is organized as follows: Section 2 outlines EDHOC authenticated with signature keys, Section 3 describes the protocol elements of EDHOC, including formatting of the ephemeral public keys, Section 4 specifies the key derivation, Section 5 specifies message processing for EDHOC authenticated with signature keys or static Diffie-Hellman keys, Section 6 describes the error messages, Section 7 describes EDHOC support for transport that does not handle message duplication, and Section 8 lists compliance requirements. Note that normative text is also used in appendices, in particular Appendix A.

1.4. Terminology and Requirements Language

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

Readers are expected to be familiar with the terms and concepts described in CBOR [RFC8949], CBOR Sequences [RFC8742], COSE structures and processing [RFC9052], COSE algorithms [RFC9053], CWT and CWT Claims Set [RFC8392], and the Concise Data Definition Language (CDDL, [RFC8610]), which is used to express CBOR data structures. Examples of CBOR and CDDL are provided in Appendix C.1. When referring to CBOR, this specification always refers to Deterministically Encoded CBOR as specified in Sections 4.2.1 and 4.2.2 of [RFC8949]. The single output from authenticated encryption (including the authentication tag) is called "ciphertext", following [RFC5116].

2. EDHOC Outline

EDHOC specifies different authentication methods of the ephemeral-ephemeral Diffie-Hellman key exchange: signature keys and static Diffie-Hellman keys. This section outlines the signature key based method. Further details of protocol elements and other authentication methods are provided in the remainder of this document.

SIGMA (SIGn-and-MAc) is a family of theoretical protocols with a large number of variants [SIGMA]. Like in IKEv2 [RFC7296] and (D)TLS 1.3 [RFC8446][RFC9147], EDHOC authenticated with signature keys is built on a variant of the SIGMA protocol, SIGMA-I, which provides identity protection against active attacks on the party initiating the protocol. Also like IKEv2, EDHOC implements the MAC-then-Sign variant of the SIGMA-I protocol. The message flow (excluding an optional fourth message) is shown in Figure 2.

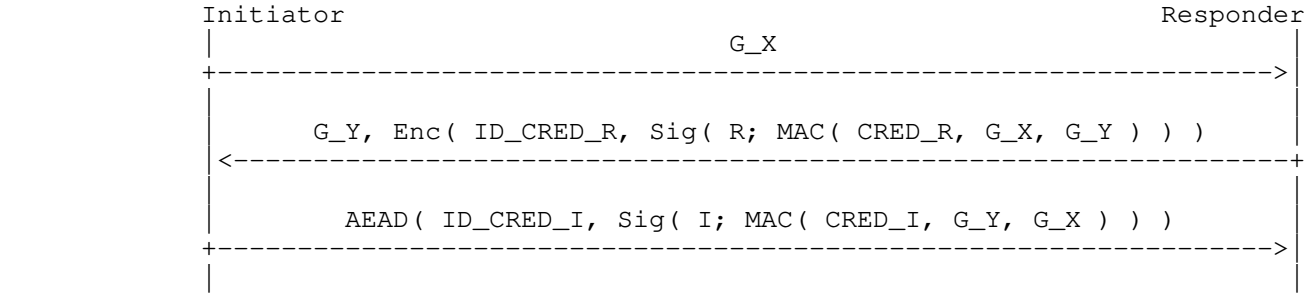


Figure 2: MAC-then-Sign variant of the SIGMA-I protocol used by EDHOC method 0.

The parties exchanging messages in an EDHOC session are called Initiator (I) and Responder (R), where the Initiator sends message_1 (see Section 3). They exchange ephemeral public keys, compute a shared secret session key PRK_out, and derive symmetric application keys used to protect application data.

* G_X and G_Y are the ECDH ephemeral public keys of I and R, respectively.

* CRED_I and CRED_R are the authentication credentials containing the public authentication keys of I and R, respectively.

* ID_CRED_I and ID_CRED_R are used to identify and optionally transport the credentials of the Initiator and the Responder, respectively.

* Sig(I; .) and Sig(R; .) denote signatures made with the private authentication key of I and R, respectively.

* Enc(), AEAD(), and MAC() denotes encryption, authenticated encryption with additional data, and message authentication code - crypto algorithms applied with keys derived from one or more shared secrets calculated during the protocol.

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

* Transcript hashes (hashes of message data) TH_2, TH_3, TH_4 used for key derivation and as additional authenticated data.

* Computationally independent keys derived from the ECDH shared secret and used for authenticated encryption of different messages.

* An optional fourth message giving key confirmation to I in deployments where no protected application data is sent from R to I.

* A keying material exporter and a key update function with forward secrecy.

* Secure negotiation of cipher suite.

* Method types, error handling, and padding.

* Selection of connection identifiers C_I and C_R which may be used in EDHOC to identify protocol state.

* Transport of external authorization data.

EDHOC is designed to encrypt and integrity protect as much information as possible. Symmetric keys and random material used in EDHOC are derived using EDHOC_KDF with as much previous information as possible, see Figure 8. EDHOC is furthermore designed to be as compact and lightweight as possible, in terms of message sizes, processing, and the ability to reuse already existing CBOR, COSE, and CoAP libraries. Like in (D)TLS, authentication is the responsibility of the application. EDHOC identifies (and optionally transports) authentication credentials, and provides proof-of-possession of the private authentication key.

To simplify for implementors, the use of CBOR and COSE in EDHOC is summarized in Appendix C. Test vectors including CBOR diagnostic notation are provided in [I-D.ietf-lake-traces].

3. Protocol Elements

3.1. General

The EDHOC protocol consists of three mandatory messages (message_1, message_2, message_3), an optional fourth message (message_4), and an error message, between an Initiator (I) and a Responder (R). The odd messages are sent by I, the even by R. Both I and R can send error messages. The roles have slightly different security properties which should be considered when the roles are assigned, see Section 9.1. All EDHOC messages are CBOR Sequences [RFC8742], and are defined to be deterministically encoded CBOR as specified in Section 4.2.1 of [RFC8949]. Figure 3 illustrates an EDHOC message flow with the optional fourth message as well as the content of each message. The protocol elements in the figure are introduced in Section 3 and Section 5. Message formatting and processing are specified in Section 5 and Section 6.

Application data may be protected using the agreed application algorithms (AEAD, hash) in the selected cipher suite (see Section 3.6) and the application can make use of the established connection identifiers C_I and C_R (see Section 3.3). Media types that may be used for EDHOC are defined in Section 10.8.

The Initiator can derive symmetric application keys after creating EDHOC message_3, see Section 4.2.1. Protected application data can therefore be sent in parallel or together with EDHOC message_3. EDHOC message_4 is typically not sent.

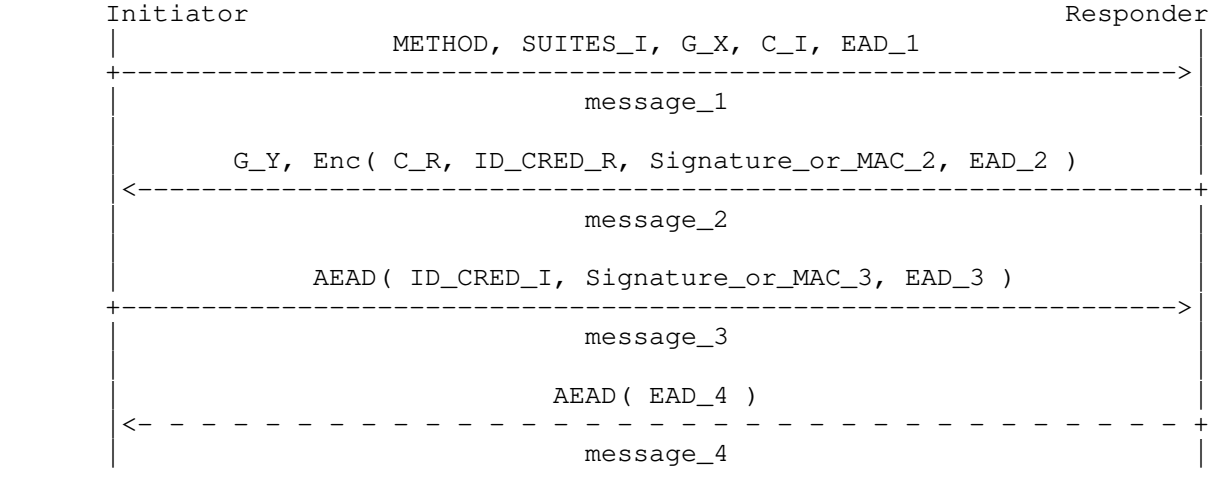


Figure 3: EDHOC message flow including the optional fourth message.

3.2. Method

The data item METHOD in message_1 (see Section 5.2.1), is an integer specifying the authentication method. EDHOC supports authentication with signature or static Diffie-Hellman keys, as defined in the four authentication methods: 0, 1, 2, and 3, see Figure 4. When using a static Diffie-Hellman key the authentication is provided by a Message Authentication Code (MAC) computed from an ephemeral-static ECDH shared secret which enables significant reductions in message sizes. Note that also in the static Diffie-Hellman based authentication methods there is an ephemeral-ephemeral Diffie-Hellman key exchange.

The Initiator and the Responder need to have agreed on a single method to be used for EDHOC, see Section 3.9.

Method Type Value	Initiator Authentication Key	Responder Authentication Key
0	Signature Key	Signature Key
1	Signature Key	Static DH Key
2	Static DH Key	Signature Key
3	Static DH Key	Static DH Key

Figure 4: Authentication keys for method types.

EDHOC does not have a dedicated message field to indicate the protocol version. Breaking changes to EDHOC can be introduced by specifying and registering new methods.

3.3. Connection Identifiers

EDHOC includes the selection of connection identifiers (C_I, C_R) identifying a connection for which keys are agreed.

Connection identifiers may be used to correlate EDHOC messages and facilitate the retrieval of protocol state during an EDHOC session (see Section 3.4), or may be used in applications of EDHOC, e.g., in OSCORE (see Section 3.3.3). The connection identifiers do not have any cryptographic purpose in EDHOC and only facilitate the retrieval of security data associated with the protocol state.

Connection identifiers in EDHOC are intrinsically byte strings. Most constrained devices only have a few connections for which short identifiers may be sufficient. In some cases minimum length identifiers are necessary to comply with overhead requirements. However, CBOR byte strings - with the exception of the empty byte

string h which encodes as one byte (0x40) - are encoded as two or more bytes. To enable one-byte encoding of certain byte strings while maintaining CBOR encoding, EDHOC represents certain identifiers as CBOR integers on the wire, see Section 3.3.2.

3.3.1. Selection of Connection Identifiers

C_I and C_R are chosen by I and R, respectively. The Initiator selects C_I and sends it in message_1 for the Responder to use as a reference to the connection in communication with the Initiator. The Responder selects C_R and sends it in message_2 for the Initiator to use as a reference to the connection in communications with the Responder.

If connection identifiers are used by an application protocol for which EDHOC establishes keys then the selected connection identifiers SHALL adhere to the requirements for that protocol, see Section 3.3.3 for an example.

3.3.2. Representation of Byte String Identifiers

To allow identifiers with minimal overhead on the wire, certain byte strings used in connection identifiers and credential identifiers (see Section Section 3.5.3) are defined to have integer representations.

The integers with one-byte CBOR encoding are -24, ..., 23, see Figure 5.

Integer:	-24	-23	...	-11	...	-2	-1	0	1	...	15	...	23
Encoding:	37	36	...	2A	...	21	20	00	01	...	0F	...	17

Figure 5: One-byte CBOR encoded integers.

The byte strings which coincide with a one-byte CBOR encoding of an integer MUST be represented by the CBOR encoding of that integer. Other byte strings are simply encoded as CBOR byte strings.

For example:

* 0x21 is represented by 0x21 (CBOR encoding of the integer -2), not by 0x4121 (CBOR encoding of the byte string 0x21).

* 0x0D is represented by 0x0D (CBOR encoding of the integer 13), not by 0x410D (CBOR encoding of the byte string 0x0D).

* 0x18 is represented by 0x4118 (CBOR encoding of the byte string 0x18).

* 0x38 is represented by 0x4138 (CBOR encoding of the byte string 0x38).

* 0xABCD is represented by 0x42ABCD (CBOR encoding of the byte string 0xABCD).

One may view this representation of byte strings as a transport encoding: a byte string which parses as the one-byte CBOR encoding of an integer (i.e., integer in the interval -24, ..., 23) is just copied directly into the message, a byte string which does not is encoded as a CBOR byte string during transport.

Implementation Note: When implementing the byte string identifier representation, it can in some programming languages help to define a new type, or other data structure, which (in its user facing API) behaves like a byte string, but when serializing to CBOR produces a CBOR byte string or a CBOR integer depending on its value.

3.3.3. Use of Connection Identifiers with OSCORE

For OSCORE, the choice of connection identifier results in the endpoint selecting its Recipient ID, see Section 3.1 of [RFC8613], for which certain uniqueness requirements apply, see Section 3.3 of [RFC8613]. Therefore, the Initiator and the Responder MUST NOT select connection identifiers such that it results in the same OSCORE Recipient ID. Since the connection identifier is a byte string, it is converted to an OSCORE Recipient ID equal to the byte string.

Examples:

* A connection identifier 0xFF (represented in the EDHOC message as 0x41FF, see Section 3.3.2) is converted to the OSCORE Recipient ID 0xFF.

* A connection identifier 0x21 (represented in the EDHOC message as 0x21, see Section 3.3.2) is converted to the OSCORE Recipient ID 0x21.

3.4. Transport

Cryptographically, EDHOC does not put requirements on the underlying layers. Received messages are processed as the expected next message according to protocol state, see Section 5. If processing fails for any reason then, typically, an error message is attempted to be sent and the EDHOC session is aborted.

EDHOC is not bound to a particular transport layer and can even be used in environments without IP. Ultimately, the application is free to choose how to transport EDHOC messages including errors. In order to avoid unnecessary message processing or protocol termination, it is RECOMMENDED to use reliable transport, such as CoAP in reliable mode, which is the default transport, see Appendix A.2. In general, the transport SHOULD handle:

* message loss,

* message duplication, see Section 7 for an alternative,

* flow control,

* congestion control,

* fragmentation and reassembly,

* demultiplexing EDHOC messages from other types of messages,

* denial-of-service mitigation,

* message correlation, see Section 3.4.1.

EDHOC does not require error free transport since a change in message content is detected through the transcript hashes in a subsequent integrity verification, see Section 5. The transport does not require additional means to handle message reordering because of the lockstep processing of EDHOC.

EDHOC is designed to enable an authenticated key exchange with small messages, where the minimum message sizes are of the order illustrated in the first column of Figure 1. There is no maximum message size specified by the protocol; this is for example dependent on the size of authentication credentials (if they are transported, see Section 3.5).

The use of transport is specified in the application profile, which in particular may specify limitations in message sizes, see Section 3.9.

3.4.1. EDHOC Message Correlation

Correlation between EDHOC messages is needed to facilitate the retrieval of protocol state and security context during an EDHOC session. It is also helpful for the Responder to get an indication that a received EDHOC message is the beginning of a new EDHOC session, such that no existing protocol state or security context needs to be retrieved.

Correlation may be based on existing mechanisms in the transport protocol, for example, the CoAP Token may be used to correlate EDHOC messages in a CoAP response and in an associated CoAP request. The connection identifiers may also be used to correlate EDHOC messages.

If correlation between consecutive messages is not provided by other means then the transport binding SHOULD mandate prepending of an appropriate connection identifier (when available from the EDHOC protocol) to the EDHOC message. If message_1 indication is not provided by other means, then the transport binding SHOULD mandate prepending of message_1 with the CBOR simple value true (0xf5).

Transport of EDHOC in CoAP payloads is described in Appendix A.2, including how to use connection identifiers and message_1 indication with CoAP. A similar construction is possible for other client-server protocols. Protocols that do not provide any correlation at all can prescribe prepending of the peer's connection identifier to all messages.

Note that correlation between EDHOC messages may be obtained without transport support or connection identifiers, for example if the endpoints only accept a single instance of the protocol at a time, and execute conditionally on a correct sequence of messages.

3.5. Authentication Parameters

EDHOC supports various settings for how the other endpoint's authentication (public) key may be transported, identified, and trusted.

EDHOC performs the following authentication related operations:

* EDHOC transports information about credentials in ID_CRED_I and ID_CRED_R (described in Section 3.5.3). Based on this information, the authentication credentials CRED_I and CRED_R (described in Section 3.5.2) can be obtained. EDHOC may also transport certain authentication related information as External Authorization Data (see Section 3.8).

* EDHOC uses the authentication credentials in two ways (see Section 5.3.2 and Section 5.4.2):

  - The authentication credential is input to the integrity verification using the MAC fields.

  - The authentication key of the authentication credential is used with the Signature_or_MAC field to verify proof-of-possession of the private key.

Other authentication related verifications are out of scope for EDHOC, and is the responsibility of the application. In particular, the authentication credential needs to be validated in the context of the connection for which EDHOC is used, see Appendix D. EDHOC MUST allow the application to read received information about credential (ID_CRED_R, ID_CRED_I). EDHOC MUST have access to the authentication key and the authentication credential.

Note that the type of authentication key, authentication credential, and the identification of the credential have a large impact on the message size. For example, the Signature_or_MAC field is much smaller with a static DH key than with a signature key. A CWT Claims Set (CCS) is much smaller than a self-signed certificate/CWT, but if it is possible to reference the credential with a COSE header like 'kid', then that is in turn much smaller than a CCS.

3.5.1. Authentication Keys

The authentication key (i.e., the public key used for authentication) MUST be a signature key or static Diffie-Hellman key. The Initiator and the Responder MAY use different types of authentication keys, e.g., one uses a signature key and the other uses a static Diffie-Hellman key.

The authentication key algorithm needs to be compatible with the method and the selected cipher suite (see Section 3.6). The authentication key algorithm needs to be compatible with the EDHOC key exchange algorithm when static Diffie-Hellman authentication is used, and compatible with the EDHOC signature algorithm when signature authentication is used.

Note that for most signature algorithms, the signature is determined by the signature algorithm and the authentication key algorithm together. When using static Diffie-Hellman keys the Initiator's and Responder's private authentication keys are denoted as I and R, respectively, and the public authentication keys are denoted G_I and G_R, respectively.

For X.509 certificates the authentication key is represented by a SubjectPublicKeyInfo field. For CWT and CCS (see Section 3.5.2)) the authentication key is represented by a 'cnf' claim [RFC8747] containing a COSE_Key [RFC9052]. In EDHOC, a raw public key (RPK) is an authentication key encoded as a COSE_Key wrapped in a CCS.

3.5.2. Authentication Credentials

The authentication credentials, CRED_I and CRED_R, contains the public authentication key of the Initiator and the Responder, respectively. We use the notation CRED_x to refer to CRED_I or CRED_R. Requirements on CRED_x applies both to CRED_I and to CRED_R. The authentication credential typically also contains other parameters that needs to be verified by the application, see Appendix D, and in particular information about the identity ("subject") of the endpoint to prevent misbinding attacks, see Appendix D.2.

EDHOC relies on COSE for identification of credentials (see Section 3.5.3), for example X.509 certificates [RFC9360], C509 certificates [I-D.ietf-cose-cbor-encoded-cert], CWTs [RFC8392] and CWT Claims Sets (CCS) [RFC8392]. When the identified credential is a chain or a bag, the authentication credential CRED_x is just the end entity X.509 or C509 certificate / CWT. In the choice between chain or bag it is RECOMMENDED to use a chain, since the certificates in a bag are unordered and may contain self-signed and extraneous certificates, which can add complexity to the process of extracting the end entity certificate. The Initiator and the Responder MAY use different types of authentication credentials, e.g., one uses an RPK and the other uses a public key certificate.

Since CRED_R is used in the integrity verification, see Section 5.3.2, it needs to be specified such that it is identical when used by Initiator or Responder. Similarly for CRED_I, see Section 5.4.2. The Initiator and Responder are expected to agree on the specific encoding of the authentication credentials, see Section 3.9. It is RECOMMENDED that the COSE 'kid' parameter, when used to identify the authentication credential, refers to a such a specific encoding of the authentication credential. The Initiator and Responder SHOULD use an available authentication credential (transported in EDHOC or otherwise provisioned) without re-encoding. If for some reason re-encoding of an authentication credential passed by reference may occur, then a potential common encoding for CBOR based credentials is deterministically encoded CBOR as specified in Sections 4.2.1 and 4.2.2 of [RFC8949].

* When the authentication credential is an X.509 certificate, CRED_x SHALL be the DER encoded certificate, encoded as a bstr [RFC9360].

* When the authentication credential is a C509 certificate, CRED_x SHALL be the C509Certificate [I-D.ietf-cose-cbor-encoded-cert].

* When the authentication credential is a CWT including a COSE_Key, CRED_x SHALL be the untagged CWT.

* When the authentication credential includes a COSE_Key but is not in a CWT, CRED_x SHALL be an untagged CWT Claims Set (CCS). This is how RPKs are encoded, see Figure 6 for an example.

  - Naked COSE_Keys are thus dressed as CCS when used in EDHOC, in its simplest form by prefixing the COSE_Key with 0xA108A101 (a map with a 'cnf' claim). In that case the resulting authentication credential contains no other identity than the public key itself, see Appendix D.2.

An example of CRED_x is shown below:

```
{                                                  /CCS/
  2 : "42-50-31-FF-EF-37-32-39",                   /sub/
  8 : {                                            /cnf/
    1 : {                                          /COSE_Key/
      1 : 1,                                       /kty/
      2 : h'00',                                   /kid/
     -1 : 4,                                       /crv/
     -2 : h'b1a3e89460e88d3a8d54211dc95f0b90       /x/
           3ff205eb71912d6db8f4af980d2db83a'
    }
  }
}
```

Figure 6: CWT Claims Set (CCS) containing an X25519 static Diffie-Hellman key and an EUI-64 identity.

3.5.3. Identification of Credentials

The ID_CRED fields, ID_CRED_R and ID_CRED_I, are transported in message_2 and message_3, respectively, see Section 5.3.2 and Section 5.4.2. We use the notation ID_CRED_x to refer to ID_CRED_I or ID_CRED_R. Requirements on ID_CRED_x applies both to ID_CRED_I and to ID_CRED_R. The ID_CRED fields are used to identify and optionally transport credentials:

* ID_CRED_R is intended to facilitate for the Initiator retrieving the authentication credential CRED_R and the authentication key of R.

* ID_CRED_I is intended to facilitate for the Responder retrieving the authentication credential CRED_I and the authentication key of I.

ID_CRED_x may contain the authentication credential CRED_x, for x = I or R, but for many settings it is not necessary to transport the authentication credential within EDHOC. For example, it may be pre-provisioned or acquired out-of-band over less constrained links. ID_CRED_I and ID_CRED_R do not have any cryptographic purpose in EDHOC since the authentication credentials are integrity protected.

EDHOC relies on COSE for identification of credentials and supports all credential types for which COSE header parameters are defined including X.509 certificates ([RFC9360]), C509 certificates ([I-D.ietf-cose-cbor-encoded-cert]), CWT (see Section 3.5.3.1) and CWT Claims Set (see Section 3.5.3.1).

ID_CRED_I and ID_CRED_R are of type COSE header_map, as defined in Section 3 of [RFC9052], and contains one or more COSE header parameters. If a map contains several header paramerers, the labels do not need to be sorted in bytewise lexicographic order. ID_CRED_I and ID_CRED_R MAY contain different header parameters. The header parameters typically provide some information about the format of the credential.

Example: X.509 certificates can be identified by a hash value using the 'x5t' parameter, see Section 2 of [RFC9360]:

* ID_CRED_x = { 34 : COSE_CertHash }, for x = I or R,

Example: CWT or CCS can be identified by a key identifier using the 'kid' parameter, see Section 3.1 of [RFC9052]:

* ID_CRED_x = { 4 : kid_x }, where kid_x : kid, for x = I or R.

Note that COSE header parameters in ID_CRED_x are used to identify the message sender's credential. There is therefore no reason to use the "-sender" header parameters, such as x5t-sender, defined in Section 3 of [RFC9360]. Instead, the corresponding parameter without "-sender", such as x5t, SHOULD be used.

As stated in Section 3.1 of [RFC9052], applications MUST NOT assume that 'kid' values are unique and several keys associated with a 'kid' may need to be checked before the correct one is found. Applications might use additional information such as 'kid context' or lower layers to determine which key to try first. Applications should strive to make ID_CRED_x as unique as possible, since the recipient may otherwise have to try several keys.

See Appendix C.3 for more examples.

3.5.3.1. COSE Header Parameters for CWT and CWT Claims Set

This document registers two new COSE header parameters 'kcwt' and 'kccs' for use with CBOR Web Token (CWT, [RFC8392]) and CWT Claims Set (CCS, [RFC8392]), respectively. The CWT/CCS MUST contain a COSE_Key in a 'cnf' claim [RFC8747]. There may be any number of additional claims present in the CWT/CCS.

CWTs sent in 'kcwt' are protected using a MAC or a signature and are similar to a certificate (when with public key cryptography) or a Kerberos ticket (when used with symmetric key cryptography). CCSs sent in 'kccs' are not protected and are therefore similar to raw public keys or self-signed certificates.

Security considerations for 'kcwt' and 'kccs' are made in Section 9.8.

3.5.3.2. Compact Encoding of ID_CRED Fields for 'kid'

To comply with the LAKE message size requirements, see [I-D.ietf-lake-reqs], two optimizations are made for the case when ID_CRED_x, for x = I or R, contains a single 'kid' parameter.

1. The CBOR map { 4 : kid_x } is replaced by the byte string kid_x.

2. The representation of identifiers specified in Section 3.3.2 is applied to kid_x.

These optimizations MUST be applied if and only if ID_CRED_x = { 4 : kid_x } and ID_CRED_x in PLAINTEXT_y of message_y, y = 2 or 3, see Section 5.3.2 and Section 5.4.2. Note that these optimizations are not applied to instances of ID_CRED_x which have no impact on message size, e.g., context_y, or the COSE protected header. Examples:

* For ID_CRED_x = { 4 : h'FF' }, the encoding in PLAINTEXT_y is not the CBOR map 0xA10441FF but the CBOR byte string h'FF', i.e., 0x41FF.

* For ID_CRED_x = { 4 : h'21' }, the encoding in PLAINTEXT_y is neither the CBOR map 0xA1044121, nor the CBOR byte string h'21', i.e., 0x4121, but the CBOR integer 0x21.

3.6. Cipher Suites

An EDHOC cipher suite consists of an ordered set of algorithms from the "COSE Algorithms" and "COSE Elliptic Curves" registries as well as the EDHOC MAC length. All algorithm names and definitions follow from COSE algorithms [RFC9053]. Note that COSE sometimes uses peculiar names such as ES256 for ECDSA with SHA-256, A128 for AES-128, and Ed25519 for the curve edwards25519. Algorithms need to be specified with enough parameters to make them completely determined. The EDHOC MAC length MUST be at least 8 bytes. Any cryptographic algorithm used in the COSE header parameters in ID_CRED fields is selected independently of the selected cipher suite. EDHOC is currently only specified for use with key exchange algorithms of type ECDH curves, but any Key Encapsulation Method (KEM), including Post-Quantum Cryptography (PQC) KEMs, can be used in method 0, see Section 9.4. Use of other types of key exchange algorithms to replace static DH authentication (method 1,2,3) would likely require a specification updating EDHOC with new methods.

EDHOC supports all signature algorithms defined by COSE. Just like in (D)TLS 1.3 [RFC8446][RFC9147] and IKEv2 [RFC7296], a signature in COSE is determined by the signature algorithm and the authentication key algorithm together, see Section 3.5.1. The exact details of the authentication key algorithm depend on the type of authentication credential. COSE supports different formats for storing the public authentication keys including COSE_Key and X.509, which use different names and ways to represent the authentication key and the authentication key algorithm.

An EDHOC cipher suite consists of the following parameters:

* EDHOC AEAD algorithm

* EDHOC hash algorithm

* EDHOC MAC length in bytes (Static DH)

* EDHOC key exchange algorithm (ECDH curve)

* EDHOC signature algorithm

* Application AEAD algorithm

* Application hash algorithm

Each cipher suite is identified with a pre-defined integer label.

EDHOC can be used with all algorithms and curves defined for COSE. Implementations can either use any combination of COSE algorithms and parameters to define their own private cipher suite, or use one of the pre-defined cipher suites. Private cipher suites can be identified with any of the four values -24, -23, -22, -21. The pre-defined cipher suites are listed in the IANA registry (Section 10.2) with initial content outlined here:

* Cipher suites 0-3, based on AES-CCM, are intended for constrained IoT where message overhead is a very important factor. Note that AES-CCM-16-64-128 and AES-CCM-16-128-128 are compatible with the IEEE CCM* mode.

  - Cipher suites 1 and 3 use a larger tag length (128-bit) in EDHOC than in the Application AEAD algorithm (64-bit).

* Cipher suites 4 and 5, based on ChaCha20, are intended for less constrained applications and only use 128-bit tag lengths.

* Cipher suite 6, based on AES-GCM, is for general non-constrained applications. It consists of high performance algorithms that are widely used in non-constrained applications.

* Cipher suites 24 and 25 are intended for high security applications such as government use and financial applications. These cipher suites do not share any algorithms. Cipher suite 24 consists of algorithms from the CNSA 1.0 suite [CNSA].

The different methods (Section 3.2) use the same cipher suites, but some algorithms are not used in some methods. The EDHOC signature algorithm is not used in methods without signature authentication.

The Initiator needs to have a list of cipher suites it supports in order of preference. The Responder needs to have a list of cipher suites it supports. SUITES_I contains cipher suites supported by the Initiator, formatted and processed as detailed in Section 5.2.1 to secure the cipher suite negotiation. Examples of cipher suite negotiation are given in Section 6.3.2.

3.7. Ephemeral Public Keys

The ephemeral public keys in EDHOC (G_X and G_Y) use compact representation of elliptic curve points, see Appendix B. In COSE, compact representation is achieved by formatting the ECDH ephemeral public keys as COSE_Keys of type EC2 or OKP according to Sections 7.1 and 7.2 of [RFC9053], but only including the 'x' parameter in G_X and G_Y. For Elliptic Curve Keys of type EC2, compact representation MAY be used also in the COSE_Key. COSE always uses compact output for

Elliptic Curve Keys of type EC2. If the COSE implementation requires a 'y' parameter, the value y = false or a calculated y-coordinate can be used, see Appendix B.

3.8. External Authorization Data (EAD)

In order to reduce round trips and the number of messages, or to simplify processing, external security applications may be integrated into EDHOC by transporting authorization related data in the messages.

EDHOC allows processing of external authorization data (EAD) to be defined in a separate specification, and sent in dedicated fields of the four EDHOC messages (EAD_1, EAD_2, EAD_3, EAD_4). EAD is opaque data to EDHOC.

Each EAD field, EAD_x for x = 1, 2, 3 or 4, is a CBOR sequence (see Appendix C.1) consisting of one or more EAD items. An EAD item ead is a CBOR sequence of an ead_label and an optional ead_value, see Figure 7 and Appendix C.2 for the CDDL definitions.

```
ead = (
  ead_label : int,
  ? ead_value : bstr,
)
```

Figure 7: EAD item.

A security application may register one or more EAD labels, see Section 10.5, and specify the associated processing and security considerations. The IANA registry contains the absolute value of the ead_label, |ead_label|; the same ead_value applies independently of sign of ead_label.

An EAD item can be either critical or non-critical, determined by the sign of the ead_label in the EAD item transported in the EAD field. A negative value indicates that the EAD item is critical and a non-negative value indicates that the EAD item is non-critical.

If an endpoint receives a critical EAD item it does not recognize, or a critical EAD item that contains information that it cannot process, then the endpoint MUST send an EDHOC error message back as defined in Section 6, and the EDHOC session MUST be aborted. The EAD item specification defines the error processing. A non-critical EAD item can be ignored.

The security application registering a new EAD item needs to describe under what conditions the EAD item is critical or non-critical, and thus whether the ead_label is used with negative or positive sign. ead_label = 0 is used for padding, see Section 3.8.1.

The security application may define multiple uses of certain EAD items, e.g., the same EAD item may be used in different EDHOC messages. Multiple occurrences of an EAD item in one EAD field may also be specified, but the criticality of the repeated EAD item is expected to be the same.

The EAD fields of EDHOC MUST only be used with registered EAD items, see Section 10.5. Examples of the use of EAD are provided in Appendix E.

3.8.1. Padding

EDHOC message_1 and the plaintext of message_2, message_3 and message_4 can be padded with the use of the corresponding EAD_x field, for x = 1, 2, 3, 4. Padding in EAD_1 mitigates amplification attacks (see Section 9.7), and padding in EAD_2, EAD_3, and EAD_4 hides the true length of the plaintext (see Section 9.6). Padding MUST be ignored and discarded by the receiving application.

Padding is obtained by using an EAD item with ead_label = 0 and a (pseudo-)randomly generated byte string of appropriate length as ead_value, noting that the ead_label and the CBOR encoding of ead_value also add bytes. Examples:

* One byte padding (optional ead_value omitted):
  - EAD_x = 0x00
* Two bytes padding, using the empty byte string (0x40) as ead_value:
  - EAD_x = 0x0040
* Three bytes padding, constructed from the pseudorandomly generated ead_value 0xe9 encoded as byte string:
  - EAD_x = 0x0041e9

Multiple occurrences of EAD items with ead_label = 0 are allowed. Certain padding lengths require the use of at least two such EAD items.

Note that padding is non-critical because the intended behaviour when receiving is to ignore it.

3.9. Application Profile

EDHOC requires certain parameters to be agreed upon between Initiator and Responder. Some parameters can be negotiated through the protocol execution (specifically, cipher suite, see Section 3.6) but other parameters are only communicated and may not be negotiated (e.g., which authentication method is used, see Section 3.2). Yet other parameters need to be known out-of-band to ensure successful completion, e.g., whether message_4 is used or not. The application decides which endpoint is Initiator and which is Responder.

The purpose of an application profile is to describe the intended use of EDHOC to allow for the relevant processing and verifications to be made, including things like:

1. How the endpoint detects that an EDHOC message is received. This includes how EDHOC messages are transported, for example in the payload of a CoAP message with a certain Uri-Path or Content-Format; see Appendix A.2.

   * The method of transporting EDHOC messages may also describe data carried along with the messages that are needed for the transport to satisfy the requirements of Section 3.4, e.g., connection identifiers used with certain messages, see Appendix A.2.

2. Authentication method (METHOD; see Section 3.2).

3. Profile for authentication credentials (CRED_I, CRED_R; see Section 3.5.2), e.g., profile for certificate or CCS, including supported authentication key algorithms (subject public key algorithm in X.509 or C509 certificate).

4. Type used to identify credentials (ID_CRED_I, ID_CRED_R; see Section 3.5.3).

5. Use and type of external authorization data (EAD_1, EAD_2, EAD_3, EAD_4; see Section 3.8).

6. Identifier used as the identity of the endpoint; see Appendix D.2.

7. If message_4 shall be sent/expected, and if not, how to ensure a protected application message is sent from the Responder to the Initiator; see Section 5.5.

The application profile may also contain information about supported cipher suites. The procedure for selecting and verifying a cipher suite is still performed as described in Section 5.2.1 and Section 6.3, but it may become simplified by this knowledge. EDHOC messages can be processed without the application profile, i.e., the EDHOC messages includes information about the type and length of all fields.

An example of an application profile is shown in Appendix F.

For some parameters, like METHOD, type of ID_CRED field or EAD, the receiver of an EDHOC message is able to verify compliance with the application profile, and if it needs to fail because of lack of compliance, to infer the reason why the EDHOC session failed.

For other encodings, like the profiling of CRED_x in the case that it is not transported, it may not be possible to verify that lack of compliance with the application profile was the reason for failure: Integrity verification in message_2 or message_3 may fail not only because of a wrong credential. For example, in case the Initiator uses a public key certificate by reference (i.e., not transported within the protocol) then both endpoints need to use an identical data structure as CRED_I or else the integrity verification will fail.

Note that it is not necessary for the endpoints to specify a single transport for the EDHOC messages. For example, a mix of CoAP and HTTP may be used along the path, and this may still allow correlation between messages.

The application profile may be dependent on the identity of the other endpoint, or other information carried in an EDHOC message, but it then applies only to the later phases of the protocol when such information is known. (The Initiator does not know the identity of the Responder before having verified message_2, and the Responder does not know the identity of the Initiator before having verified message_3.)

Other conditions may be part of the application profile, such as what is the target application or use (if there is more than one application/use) to the extent that EDHOC can distinguish between them. In case multiple application profiles are used, the receiver needs to be able to determine which is applicable for a given EDHOC session, for example based on URI to which the EDHOC message is sent, or external authorization data type.

4. Key Derivation

4.1. Keys for EDHOC Message Processing

EDHOC uses Extract-and-Expand [RFC5869] with the EDHOC hash algorithm in the selected cipher suite to derive keys used in message processing. This section defines EDHOC_Extract (Section 4.1.1) and EDHOC_Expand (Section 4.1.2), and how to use them to derive PRK_out (Section 4.1.3) which is the shared secret session key resulting from a completed EDHOC session.

EDHOC_Extract is used to derive fixed-length uniformly pseudorandom keys (PRK) from ECDH shared secrets. EDHOC_Expand is used to define EDHOC_KDF for generating MACs and for deriving output keying material (OKM) from PRKs.

In EDHOC a specific message is protected with a certain pseudorandom key, but how the key is derived depends on the authentication method (Section 3.2) as detailed in Section 5.

4.1.1. EDHOC_Extract

The pseudorandom keys (PRKs) used for EDHOC message processing are derived using EDHOC_Extract:

```
PRK = EDHOC_Extract( salt, IKM )
```

where the input keying material (IKM) and salt are defined for each PRK below.

The definition of EDHOC_Extract depends on the EDHOC hash algorithm of the selected cipher suite:

* if the EDHOC hash algorithm is SHA-2, then EDHOC_Extract(salt, IKM) = HKDF-Extract(salt, IKM) [RFC5869]

* if the EDHOC hash algorithm is SHAKE128, then EDHOC_Extract(salt, IKM) = KMAC128(salt, IKM, 256, "")

* if the EDHOC hash algorithm is SHAKE256, then EDHOC_Extract(salt, IKM) = KMAC256(salt, IKM, 512, "")

where the Keccak message authentication code (KMAC) is specified in [SP800-185].

The rest of the section defines the pseudorandom keys PRK_2e, PRK_3e2m and PRK_4e3m; their use is shown in Figure 8. The index of a PRK indicates its use or in what message protection operation it is involved. For example, PRK_3e2m is involved in the encryption of message 3 and in calculating the MAC of message 2.

4.1.1.1. PRK_2e

The pseudorandom key PRK_2e is derived with the following input:

* The salt SHALL be TH_2.

* The IKM SHALL be the ephemeral-ephemeral ECDH shared secret G_XY (calculated from G_X and Y or G_Y and X) as defined in Section 6.3.1 of [RFC9053]. The use of G_XY gives forward secrecy, in the sense that compromise of the private authentication keys does not compromise past session keys.

Example: Assuming the use of curve25519, the ECDH shared secret G_XY is the output of the X25519 function [RFC7748]:

```
G_XY = X25519( Y, G_X ) = X25519( X, G_Y )
```

Example: Assuming the use of SHA-256 the extract phase of HKDF produces PRK_2e as follows:

```
PRK_2e = HMAC-SHA-256( TH_2, G_XY )
```

4.1.1.2. PRK_3e2m

The pseudorandom key PRK_3e2m is derived as follows:

If the Responder authenticates with a static Diffie-Hellman key, then PRK_3e2m = EDHOC_Extract(SALT_3e2m, G_RX), where

* SALT_3e2m is derived from PRK_2e, see Section 4.1.2, and

* G_RX is the ECDH shared secret calculated from G_R and X, or G_X and R (the Responder's private authentication key, see Section 3.5.1),

else PRK_3e2m = PRK_2e.

4.1.1.3. PRK_4e3m

The pseudorandom key PRK_4e3m is derived as follows:

If the Initiator authenticates with a static Diffie-Hellman key, then PRK_4e3m = EDHOC_Extract(SALT_4e3m, G_IY), where

* SALT_4e3m is derived from PRK_3e2m, see Section 4.1.2, and

* G_IY is the ECDH shared secret calculated from G_I and Y, or G_Y and I (the Initiator's private authentication key, see Section 3.5.1),

else PRK_4e3m = PRK_3e2m.

4.1.2. EDHOC_Expand and EDHOC_KDF

The output keying material (OKM) - including keys, IVs, and salts - are derived from the PRKs using the EDHOC_KDF, which is defined through EDHOC_Expand:

```
OKM = EDHOC_KDF( PRK, info_label, context, length )
    = EDHOC_Expand( PRK, info, length )
```

where info is encoded as the CBOR sequence

```
info = (
  info_label : int,
  context : bstr,
  length : uint,
)
```

where

* info_label is an int

* context is a bstr

* length is the length of OKM in bytes

When EDHOC_KDF is used to derive OKM for EDHOC message processing, then context includes one of the transcript hashes TH_2, TH_3, or TH_4 defined in Sections 5.3.2 and 5.4.2.

The definition of EDHOC_Expand depends on the EDHOC hash algorithm of the selected cipher suite:

* if the EDHOC hash algorithm is SHA-2, then EDHOC_Expand(PRK, info, length) = HKDF-Expand(PRK, info, length) [RFC5869]

* if the EDHOC hash algorithm is SHAKE128, then EDHOC_Expand(PRK, info, length) = KMAC128(PRK, info, L, "")

* if the EDHOC hash algorithm is SHAKE256, then EDHOC_Expand(PRK, info, length) = KMAC256(PRK, info, L, "")

where L = 8 length, the output length in bits.

Figure 8 lists derivations made with EDHOC_KDF, where

* hash_length - length of output size of the EDHOC hash algorithm of the selected cipher suite

* key_length - length of the encryption key of the EDHOC AEAD algorithm of the selected cipher suite

* iv_length - length of the initialization vector of the EDHOC AEAD algorithm of the selected cipher suite

Further details of the key derivation and how the output keying material is used are specified in Section 5.

```
KEYSTREAM_2   = EDHOC_KDF( PRK_2e,   0, TH_2,      plaintext_length )
SALT_3e2m     = EDHOC_KDF( PRK_2e,   1, TH_2,      hash_length )
MAC_2         = EDHOC_KDF( PRK_3e2m, 2, context_2, mac_length_2 )
K_3           = EDHOC_KDF( PRK_3e2m, 3, TH_3,      key_length )
IV_3          = EDHOC_KDF( PRK_3e2m, 4, TH_3,      iv_length )
SALT_4e3m     = EDHOC_KDF( PRK_3e2m, 5, TH_3,      hash_length )
MAC_3         = EDHOC_KDF( PRK_4e3m, 6, context_3, mac_length_3 )
PRK_out       = EDHOC_KDF( PRK_4e3m, 7, TH_4,      hash_length )
K_4           = EDHOC_KDF( PRK_4e3m, 8, TH_4,      key_length )
IV_4          = EDHOC_KDF( PRK_4e3m, 9, TH_4,      iv_length )
PRK_exporter  = EDHOC_KDF( PRK_out, 10, h'',       hash_length )
```

Figure 8: Key derivations using EDHOC_KDF. h'' is CBOR diagnostic notation for the empty byte string, 0x40.

4.1.3. PRK_out

The pseudorandom key PRK_out, derived as shown in Figure 8, is the output session key of a completed EDHOC session.

Keys for applications are derived using EDHOC_Exporter (see Section 4.2.1) from PRK_exporter, which in turn is derived from PRK_out as shown in Figure 8. For the purpose of generating application keys, it is sufficient to store PRK_out or PRK_exporter. (Note that the word "store" used here does not imply that the application has access to the plaintext PRK_out since that may be reserved for code within a Trusted Execution Environment, see Section 9.8).

4.2. Keys for EDHOC Applications

This section defines EDHOC_Exporter in terms of EDHOC_KDF and PRK_exporter. A key update function is defined in Appendix H.

4.2.1. EDHOC_Exporter

Keying material for the application can be derived using the EDHOC_Exporter interface defined as:

```
EDHOC_Exporter(exporter_label, context, length)
  = EDHOC_KDF(PRK_exporter, exporter_label, context, length)
```

where

* exporter_label is a registered uint from the EDHOC_Exporter Label registry (Section 10.1)

* context is a bstr defined by the application

* length is a uint defined by the application

The (exporter_label, context) pair used in EDHOC_Exporter must be unique, i.e., an (exporter_label, context) MUST NOT be used for two different purposes. However, an application can re-derive the same key several times as long as it is done securely. For example, in most encryption algorithms the same key can be reused with different nonces. The context can for example be the empty CBOR byte string.

Examples of use of the EDHOC_Exporter are given in Appendix A.

5. Message Formatting and Processing

This section specifies formatting of the messages and processing steps. Error messages are specified in Section 6. Annotated traces of EDHOC sessions are provided in [I-D.ietf-lake-traces].

An EDHOC message is encoded as a sequence of CBOR data items (CBOR Sequence, [RFC8742]). Additional optimizations are made to reduce message overhead.

While EDHOC uses the COSE_Key, COSE_Sign1, and COSE_Encrypt0 structures, only a subset of the parameters is included in the EDHOC messages, see Appendix C.3. In order to recreate the COSE object, the recipient endpoint may need to add parameters to the COSE headers not included in the EDHOC message, for example the parameter 'alg' to COSE_Sign1 or COSE_Encrypt0.

5.1. EDHOC Message Processing Outline

For each new/ongoing EDHOC session, the endpoints are assumed to keep an associated protocol state containing identifiers, keying material, etc. used for subsequent processing of protocol related data. The protocol state is assumed to be associated with an application profile (Section 3.9) which provides the context for how messages are transported, identified, and processed.

EDHOC messages SHALL be processed according to the current protocol state. The following steps are expected to be performed at reception of an EDHOC message:

1. Detect that an EDHOC message has been received, for example by means of port number, URI, or media type (Section 3.9).

2. Retrieve the protocol state according to the message correlation, see Section 3.4.1. If there is no protocol state, in the case of message_1, a new protocol state is created. The Responder endpoint needs to make use of available denial-of-service mitigation (Section 9.7).

3. If the message received is an error message, then process it according to Section 6, else process it as the expected next message according to the protocol state.

The message processing steps SHALL be processed in order, unless otherwise stated. If the processing fails for some reason then, typically, an error message is sent, the EDHOC session is aborted, and the protocol state erased. When the composition and sending of one message is completed and before the next message is received, error messages SHALL NOT be sent.

After having successfully processed the last message (message_3 or message_4 depending on application profile) the EDHOC session is completed, after which no error messages are sent and EDHOC session output MAY be maintained even if error messages are received. Further details are provided in the following subsections and in Section 6.

Different instances of the same message MUST NOT be processed in one EDHOC session. Note that processing will fail if the same message appears a second time for EDHOC processing in the same EDHOC session because the state of the protocol has moved on and now expects something else. Message deduplication MUST be done by the transport protocol (see Section 3.4) or, if not supported by the transport, as described in Section 7.

5.2. EDHOC Message 1

5.2.1. Formatting of Message 1

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

```
message_1 = (
  METHOD : int,
  SUITES_I : suites,
  G_X : bstr,
  C_I : bstr / -24..23,
  ? EAD_1,
)

suites = [ 2* int ] / int
EAD_1 = 1* ead
```

where:

* METHOD - authentication method, see Section 3.2.

* SUITES_I - array of cipher suites which the Initiator supports constructed as specified in Section 5.2.2.

* G_X - the ephemeral public key of the Initiator

* C_I - variable length connection identifier. Note that connection identifiers are byte strings but certain values are represented as integers in the message, see Section 3.3.2.

* EAD_1 - external authorization data, see Section 3.8.

5.2.2. Initiator Composition of Message 1

The processing steps are detailed below and in Section 6.3.

The Initiator SHALL compose message_1 as follows:

* Construct SUITES_I as an array of cipher suites supported by I in order of preference by I with the first cipher suite in the array being the most preferred by I, and the last being the one selected by I for this EDHOC session. If the cipher suite most preferred by I is selected then SUITES_I contains only that cipher suite and is encoded as an int. All cipher suites, if any, preferred by I over the selected one MUST be included. (See also Section 6.3.)

- The selected suite is based on what the Initiator can assume to be supported by the Responder; if the Initiator previously received from the Responder an error message with error code 2 containing SUITES_R (see Section 6.3) indicating cipher suites supported by the Responder, then the Initiator SHOULD select its most preferred supported cipher suite among those (bearing in mind that error messages may be forged).

- The Initiator MUST NOT change its order of preference for cipher suites, and MUST NOT omit a cipher suite preferred to the selected one because of previous error messages received from the Responder.

* Generate an ephemeral ECDH key pair using the curve in the selected cipher suite and format it as a COSE_Key. Let G_X be the 'x' parameter of the COSE_Key.

* Choose a connection identifier C_I and store it during the EDHOC session.

* Encode message_1 as a sequence of CBOR encoded data items as specified in Section 5.2.1

5.2.3. Responder Processing of Message 1

The Responder SHALL process message_1 in the following order:

* Decode message_1 (see Appendix C.1).

* Process message_1, in particular verify that the selected cipher suite is supported and that no prior cipher suite as ordered in SUITES_I is supported.

* If all processing completed successfully, and if EAD_1 is present, then make it available to the application for EAD processing.

If any processing step fails, then the Responder MUST send an EDHOC error message back as defined in Section 6, and the EDHOC session MUST be aborted.

5.3. EDHOC Message 2

5.3.1. Formatting of Message 2

message_2 SHALL be a CBOR Sequence (see Appendix C.1) as defined below

```
message_2 = (
  G_Y_CIPHERTEXT_2 : bstr,
)
```

where:

* G_Y_CIPHERTEXT_2 - the concatenation of G_Y (i.e., the ephemeral public key of the Responder) and CIPHERTEXT_2.

5.3.2. Responder Composition of Message 2

The Responder SHALL compose message_2 as follows:

* Generate an ephemeral ECDH key pair using the curve in the selected cipher suite and format it as a COSE_Key. Let G_Y be the 'x' parameter of the COSE_Key.

* Choose a connection identifier C_R and store it for the length of the EDHOC session.

* Compute the transcript hash TH_2 = H(G_Y, H(message_1)) where H() is the EDHOC hash algorithm of the selected cipher suite. The input to the hash function is a CBOR Sequence. Note that H(message_1) can be computed and cached already in the processing of message_1.

* Compute MAC_2 as in Section 4.1.2 with context_2 = << C_R, ID_CRED_R, TH_2, CRED_R, ? EAD_2 >> (see Appendix C.1 for notation)

  - If the Responder authenticates with a static Diffie-Hellman key (method equals 1 or 3), then mac_length_2 is the EDHOC MAC length of the selected cipher suite. If the Responder authenticates with a signature key (method equals 0 or 2), then mac_length_2 is equal to hash_length.

  - C_R - variable length connection identifier. Note that connection identifiers are byte strings but certain values are represented as integers in the message, see Section 3.3.2.

  - ID_CRED_R - identifier to facilitate the retrieval of CRED_R, see Section 3.5.3

  - CRED_R - CBOR item containing the authentication credential of the Responder, see Section 3.5.2

  - EAD_2 - external authorization data, see Section 3.8

* If the Responder authenticates with a static Diffie-Hellman key (method equals 1 or 3), then Signature_or_MAC_2 is MAC_2. If the Responder authenticates with a signature key (method equals 0 or 2), then Signature_or_MAC_2 is the 'signature' field of a COSE_Sign1 object, computed as specified in Section 4.4 of [RFC9053] using the signature algorithm of the selected cipher suite, the private authentication key of the Responder, and the following parameters as input (see Appendix C.3 for an overview of COSE and Appendix C.1 for notation):

  - protected = << ID_CRED_R >>

  - external_aad = << TH_2, CRED_R, ? EAD_2 >>

  - payload = MAC_2

* CIPHERTEXT_2 is calculated with a binary additive stream cipher, using a keystream generated with EDHOC_Expand, and the following plaintext:

  - PLAINTEXT_2 = (C_R, ID_CRED_R / bstr / -24..23, Signature_or_MAC_2, ? EAD_2)

    - If ID_CRED_R contains a single 'kid' parameter, i.e., ID_CRED_R = { 4 : kid_R }, then the compact encoding is applied, see Section 3.5.3.2.

    - C_R - variable length connection identifier. Note that connection identifiers are byte strings but certain values are represented as integers in the message, see Section 3.3.2.

  - Compute KEYSTREAM_2 as in Section 4.1.2, where plaintext_length is the length of PLAINTEXT_2. For the case of plaintext_length exceeding the EDHOC_KDF output size, see Appendix G.

  - CIPHERTEXT_2 = PLAINTEXT_2 XOR KEYSTREAM_2

* Encode message_2 as a sequence of CBOR encoded data items as specified in Section 5.3.1.

5.3.3. Initiator Processing of Message 2

The Initiator SHALL process message_2 in the following order:

* Decode message_2 (see Appendix C.1).

* Retrieve the protocol state using available message correlation (e.g., the CoAP Token, the 5-tuple, or the prepended C_I, see Section 3.4.1).

* Decrypt CIPHERTEXT_2, see Section 5.3.2.

* If all processing completed successfully, then make ID_CRED_R and (if present) EAD_2 available to the application for authentication- and EAD processing. When and how to perform authentication is up to the application.

* Obtain the authentication credential (CRED_R) and the authentication key of R from the application (or by other means).

* Verify Signature_or_MAC_2 using the algorithm in the selected cipher suite. The verification process depends on the method, see Section 5.3.2. Make the result of the verification available to the application.

If any processing step fails, then the Initiator MUST send an EDHOC error message back as defined in Section 6, and the EDHOC session MUST be aborted.

5.4. EDHOC Message 3

5.4.1. Formatting of Message 3

message_3 SHALL be a CBOR Sequence (see Appendix C.1) as defined below

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

5.4.2. Initiator Composition of Message 3

The Initiator SHALL compose message_3 as follows:

* Compute the transcript hash TH_3 = H(TH_2, PLAINTEXT_2, CRED_R) where H() is the EDHOC hash algorithm of the selected cipher suite. The input to the hash function is a CBOR Sequence. Note that TH_3 can be computed and cached already in the processing of message_2.

* Compute MAC_3 as in Section 4.1.2, with context_3 = << ID_CRED_I, TH_3, CRED_I, ? EAD_3 >>

- If the Initiator authenticates with a static Diffie-Hellman key (method equals 2 or 3), then mac_length_3 is the EDHOC MAC length of the selected cipher suite. If the Initiator authenticates with a signature key (method equals 0 or 1), then mac_length_3 is equal to hash_length.

- ID_CRED_I - identifier to facilitate the retrieval of CRED_I, see Section 3.5.3

- CRED_I - CBOR item containing the authentication credential of the Initiator, see Section 3.5.2

- EAD_3 - external authorization data, see Section 3.8

* If the Initiator authenticates with a static Diffie-Hellman key (method equals 2 or 3), then Signature_or_MAC_3 is MAC_3. If the Initiator authenticates with a signature key (method equals 0 or 1), then Signature_or_MAC_3 is the 'signature' field of a COSE_Sign1 object, computed as specified in Section 4.4 of [RFC9052] using the signature algorithm of the selected cipher suite, the private authentication key of the Initiator, and the following parameters as input (see Appendix C.3):

  - protected = << ID_CRED_I >>

  - external_aad = << TH_3, CRED_I, ? EAD_3 >>

  - payload = MAC_3

* Compute a COSE_Encrypt0 object as defined in Sections 5.2 and 5.3 of [RFC9052], with the EDHOC AEAD algorithm of the selected cipher suite, using the encryption key K_3, the initialization vector IV_3 (if used by the AEAD algorithm), the plaintext PLAINTEXT_3, and the following parameters as input (see Appendix C.3):

  - protected = h''

  - external_aad = TH_3

  - K_3 and IV_3 are defined in Section 4.1.2

  - PLAINTEXT_3 = (ID_CRED_I / bstr / -24..23, Signature_or_MAC_3, ? EAD_3)

    o If ID_CRED_I contains a single 'kid' parameter, i.e., ID_CRED_I = { 4 : kid_I }, then the compact encoding is applied, see Section 3.5.3.2.

CIPHERTEXT_3 is the 'ciphertext' of COSE_Encrypt0.

* Compute the transcript hash TH_4 = H(TH_3, PLAINTEXT_3, CRED_I) where H() is the EDHOC hash algorithm of the selected cipher suite. The input to the hash function is a CBOR Sequence.

* Calculate PRK_out as defined in Figure 8. The Initiator can now derive application keys using the EDHOC_Exporter interface, see Section 4.2.1.

* Encode message_3 as a CBOR data item as specified in Section 5.4.1.

* Make the connection identifiers (C_I, C_R) and the application algorithms in the selected cipher suite available to the application.

After creating message_3, the Initiator can compute PRK_out, see Section 4.1.3, and derive application keys using the EDHOC_Exporter interface, see Section 4.2.1. The Initiator SHOULD NOT persistently store PRK_out or application keys until the Initiator has verified message_4 or a message protected with a derived application key, such as an OSCORE message, from the Responder and the application has authenticated the Responder. This is similar to waiting for an acknowledgment (ACK) in a transport protocol. The Initiator SHOULD NOT send protected application data until the application has authenticated the Responder.

5.4.3. Responder Processing of Message 3

The Responder SHALL process message_3 in the following order:

* Decode message_3 (see Appendix C.1).

* Retrieve the protocol state using available message correlation (e.g., the CoAP Token, the 5-tuple, or the prepended C_R, see Section 3.4.1).

* Decrypt and verify the COSE_Encrypt0 as defined in Sections 5.2 and 5.3 of [RFC9052], with the EDHOC AEAD algorithm in the selected cipher suite, and the parameters defined in Section 5.4.2.

* If all processing completed successfully, then make ID_CRED_I and (if present) EAD_3 available to the application for authentication- and EAD processing. When and how to perform authentication is up to the application.

* Obtain the authentication credential (CRED_I) and the authentication key of I from the application (or by other means).

* Verify Signature_or_MAC_3 using the algorithm in the selected cipher suite. The verification process depends on the method, see Section 5.4.2. Make the result of the verification available to the application.

* Make the connection identifiers (C_I, C_R) and the application algorithms in the selected cipher suite available to the application.

After processing message_3, the Responder can compute PRK_out, see Section 4.1.3, and derive application keys using the EDHOC_Exporter interface, see Section 4.2.1. The Responder SHOULD NOT persistently store PRK_out or application keys until the application has authenticated the Initiator. The Responder SHOULD NOT send protected application data until the application has authenticated the Initiator.

If any processing step fails, then the Responder MUST send an EDHOC error message back as defined in Section 6, and the EDHOC session MUST be aborted.

5.5. EDHOC Message 4

This section specifies message_4 which is OPTIONAL to support. Key confirmation is normally provided by sending an application message from the Responder to the Initiator protected with a key derived with the EDHOC_Exporter, e.g., using OSCORE (see Appendix A). In deployments where no protected application message is sent from the Responder to the Initiator, message_4 MUST be supported and MUST be used. Two examples of such deployments are:

1. When EDHOC is only used for authentication and no application data is sent.

2. When application data is only sent from the Initiator to the Responder.

Further considerations about when to use message_4 are provided in Section 3.9 and Section 9.1.

5.5.1. Formatting of Message 4

message_4 SHALL be a CBOR Sequence (see Appendix C.1) as defined below

```
message_4 = (
  CIPHERTEXT_4 : bstr,
)
```

5.5.2. Responder Composition of Message 4

The Responder SHALL compose message_4 as follows:

* Compute a COSE_Encrypt0 as defined in Sections 5.2 and 5.3 of [RFC9052], with the EDHOC AEAD algorithm of the selected cipher suite, using the encryption key K_4, the initialization vector IV_4 (if used by the AEAD algorithm), the plaintext PLAINTEXT_4, and the following parameters as input (see Appendix C.3):

  - protected = h''

  - external_aad = TH_4

  - K_4 and IV_4 are defined in Section 4.1.2

  - PLAINTEXT_4 = (? EAD_4)

    - EAD_4 - external authorization data, see Section 3.8.

  CIPHERTEXT_4 is the 'ciphertext' of COSE_Encrypt0.

* Encode message_4 as a CBOR data item as specified in Section 5.5.1.

5.5.3. Initiator Processing of Message 4

The Initiator SHALL process message_4 as follows:

* Decode message_4 (see Appendix C.1).

* Retrieve the protocol state using available message correlation (e.g., the CoAP Token, the 5-tuple, or the prepended C_I, see Section 3.4.1).

* Decrypt and verify the COSE_Encrypt0 as defined in Sections 5.2 and 5.3 of [RFC9052], with the EDHOC AEAD algorithm in the selected cipher suite, and the parameters defined in Section 5.5.2.

* Make (if present) EAD_4 available to the application for EAD processing.

If any processing step fails, then the Initiator MUST send an EDHOC error message back as defined in Section 6, and the EDHOC session MUST be aborted.

After verifying message_4, the Initiator is assured that the Responder has calculated the key PRK_out (key confirmation) and that no other party can derive the key.

6. Error Handling

This section defines the format for error messages, and the processing associated with the currently defined error codes. Additional error codes may be registered, see Section 10.4.

Many kinds of errors that can occur during EDHOC processing. As in CoAP, an error can be triggered by errors in the received message or internal errors in the receiving endpoint. Except for processing and formatting errors, it is up to the application when to send an error message. Sending error messages is essential for debugging but MAY be skipped if, for example, an EDHOC session cannot be found or due to denial-of-service reasons, see Section 9.7. Error messages in EDHOC are always fatal. After sending an error message, the sender MUST abort the EDHOC session. The receiver SHOULD treat an error message as an indication that the other party likely has aborted the EDHOC session. But since error messages might be forged, the receiver MAY try to continue the EDHOC session.

An EDHOC error message can be sent by either endpoint as a reply to any non-error EDHOC message. How errors at the EDHOC layer are transported depends on lower layers, which need to enable error messages to be sent and processed as intended.

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

```
error = (
  ERR_CODE : int,
  ERR_INFO : any,
)
```

Figure 9: EDHOC error message.

where:

* ERR_CODE - error code encoded as an integer. The value 0 is reserved for success and can only be used internally, all other values (negative or positive) indicate errors.

* ERR_INFO - error information. Content and encoding depend on error code.

The remainder of this section specifies the currently defined error codes, see Figure 10. Additional error codes and corresponding error information may be specified.

ERR_CODE	ERR_INFO Type	Description
0		This value is reserved
1	tstr	Unspecified error
2	suites	Wrong selected cipher suite
3	true	Unknown credential referenced

Figure 10: EDHOC error codes and error information.

6.1. Success

Error code 0 MAY be used internally in an application to indicate success, i.e., as a standard value in case of no error, e.g., in status reporting or log files. Error code 0 MUST NOT be used as part of the EDHOC message exchange. If an endpoint receives an error message with error code 0, then it MUST abort the EDHOC session and MUST NOT send an error message.

6.2. Unspecified Error

Error code 1 is used for errors that do not have a specific error code defined. ERR_INFO MUST be a text string containing a human-readable diagnostic message which SHOULD be written in English, for example "Method not supported". The diagnostic text message is mainly intended for software engineers that during debugging need to interpret it in the context of the EDHOC specification. The diagnostic message SHOULD be provided to the calling application where it SHOULD be logged.

6.3. Wrong Selected Cipher Suite

Error code 2 MUST only be used when replying to message_1 in case the cipher suite selected by the Initiator is not supported by the Responder, or if the Responder supports a cipher suite more preferred by the Initiator than the selected cipher suite, see Section 5.2.3. In this case, ERR_INFO = SUITES_R and is of type suites, see Section 5.2.1. If the Responder does not support the selected cipher suite, then SUITES_R MUST include one or more supported cipher suites. If the Responder supports a cipher suite in SUITES_I other than the selected cipher suite (independently of if the selected cipher suite is supported or not) then SUITES_R MUST include the supported cipher suite in SUITES_I which is most preferred by the Initiator. SUITES_R MAY include a single cipher suite, in which case it is encoded as an int. If the Responder does not support any cipher suite in SUITES_I, then it SHOULD include all its supported cipher suites in SUITES_R.

In contrast to SUITES_I, the order of the cipher suites in SUITES_R has no significance.

6.3.1. Cipher Suite Negotiation

After receiving SUITES_R, the Initiator can determine which cipher suite to select (if any) for the next EDHOC run with the Responder.

If the Initiator intends to contact the Responder in the future, the Initiator SHOULD remember which selected cipher suite to use until the next message_1 has been sent, otherwise the Initiator and Responder will likely run into an infinite loop where the Initiator selects its most preferred cipher suite and the Responder sends an error with supported cipher suites. After a completed EDHOC session, the Initiator MAY remember the selected cipher suite to use in future EDHOC sessions. Note that if the Initiator or Responder is updated with new cipher suite policies, any cached information may be outdated.

Note that the Initiator’s list of supported cipher suites and order of preference is fixed (see Section 5.2.1 and Section 5.2.2). Furthermore, the Responder SHALL only accept message_1 if the selected cipher suite is the first cipher suite in SUITES_I that the Responder also supports (see Section 5.2.3). Following this procedure ensures that the selected cipher suite is the most preferred (by the Initiator) cipher suite supported by both parties. For examples, see Section 6.3.2.

If the selected cipher suite is not the first cipher suite which the Responder supports in SUITES_I received in message_1, then the Responder MUST abort the EDHOC session, see Section 5.2.3. If SUITES_I in message_1 is manipulated, then the integrity verification of message_2 containing the transcript hash TH_2 will fail and the Initiator will abort the EDHOC session.

6.3.2. Examples

Assume that the Initiator supports the five cipher suites 5, 6, 7, 8, and 9 in decreasing order of preference. Figures 11 and 12 show two examples of how the Initiator can format SUITES_I and how SUITES_R is used by Responders to give the Initiator information about the cipher suites that the Responder supports.

In Example 1 (Figure 11), the Responder supports cipher suite 6 but not the initially selected cipher suite 5. The Responder rejects the first message_1 with an error indicating support for suite 6 in SUITES_R. The Initiator also supports suite 6, and therefore selects suite 6 in the second message_1. The Initiator prepends in SUITES_I the selected suite 6 with the more preferred suites, in this case suite 5, to mitigate a potential attack on the cipher suite negotiation.

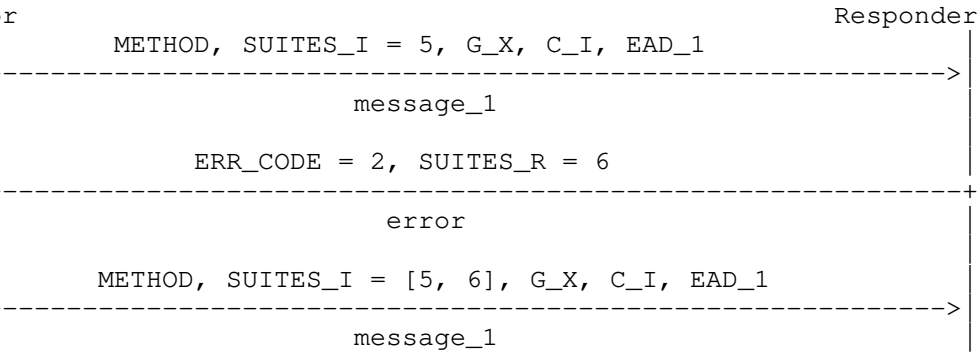


Figure 11: Cipher suite negotiation example 1.

In Example 2 (Figure 12), the Responder supports cipher suites 8 and 9 but not the more preferred (by the Initiator) cipher suites 5, 6 or 7. To illustrate the negotiation mechanics we let the Initiator first make a guess that the Responder supports suite 6 but not suite 5. Since the Responder supports neither 5 nor 6, it rejects the first message_1 with an error indicating support for suites 8 and 9 in SUITES_R (in any order). The Initiator also supports suites 8 and 9, and prefers suite 8, so selects suite 8 in the second message_1. The Initiator prepends in SUITES_I the selected suite 8 with the more preferred suites in order of preference, in this case suites 5, 6 and 7, to mitigate a potential attack on the cipher suite negotiation.

Note 1. If the Responder had supported suite 5, then the first message_1 would not have been accepted either, since the Responder observes that suite 5 is more preferred by the Initiator than the selected suite 6. In that case the Responder would have included suite 5 in SUITES_R of the response, and it would then have become the selected and only suite in the second message_1.

Note 2. For each message_1 the Initiator MUST generate a new ephemeral ECDH key pair matching the selected cipher suite.

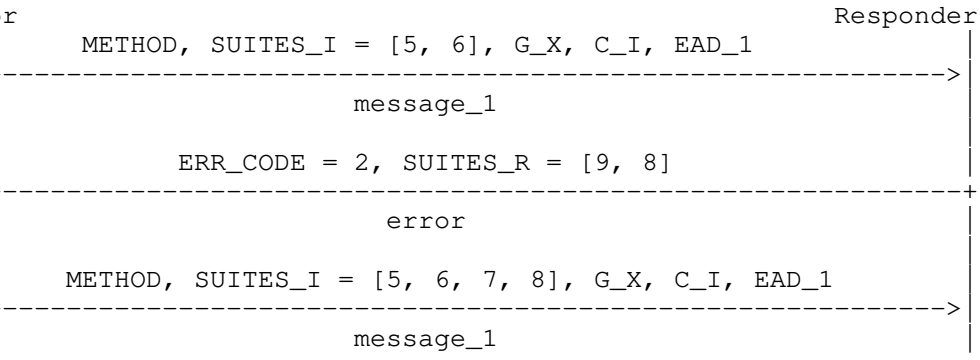


Figure 12: Cipher suite negotiation example 2.

6.4. Unknown Credential Referenced

Error code 3 is used for errors due to a received credential identifier (ID_CRED_R in message_2 or ID_CRED_I message_3) containing a reference to a credential which the receiving endpoint does not have access to. The intent with this error code is that the endpoint who sent the credential identifier should for the next EDHOC session try another credential identifier supported according to the application profile.

For example, an application profile could list x5t and x5chain as supported credential identifiers, and state that x5t should be used if it can be assumed that the X.509 certificate is available at the receiving side. This error code thus enables the certificate chain to be sent only when needed, bearing in mind that error messages are not protected so an adversary can try to cause unnecessary large credential identifiers.

For the error code 3, the error information SHALL be the CBOR simple value true (0xf5). Error code 3 MUST NOT be used when the received credential identifier type is not supported.

7. EDHOC Message Deduplication

EDHOC by default assumes that message duplication is handled by the transport, in this section exemplified with CoAP, see Appendix A.2.

Deduplication of CoAP messages is described in Section 4.5 of [RFC7252]. This handles the case when the same Confirmable (CON) message is received multiple times due to missing acknowledgment on the CoAP messaging layer. The recommended processing in [RFC7252] is that the duplicate message is acknowledged (ACK), but the received message is only processed once by the CoAP stack.

Message deduplication is resource demanding and therefore not supported in all CoAP implementations. Since EDHOC is targeting constrained environments, it is desirable that EDHOC can optionally support transport layers which do not handle message duplication. Special care is needed to avoid issues with duplicate messages, see Section 5.1.

The guiding principle here is similar to the deduplication processing on the CoAP messaging layer: a received duplicate EDHOC message SHALL NOT result in another instance of the next EDHOC message. The result MAY be that a duplicate next EDHOC message is sent, provided it is still relevant with respect to the current protocol state. In any case, the received message MUST NOT be processed more than once in the same EDHOC session. This is called "EDHOC message deduplication".

An EDHOC implementation MAY store the previously sent EDHOC message to be able to resend it.

In principle, if the EDHOC implementation would deterministically regenerate the identical EDHOC message previously sent, it would be possible to instead store the protocol state to be able to recreate and resend the previously sent EDHOC message. However, even if the protocol state is fixed, the message generation may introduce

differences which compromise security. For example, in the generation of message_3, if I is performing a (non-deterministic) ECDSA signature (say, method 0 or 1, cipher suite 2 or 3) then PLAINTEXT_3 is randomized, but K_3 and IV_3 are the same, leading to a key and nonce reuse.

The EDHOC implementation MUST NOT store previous protocol state and regenerate an EDHOC message if there is a risk that the same key and IV are used for two (or more) distinct messages.

The previous message or protocol state MUST NOT be kept longer than what is required for retransmission, for example, in the case of CoAP transport, no longer than the EXCHANGE_LIFETIME (see Section 4.8.2 of [RFC7252]).

8. Compliance Requirements

In the absence of an application profile specifying otherwise:

An implementation MAY support only Initiator or only Responder.

An implementation MAY support only a single method. None of the methods are mandatory-to-implement.

Implementations MUST support 'kid' parameters. None of the other COSE header parameters are mandatory-to-implement.

An implementation MAY support only a single credential type (CCS, CWT, X.509, C509). None of the credential types are mandatory-to-implement.

Implementations MUST support the EDHOC_Exporter.

Implementations MAY support message_4. Error codes (ERR_CODE) 1 and 2 MUST be supported.

Implementations MUST support EAD.

Implementations MUST support cipher suite 2 and 3. Cipher suites 2 (AES-CCM-16-64-128, SHA-256, 8, P-256, ES256, AES-CCM-16-64-128, SHA-256) and 3 (AES-CCM-16-128-128, SHA-256, 16, P-256, ES256, AES-CCM-16-64-128, SHA-256) only differ in the size of the MAC length, so supporting one or both of these is not significantly different. Implementations only need to implement the algorithms needed for their supported methods.

9. Security Considerations

9.1. Security Properties

EDHOC has similar security properties as can be expected from the theoretical SIGMA-I protocol [SIGMA] and the Noise XX pattern [Noise], which are similar to methods 0 and 3, respectively. Proven security properties are detailed in the security analysis publications referenced at the end of this section.

Using the terminology from [SIGMA], EDHOC provides forward secrecy, mutual authentication with aliveness, consistency, and peer awareness. As described in [SIGMA], message_3 provides peer awareness to the Responder while message_4 provides peer awareness to the Initiator. By including the authentication credentials in the transcript hash, EDHOC protects against Duplicate Signature Key Selection (DSKS)-like identity mis-binding attack that the MAC-then-Sign variant of SIGMA-I is otherwise vulnerable to.

As described in [SIGMA], different levels of identity protection are provided to the Initiator and the Responder. EDHOC provides identity protection of the Initiator against active attacks and identity protection of the Responder against passive attacks. An active attacker can get the credential identifier of the Responder by eavesdropping on the destination address used for transporting message_1 and then sending its own message_1 to the same address. The roles should be assigned to protect the most sensitive identity/identifier, typically that which is not possible to infer from routing information in the lower layers.

EDHOC messages might change in transit due to a noisy channel or through modification by an attacker. Changes in message_1 and message_2 (except Signature_or_MAC_2 when the signature scheme is not strongly unforgeable) are detected when verifying Signature_or_MAC_2. Changes to not strongly unforgeable Signature_or_MAC_2, and message_3 are detected when verifying CIPHERTEXT_3. Changes to message_4 are detected when verifying CIPHERTEXT_4.

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

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

As required by [RFC7258], IETF protocols need to mitigate pervasive monitoring when possible. EDHOC therefore only supports methods with ephemeral Diffie-Hellman and provides a key update function (see Appendix H) for lightweight application protocol rekeying. Either of these provides forward secrecy, in the sense that compromise of the private authentication keys does not compromise past session keys (PRK_out), and compromise of a session key does not compromise past session keys. Frequently re-running EDHOC with ephemeral Diffie-Hellman forces attackers to perform dynamic key exfiltration where the attacker must have continuous interactions with the collaborator, which is a significant sustained attack.

To limit the effect of breaches, it is important to limit the use of symmetric group keys for bootstrapping. EDHOC therefore strives to make the additional cost of using raw public keys and self-signed certificates as small as possible. Raw public keys and self-signed certificates are not a replacement for a public key infrastructure but SHOULD be used instead of symmetric group keys for bootstrapping.

Compromise of the long-term keys (private signature or static DH keys) does not compromise the security of completed EDHOC sessions. Compromising the private authentication keys of one party lets an active attacker impersonate that compromised party in EDHOC sessions with other parties but does not let the attacker impersonate other parties in EDHOC sessions with the compromised party. Compromise of the long-term keys does not enable a passive attacker to compromise future session keys (PRK_out). Compromise of the HDKF input parameters (ECDH shared secret) leads to compromise of all session keys derived from that compromised shared secret. Compromise of one session key does not compromise other session keys. Compromise of PRK_out leads to compromise of all keying material derived with the EDHOC_Exporter.

Based on the cryptographic algorithms requirements Section 9.3, EDHOC provides a minimum of 64-bit security against online brute force attacks and a minimum of 128-bit security against offline brute force attacks. To break 64-bit security against online brute force an attacker would on average have to send 4.3 billion messages per second for 68 years, which is infeasible in constrained IoT radio

technologies. A forgery against a 64-bit MAC in EDHOC breaks the security of all future application data, while a forgery against a 64-bit MAC in the subsequent application protocol (e.g., OSCORE [RFC8613]) typically only breaks the security of the data in the forged packet.

As the EDHOC session is aborted when verification fails, the security against online attacks is given by the sum of the strength of the verified signatures and MACs (including MAC in AEAD). As an example, if EDHOC is used with method 3, cipher suite 2, and message_4, the Responder is authenticated with 128-bit security against online attacks (the sum of the 64-bit MACs in message_2 and message_4). The same principle applies for MACs in an application protocol keyed by EDHOC as long as EDHOC is rerun when verification of the first MACs in the application protocol fails. As an example, if EDHOC with method 3 and cipher suite 2 is used as in Figure 2 of [I-D.ietf-core-oscore-edhoc], 128-bit mutual authentication against online attacks can be achieved after completion of the first OSCORE request and response.

After sending message_3, the Initiator is assured that no other party than the Responder can compute the key PRK_out. While the Initiator can securely send protected application data, the Initiator SHOULD NOT persistently store the keying material PRK_out until the Initiator has verified message_4 or a message protected with a derived application key, such as an OSCORE message, from the Responder. After verifying message_3, the Responder is assured that an honest Initiator has computed the key PRK_out. The Responder can securely derive and store the keying material PRK_out, and send protected application data.

External authorization data sent in message_1 (EAD_1) or message_2 (EAD_2) should be considered unprotected by EDHOC, see Section 9.5. EAD_2 is encrypted but the Responder has not yet authenticated the Initiator and the encryption does not provide confidentiality against active attacks.

External authorization data sent in message_3 (EAD_3) or message_4 (EAD_4) is protected between Initiator and Responder by the protocol, but note that EAD fields may be used by the application before the message verification is completed, see Section 3.8. Designing a secure mechanism that uses EAD is not necessarily straightforward. This document only provides the EAD transport mechanism, but the problem of agreeing on the surrounding context and the meaning of the information passed to and from the application remains. Any new uses of EAD should be subject to careful review.

Key compromise impersonation (KCI): In EDHOC authenticated with signature keys, EDHOC provides KCI protection against an attacker having access to the long-term key or the ephemeral secret key. With static Diffie-Hellman key authentication, KCI protection would be provided against an attacker having access to the long-term Diffie-Hellman key, but not to an attacker having access to the ephemeral secret key. Note that the term KCI has typically been used for compromise of long-term keys, and that an attacker with access to the ephemeral secret key can only attack that specific EDHOC session.

Repudiation: If an endpoint authenticates with a signature, the other endpoint can prove that the endpoint performed a run of the protocol by presenting the data being signed as well as the signature itself. With static Diffie-Hellman key authentication, the authenticating endpoint can deny having participated in the protocol.

Earlier versions of EDHOC have been formally analyzed [Bruni18] [Norrman20] [CottierPointcheval22] [Jacomme23] [GuentherIlunga22] and the specification has been updated based on the analysis.

9.2. Cryptographic Considerations

The SIGMA protocol requires that the encryption of message_3 provides confidentiality against active attackers and EDHOC message_4 relies on the use of authenticated encryption. Hence, the message authenticating functionality of the authenticated encryption in EDHOC is critical: authenticated encryption MUST NOT be replaced by plain encryption only, even if authentication is provided at another level or through a different mechanism.

To reduce message overhead EDHOC does not use explicit nonces and instead relies on the ephemeral public keys to provide randomness to each EDHOC session. A good amount of randomness is important for the key generation, to provide liveness, and to protect against interleaving attacks. For this reason, the ephemeral keys MUST NOT be used in more than one EDHOC message, and both parties SHALL generate fresh random ephemeral key pairs. Note that an ephemeral key may be used to calculate several ECDH shared secrets. When static Diffie-Hellman authentication is used the same ephemeral key is used in both ephemeral-ephemeral and ephemeral-static ECDH.

As discussed in [SIGMA], the encryption of message_2 does only need to protect against passive attacker as active attackers can always get the Responder's identity by sending their own message_1. EDHOC uses the EDHOC_Expand function (typically HKDF-Expand) as a binary additive stream cipher which is proven secure as long as the expand function is a PRF. HKDF-Expand is not often used as a stream cipher as it is slow on long messages, and most applications require both

confidentiality with indistinguishability under chosen ciphertext (IND-CCA) as well as integrity protection. For the encryption of message_2, any speed difference is negligible, IND-CCA does not increase security, and integrity is provided by the inner MAC (and signature depending on method).

Requirements for how to securely generate, validate, and process the public keys depend on the elliptic curve. For X25519 and X448, the requirements are defined in [RFC7748]. For X25519 and X448, the check for all-zero output as specified in Section 6 of [RFC7748] MUST be done. For secp256r1, secp384r1, and secp521r1, the requirements are defined in Section 5 of [SP-800-56A]. For secp256r1, secp384r1, and secp521r1, at least partial public-key validation MUST be done.

The same authentication credential MAY be used for both the Initiator and Responder roles. As noted in Section 12 of [RFC9052] the use of a single key for multiple algorithms is strongly discouraged unless proven secure by a dedicated cryptographic analysis. In particular this recommendation applies to using the same private key for static Diffie-Hellman authentication and digital signature authentication. A preliminary conjecture is that a minor change to EDHOC may be sufficient to fit the analysis of secure shared signature and ECDH key usage in [Degabriele11] and [Thormarker21].

The property that a completed EDHOC session implies that another identity has been active is upheld as long as the Initiator does not have its own identity in the set of Responder identities it is allowed to communicate with. In Trust on first use (TOFU) use cases, see Appendix D.5, the Initiator should verify that the Responder's identity is not equal to its own. Any future EDHOC methods using e.g., pre-shared keys might need to mitigate this in other ways. However, an active attacker can gain information about the set of identities an Initiator is willing to communicate with. If the Initiator is willing to communicate with all identities except its own an attacker can determine that a guessed Initiator identity is correct. To not leak any long-term identifiers, using a freshly generated authentication key as identity in each initial TOFU session is RECOMMENDED.

NIST SP 800-56A [SP-800-56A] forbids deriving secret and non-secret randomness from the same KDF instance, but this decision has been criticized by Krawczyk [HKDFpaper] and doing so is common practice. In addition to IVs, other examples are the challenge in EAP-TTLS, the RAND in 3GPP AKAs, and the Session-Id in EAP-TLS 1.3. Note that part of KEYSTREAM_2 is also non-secret randomness as it is known or predictable to an attacker. The more recent NIST SP 800-108 [SP-800-108] aligns with [HKDFpaper] and states that for a secure KDF, the revelation of one portion of the derived keying material must not degrade the security of any other portion of that keying material.

9.3. Cipher Suites and Cryptographic Algorithms

When using private cipher suite or registering new cipher suites, the choice of key length used in the different algorithms needs to be harmonized, so that a sufficient security level is maintained for authentication credentials, the EDHOC session, and the protection of application data. The Initiator and the Responder should enforce a minimum security level.

The output size of the EDHOC hash algorithm MUST be at least 256-bits, i.e., the hash algorithms SHA-1 and SHA-256/64 (SHA-256 truncated to 64-bits) SHALL NOT be supported for use in EDHOC except for certificate identification with x5t and c5t. For security considerations of SHA-1, see [RFC6194]. As EDHOC integrity protects the whole authentication credentials, the choice of hash algorithm in x5t and c5t does not affect security, and using the same hash algorithm as in the cipher suite, but with as much truncation as possible, is RECOMMENDED. That is, when the EDHOC hash algorithm is SHA-256, using SHA-256/64 in x5t and c5t is RECOMMENDED. The EDHOC MAC length MUST be at least 8 bytes and the tag length of the EDHOC AEAD algorithm MUST be at least 64-bits. Note that secp256k1 is only defined for use with ECDSA and not for ECDH. Note that some COSE algorithms are marked as not recommended in the COSE IANA registry.

9.4. Post-Quantum Considerations

As of the publication of this specification, it is unclear when or even if a quantum computer of sufficient size and power to exploit public key cryptography will exist. Deployments that need to consider risks decades into the future should transition to Post-Quantum Cryptography (PQC) in the not-too-distant future. Many other systems should take a slower wait-and-see approach where PQC is phased in when the quantum threat is more imminent. Current PQC algorithms have limitations compared to Elliptic Curve Cryptography (ECC) and the data sizes would be problematic in many constrained IoT systems.

Symmetric algorithms used in EDHOC such as SHA-256 and AES-CCM-16-64-128 are practically secure against even large quantum computers. Two of NIST's security levels for quantum-resistant public-key cryptography are based on AES-128 and SHA-256. Quantum computer will likely be expensive, slow due to heavy error correction, and Grovers algorithm, which is proven to be optimal, cannot effectively be parallelized. Grovers algorithm will provide little or no advantage in attacking AES, and AES-128 will remain secure for decades to come [NISTPQC].

EDHOC supports all signature algorithms defined by COSE, including PQC signature algorithms such as HSS-LMS. EDHOC is currently only specified for use with key exchange algorithms of type ECDH curves, but any Key Encapsulation Method (KEM), including PQC KEMs, can be used in method 0. While the key exchange in method 0 is specified with terms of the Diffie-Hellman protocol, the key exchange adheres to a KEM interface: G_X is then the public key of the Initiator, G_Y is the encapsulation, and G_XY is the shared secret. Use of PQC KEMs to replace static DH authentication would likely require a specification updating EDHOC with new methods.

9.5. Unprotected Data and Privacy

The Initiator and the Responder must make sure that unprotected data and metadata do not reveal any sensitive information. This also applies for encrypted data sent to an unauthenticated party. In particular, it applies to EAD_1, ID_CRED_R, EAD_2, and error messages. Using the same EAD_1 in several EDHOC sessions allows passive eavesdroppers to correlate the different sessions. Note that even if ead_value is encrypted outside of EDHOC, the ead_labels in EAD_1 is revealed to passive attackers and the ead_labels in EAD_2 is revealed to active attackers. Another consideration is that the list of supported cipher suites may potentially be used to identify the application. The Initiator and the Responder must also make sure that unauthenticated data does not trigger any harmful actions. In particular, this applies to EAD_1 and error messages.

An attacker observing network traffic may use connection identifiers sent in clear in EDHOC or the subsequent application protocol to correlate packets sent on different paths or at different times. The attacker may use this information for traffic flow analysis or to track an endpoint. Application protocols using connection identifiers from EDHOC SHOULD provide mechanisms to update the connection identifiers and MAY provide mechanisms to issue several simultaneously active connection identifiers. See [RFC9000] for a non-constrained example of such mechanisms. Connection identifiers can e.g., be chosen randomly among the set of unused 1-byte connection identifiers. Connection identity privacy mechanisms are only useful when there are not fixed identifiers such as IP address or MAC address in the lower layers.

9.6. Updated Internet Threat Model Considerations

Since the publication of [RFC3552] there has been an increased awareness of the need to protect against endpoints that are compromised, malicious, or whose interests simply do not align with the interests of users [I-D.arkko-arch-internet-threat-model-guidance]. [RFC7624] describes an updated threat model for Internet confidentiality, see Section 9.1. [I-D.arkko-arch-internet-threat-model-guidance] further expands the threat model. Implementations and users should take these threat models into account and consider actions to reduce the risk of tracking by other endpoints. In particular, even data sent protected to the other endpoint such as ID_CRED fields and EAD fields can be used for tracking, see Section 2.7 of [I-D.arkko-arch-internet-threat-model-guidance].

The fields ID_CRED_I, ID_CRED_R, EAD_2, EAD_3, and EAD_4 have variable length, and information regarding the length may leak to an attacker. A passive attacker may, e.g., be able to differentiate endpoints using identifiers of different length. To mitigate this information leakage an implementation may ensure that the fields have fixed length or use padding. An implementation may, e.g., only use fixed length identifiers like 'kid' of length 1. Alternatively, padding may be used (see Section 3.8.1) to hide the true length of, e.g., certificates by value in 'x5chain' or 'c5c'.

9.7. Denial-of-Service

EDHOC itself does not provide countermeasures against denial-of-service attacks. In particular, by sending a number of new or replayed message_1 an attacker may cause the Responder to allocate state, perform cryptographic operations, and amplify messages. To mitigate such attacks, an implementation SHOULD make use of available lower layer mechanisms. For instance, when EDHOC is transferred as

an exchange of CoAP messages, the CoAP server can use the Echo option defined in [RFC9175] which forces the CoAP client to demonstrate reachability at its apparent network address. To avoid an additional roundtrip the Initiator can reduce the amplification factor by padding message_1, i.e., using EAD_1, see Section 3.8.1. Note that while the Echo option mitigates some resource exhaustion aspects of spoofing, it does not protect against a distributed denial-of-service attack made by real, potentially compromised, clients. Similarly, limiting amplification only reduces the impact, which still may be significant because of a large number of clients engaged in the attack.

An attacker can also send faked message_2, message_3, message_4, or error in an attempt to trick the receiving party to send an error message and abort the EDHOC session. EDHOC implementations MAY evaluate if a received message is likely to have been forged by an attacker and ignore it without sending an error message or aborting the EDHOC session.

9.8. Implementation Considerations

The availability of a secure random number generator is essential for the security of EDHOC. If no true random number generator is available, a random seed MUST be provided from an external source and used with a cryptographically secure pseudorandom number generator. As each pseudorandom number must only be used once, an implementation needs to get a unique input to the pseudorandom number generator after reboot, or continuously store state in nonvolatile memory. Appendix B.1.1 in [RFC8613] describes issues and solution approaches for writing to nonvolatile memory. Intentionally or unintentionally weak or predictable pseudorandom number generators can be abused or exploited for malicious purposes. [RFC8937] describes a way for security protocol implementations to augment their (pseudo)random number generators using a long-term private key and a deterministic signature function. This improves randomness from broken or otherwise subverted random number generators. The same idea can be used with other secrets and functions such as a Diffie-Hellman function or a symmetric secret and a PRF like HMAC or KMAC. It is RECOMMENDED to not trust a single source of randomness and to not put unaugmented random numbers on the wire.

For many constrained IoT devices it is problematic to support several crypto primitives. Existing devices can be expected to support either ECDSA or EdDSA. If ECDSA is supported, "deterministic ECDSA" as specified in [RFC6979] MAY be used. Pure deterministic elliptic-curve signatures such as deterministic ECDSA and EdDSA have gained popularity over randomized ECDSA as their security do not depend on a source of high-quality randomness. Recent research has however found

that implementations of these signature algorithms may be vulnerable to certain side-channel and fault injection attacks due to their determinism. See e.g., Section 1 of [I-D.irtf-cfrg-det-sigs-with-noise] for a list of attack papers. As suggested in Section 2.1.1 of [RFC9053] this can be addressed by combining randomness and determinism.

Appendix D of [I-D.ietf-lwig-curve-representations] describes how Montgomery curves such as X25519 and X448 and (twisted) Edwards curves as curves such as Ed25519 and Ed448 can be mapped to and from short-Weierstrass form for implementation on platforms that accelerate elliptic curve group operations in short-Weierstrass form.

All private keys, symmetric keys, and IVs MUST be secret. Only the Responder SHALL have access to the Responder's private authentication key and only the Initiator SHALL have access to the Initiator's private authentication key. Implementations should provide countermeasures to side-channel attacks such as timing attacks. Intermediate computed values such as ephemeral ECDH keys and ECDH shared secrets MUST be deleted after key derivation is completed.

The Initiator and the Responder are responsible for verifying the integrity and validity of certificates. Verification of validity may require the use of a Real-Time Clock (RTC). The selection of trusted CAs should be done very carefully and certificate revocation should be supported. The choice of revocation mechanism is left to the application. For example, in case of X.509 certificates, Certificate Revocation Lists [RFC5280] or OCSP [RFC6960] may be used.

Similar considerations as for certificates are needed for CWT/CCS. The endpoints are responsible for verifying the integrity and validity of CWT/CCS, and to handle revocation. The application needs to determine what trust anchors are relevant, and have a well-defined trust-establishment process. A self-signed certificate/CWT or CCS appearing in the protocol cannot be a trigger to modify the set of trust anchors. One common way for a new trust anchor to be added to (or removed from) a device is by means firmware upgrade. See [RFC9360] for a longer discussion on trust and validation in constrained devices.

Just like for certificates, the contents of the COSE header parameters 'kcwt' and 'kccs' defined in Section 10.6 must be processed as untrusted input. Endpoints that intend to rely on the assertions made by a CWT/CCS obtained from any of these methods need to validate the contents. For 'kccs', which enables transport of raw public keys, the data structure used does not include any protection or verification data. 'kccs' may be used for unauthenticated operations, e.g. trust on first use, with the limitations and caveats entailed, see Appendix D.5.

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

If two nodes unintentionally initiate two simultaneous EDHOC sessions with each other even if they only want to complete a single EDHOC session, they MAY abort the EDHOC session with the lexicographically smallest G_X. Note that in cases where several EDHOC sessions with different parameter sets (method, COSE headers, etc.) are used, an attacker can affect which parameter set will be used by blocking some of the parameter sets.

If supported by the device, it is RECOMMENDED that at least the long-term private keys are stored in a Trusted Execution Environment (TEE, see for example [RFC9397]) and that sensitive operations using these keys are performed inside the TEE. To achieve even higher security it is RECOMMENDED that additional operations such as ephemeral key generation, all computations of shared secrets, and storage of the PRK keys can be done inside the TEE. The use of a TEE aims at preventing code within that environment to be tampered with, and preventing data used by such code to be read or tampered with by code outside that environment.

Note that HKDF-Expand has a relatively small maximum output length of 255 hash_length, where hash_length is the output size in bytes of the EDHOC hash algorithm of the selected cipher suite. This means that when SHA-256 is used as hash algorithm, PLAINTEXT_2 cannot be longer than 8160 bytes. This is probably not a limitation for most intended applications, but to be able to support for example long certificate chains or large external authorization data, there is a backwards compatible method specified in Appendix G.

The sequence of transcript hashes in EDHOC (TH_2, TH_3, TH_4) does not make use of a so-called running hash. This is a design choice as running hashes are often not supported on constrained platforms.

When parsing a received EDHOC message, implementations MUST abort the EDHOC session if the message does not comply with the CDDL for that message. Implementations are not required to support non-deterministic encodings and MAY abort the EDHOC session if the received EDHOC message is not encoded using deterministic CBOR. Implementations MUST abort the EDHOC session if validation of a received public key fails or if any cryptographic field has the wrong length.

10. IANA Considerations

This Section gives IANA Considerations and, unless otherwise noted, conforms with [RFC8126].

10.1. EDHOC Exporter Label Registry

IANA is requested to create a new registry under the new registry group "Ephemeral Diffie-Hellman Over COSE (EDHOC)" as follows:

Registry Name: EDHOC Exporter Label

Reference: [[this document]]

Label	Description	Reference
0	Derived OSCORE Master Secret	[[this document]]
1	Derived OSCORE Master Salt	[[this document]]
2-22	Unassigned	
23	Reserved	[[this document]]
24-32767	Unassigned	
32768-65535	Private Use	

Figure 13: EDHOC exporter label.

Range	Registration Procedures
0 to 23	Standards Action
24 to 32767	Expert Review

10.2. EDHOC Cipher Suites Registry

IANA is requested to create a new registry under the new registry group "Ephemeral Diffie-Hellman Over COSE (EDHOC)" as follows:

Registry Name: EDHOC Cipher Suites

Reference: [[this document]]

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

Value: -24
Array: N/A
Description: Private Use
Reference: [[this document]]

Value: -23
Array: N/A
Description: Private Use
Reference: [[this document]]

Value: -22
Array: N/A
Description: Private Use
Reference: [[this document]]

Value: -21
Array: N/A
Description: Private Use
Reference: [[this document]]

Value: 0
Array: 10, -16, 8, 4, -8, 10, -16
Description: AES-CCM-16-64-128, SHA-256, 8, X25519, EdDSA,
AES-CCM-16-64-128, SHA-256
Reference: [[this document]]

Value: 1
Array: 30, -16, 16, 4, -8, 10, -16
Description: AES-CCM-16-128-128, SHA-256, 16, X25519, EdDSA,
AES-CCM-16-64-128, SHA-256
Reference: [[this document]]

Value: 2
Array: 10, -16, 8, 1, -7, 10, -16
Description: AES-CCM-16-64-128, SHA-256, 8, P-256, ES256,
AES-CCM-16-64-128, SHA-256
Reference: [[this document]]

Value: 3
Array: 30, -16, 16, 1, -7, 10, -16
Description: AES-CCM-16-128-128, SHA-256, 16, P-256, ES256,
AES-CCM-16-64-128, SHA-256
Reference: [[this document]]

Value: 4
Array: 24, -16, 16, 4, -8, 24, -16
Description: ChaCha20/Poly1305, SHA-256, 16, X25519, EdDSA,
ChaCha20/Poly1305, SHA-256
Reference: [[this document]]

Value: 5
Array: 24, -16, 16, 1, -7, 24, -16
Description: ChaCha20/Poly1305, SHA-256, 16, P-256, ES256,
ChaCha20/Poly1305, SHA-256
Reference: [[this document]]

Value: 6
Array: 1, -16, 16, 4, -7, 1, -16
Description: A128GCM, SHA-256, 16, X25519, ES256,
A128GCM, SHA-256
Reference: [[this document]]

Value: 23
Reserved
Reference: [[this document]]

Value: 24
Array: 3, -43, 16, 2, -35, 3, -43
Description: A256GCM, SHA-384, 16, P-384, ES384,
A256GCM, SHA-384
Reference: [[this document]]

Value: 25
Array: 24, -45, 16, 5, -8, 24, -45
Description: ChaCha20/Poly1305, SHAKE256, 16, X448, EdDSA,
ChaCha20/Poly1305, SHAKE256
Reference: [[this document]]

Range	Registration Procedures
-65536 to -25	Specification Required
-20 to 23	Standards Action with Expert Review
24 to 65535	Specification Required

10.3. EDHOC Method Type Registry

IANA is requested to create a new registry under the new registry group "Ephemeral Diffie-Hellman Over COSE (EDHOC)" as follows:

Registry Name: EDHOC Method Type

Reference: [[this document]]

The columns of the registry are Value, Initiator Authentication Key, and Responder Authentication Key, and Reference, where Value is an integer and the key columns are text strings describing the authentication keys.

The initial contents of the registry are shown in Figure 4. Method 23 is Reserved.

Range	Registration Procedures
-65536 to -25	Specification Required
-24 to 23	Standards Action with Expert Review
24 to 65535	Specification Required

10.4. EDHOC Error Codes Registry

IANA is requested to create a new registry under the new registry group "Ephemeral Diffie-Hellman Over COSE (EDHOC)" as follows:

Registry Name: EDHOC Error Codes

Reference: [[this document]]

The columns of the registry are ERR_CODE, ERR_INFO Type, Description, and Reference, where ERR_CODE is an integer, ERR_INFO is a CDDL defined type, and Description is a text string. The initial contents of the registry are shown in Figure 10. Error code 23 is Reserved.

Range	Registration Procedures
-65536 to -25	Expert Review
-24 to 23	Standards Action
24 to 65535	Expert Review

10.5. EDHOC External Authorization Data Registry

IANA is requested to create a new registry under the new registry group "Ephemeral Diffie-Hellman Over COSE (EDHOC)" as follows:

Registry Name: EDHOC External Authorization Data

Reference: [[this document]]

The columns of the registry are Name, Label, Description, and Reference, where Label is a non-negative integer and the other columns are text strings. The initial contents of the registry is shown in Figure 14. EAD label 23 is Reserved.

Name	Label	Description	Reference
Padding	0	Randomly generated CBOR byte string	[[this document]] Section 3.8.1

Figure 14: EAD labels.

Range	Registration Procedures
0 to 23	Standards Action with Expert Review
24 to 65535	Specification Required

10.6. COSE Header Parameters Registry

IANA is requested to register the following entries in the "COSE Header Parameters" registry under the registry group "CBOR Object Signing and Encryption (COSE)" (see Figure 15): The value of the 'kcwt' header parameter is a COSE Web Token (CWT) [RFC8392], and the value of the 'kccs' header parameter is a CWT Claims Set (CCS), see Section 1.4. The CWT/CCS must contain a COSE_Key in a 'cnf' claim [RFC8747]. The Value Registry for this item is empty and omitted from the table below.

Name	Label	Value Type	Description
kcwt	TBD1	COSE_Messages	A CBOR Web Token (CWT) containing a COSE_Key in a 'cnf' claim and possibly other claims. CWT is defined in RFC 8392. COSE_Messages is defined in RFC 9052.
kccs	TBD2	map	A CWT Claims Set (CCS) containing a COSE_Key in a 'cnf' claim and possibly other claims. CCS is defined in RFC 8392.

Figure 15: COSE header parameter labels.

10.7. The Well-Known URI Registry

IANA is requested to add the well-known URI "edhoc" to the "Well-Known URIs" registry.

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

10.8. Media Types Registry

IANA is requested to add the media types "application/edhoc+cbor-seq" and "application/cid-edhoc+cbor-seq" to the "Media Types" registry.

10.8.1. application/edhoc+cbor-seq Media Type Registration

* Type name: application
* Subtype name: edhoc+cbor-seq
* Required parameters: N/A
* Optional parameters: N/A
* Encoding considerations: binary
* Security considerations: See Section 7 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: See "Authors' Addresses" section.
* Intended usage: COMMON
* Restrictions on usage: N/A
* Author: See "Authors' Addresses" section.
* Change Controller: IESG

10.8.2. application/cid-edhoc+cbor-seq Media Type Registration

* Type name: application
* Subtype name: cid-edhoc+cbor-seq
* Required parameters: N/A
* Optional parameters: N/A
* Encoding considerations: binary
* Security considerations: See Section 7 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: See "Authors' Addresses" section.
* Intended usage: COMMON
* Restrictions on usage: N/A

* Author: See "Authors' Addresses" section.

* Change Controller: IESG

10.9. CoAP Content-Formats Registry

IANA is requested to add the media types "application/edhoc+cbor-seq" and "application/cid-edhoc+cbor-seq" to the "CoAP Content-Formats" registry under the registry group "Constrained RESTful Environments (CoRE) Parameters".

Media Type	Encoding	ID	Reference
application/edhoc+cbor-seq	-	TBD5	[[this document]]
application/cid-edhoc+cbor-seq	-	TBD6	[[this document]]

Figure 16: CoAP Content-Format IDs

10.10. Resource Type (rt=) Link Target Attribute Values Registry

IANA is requested to add the resource type "core.edhoc" to the "Resource Type (rt=) Link Target Attribute Values" registry under the registry group "Constrained RESTful Environments (CoRE) Parameters".

* Value: "core.edhoc"

* Description: EDHOC resource.

* Reference: [[this document]]

10.11. Expert Review Instructions

The IANA Registries established in this document are defined as "Expert Review", "Specification Required" or "Standards Action with Expert Review". This section gives some general guidelines for what the experts should be looking for, but they are being designated as experts for a reason so they should be given substantial latitude.

Expert reviewers should take into consideration the following points:

* Clarity and correctness of registrations. Experts are expected to check the clarity of purpose and use of the requested entries. Expert needs to make sure the values of algorithms are taken from the right registry, when that is required. Experts should consider requesting an opinion on the correctness of registered parameters from relevant IETF working groups. Encodings that do not meet these objective of clarity and completeness should not be registered.

* Experts should take into account the expected usage of fields when approving code point assignment. The length of the encoded value should be weighed against how many code points of that length are left, the size of device it will be used on, and the number of code points left that encode to that size.

* Even for "Expert Review" specifications are recommended. When specifications are not provided for a request where Expert Review is the assignment policy, the description provided needs to have sufficient information to verify the code points above.

11. References

11.1. Normative References

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

[RFC3279] Bassham, L., Polk, W., and R. Housley, "Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 3279, DOI 10.17487/RFC3279, April 2002, <https://www.rfc-editor.org/info/rfc3279>.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/info/rfc3552>.

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

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

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

[RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <https://www.rfc-editor.org/info/rfc6960>.

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

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

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

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

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8392] Jones, M., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "CBOR Web Token (CWT)", RFC 8392, DOI 10.17487/RFC8392, May 2018, <https://www.rfc-editor.org/info/rfc8392>.

[RFC8410] Josefsson, S. and J. Schaad, "Algorithm Identifiers for Ed25519, Ed448, X25519, and X448 for Use in the Internet X.509 Public Key Infrastructure", RFC 8410, DOI 10.17487/RFC8410, August 2018, <https://www.rfc-editor.org/info/rfc8410>.

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/info/rfc8610>.

[RFC8613] Selander, G., Mattsson, J., Palombini, F., and L. Seitz, "Object Security for Constrained RESTful Environments (OSCORE)", RFC 8613, DOI 10.17487/RFC8613, July 2019, <https://www.rfc-editor.org/info/rfc8613>.

[RFC8724] Minaburo, A., Toutain, L., Gomez, C., Barthel, D., and JC. Zuniga, "SCHC: Generic Framework for Static Context Header Compression and Fragmentation", RFC 8724, DOI 10.17487/RFC8724, April 2020, <https://www.rfc-editor.org/info/rfc8724>.

[RFC8742] Bormann, C., "Concise Binary Object Representation (CBOR) Sequences", RFC 8742, DOI 10.17487/RFC8742, February 2020, <https://www.rfc-editor.org/info/rfc8742>.

[RFC8747] Jones, M., Seitz, L., Selander, G., Erdtman, S., and H. Tschofenig, "Proof-of-Possession Key Semantics for CBOR Web Tokens (CWTs)", RFC 8747, DOI 10.17487/RFC8747, March 2020, <https://www.rfc-editor.org/info/rfc8747>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/info/rfc8949>.

[RFC9052] Schaad, J., "CBOR Object Signing and Encryption (COSE): Structures and Process", STD 96, RFC 9052, DOI 10.17487/RFC9052, August 2022, <https://www.rfc-editor.org/info/rfc9052>.

[RFC9053] Schaad, J., "CBOR Object Signing and Encryption (COSE): Initial Algorithms", RFC 9053, DOI 10.17487/RFC9053, August 2022, <https://www.rfc-editor.org/info/rfc9053>.

[RFC9175] Amsüss, C., Preuß Mattsson, J., and G. Selander, "Constrained Application Protocol (CoAP): Echo, Request-Tag, and Token Processing", RFC 9175, DOI 10.17487/RFC9175, February 2022, <https://www.rfc-editor.org/info/rfc9175>.

[RFC9360] Schaad, J., "CBOR Object Signing and Encryption (COSE): Header Parameters for Carrying and Referencing X.509 Certificates", RFC 9360, DOI 10.17487/RFC9360, February 2023, <https://www.rfc-editor.org/info/rfc9360>.

11.2. Informative References

[Bruni18] Bruni, A., Sahl Jørgensen, T., Grønbech Petersen, T., and C. Schürmann, "Formal Verification of Ephemeral Diffie-Hellman Over COSE (EDHOC)", November 2018, <https://www.springerprofessional.de/en/formal-verification-of-ephemeral-diffie-hellman-over-cose-edhoc/16284348>.

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

[CNSA] NSA, "Commercial National Security Algorithm Suite", August 2015, <https://en.wikipedia.org/wiki/Commercial_National_Security_Algorithm_Suite>.

[CottierPointcheval22] Cottier, B. and D. Pointcheval, "Security Analysis of the EDHOC protocol", September 2022, <https://arxiv.org/abs/2209.03599>.

[Degabriele11] Degabriele, J. P., Lehmann, A., Paterson, K. G., Smart, N. P., and M. Strefler, "On the Joint Security of Encryption and Signature in EMV", December 2011, <https://eprint.iacr.org/2011/615>.

[GuentherIlunga22] Günther, F. and M. Ilunga, "Careful with MAc-then-SIGn: A Computational Analysis of the EDHOC Lightweight Authenticated Key Exchange Protocol", December 2022, <https://eprint.iacr.org/2022/1705>.

[HKDFpaper] Krawczyk, H., "Cryptographic Extraction and Key Derivation: The HKDF Scheme", May 2010, <https://eprint.iacr.org/2010/264.pdf>.

[I-D.arkko-arch-internet-threat-model-guidance]
Arkko, J. and S. Farrell, "Internet Threat Model Guidance", Work in Progress, Internet-Draft, draft-arkko-arch-internet-threat-model-guidance-00, 12 July 2021, <https://datatracker.ietf.org/doc/html/draft-arkko-arch-internet-threat-model-guidance-00>.

[I-D.ietf-core-oscore-edhoc]
Palombini, F., Tiloca, M., Höglund, R., Hristozov, S., and G. Selander, "Using Ephemeral Diffie-Hellman Over COSE (EDHOC) with the Constrained Application Protocol (CoAP) and Object Security for Constrained RESTful Environments (OSCORE)", Work in Progress, Internet-Draft, draft-ietf-core-oscore-edhoc-10, 29 November 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-core-oscore-edhoc-10>.

[I-D.ietf-core-oscore-key-update]
Höglund, R. and M. Tiloca, "Key Update for OSCORE (KUDOS)", Work in Progress, Internet-Draft, draft-ietf-core-oscore-key-update-06, 23 October 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-core-oscore-key-update-06>.

[I-D.ietf-cose-cbor-encoded-cert]
Mattsson, J. P., Selander, G., Raza, S., Höglund, J., and M. Furuhed, "CBOR Encoded X.509 Certificates (C509 Certificates)", Work in Progress, Internet-Draft, draft-ietf-cose-cbor-encoded-cert-07, 20 October 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-cose-cbor-encoded-cert-07>.

[I-D.ietf-iotops-security-protocol-comparison]
Mattsson, J. P., Palombini, F., and M. Vuini, "Comparison of CoAP Security Protocols", Work in Progress, Internet-Draft, draft-ietf-iotops-security-protocol-comparison-03, 23 October 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-iotops-security-protocol-comparison-03>.

[I-D.ietf-lake-reqs]
Vuini, M., Selander, G., Mattsson, J. P., and D. Garcia-Carillo, "Requirements for a Lightweight AKE for OSCORE", Work in Progress, Internet-Draft, draft-ietf-lake-reqs-04, 8 June 2020, <https://datatracker.ietf.org/doc/html/draft-ietf-lake-reqs-04>.

[I-D.ietf-lake-traces]
Selander, G., Mattsson, J. P., Serafin, M., Tiloca, M., and M. Vuini, "Traces of EDHOC", Work in Progress, Internet-Draft, draft-ietf-lake-traces-08, 22 September 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-lake-traces-08>.

[I-D.ietf-lwig-curve-representations]
Struik, R., "Alternative Elliptic Curve Representations", Work in Progress, Internet-Draft, draft-ietf-lwig-curve-representations-23, 21 January 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-lwig-curve-representations-23>.

[I-D.ietf-rats-eat]
Lundblade, L., Mandyam, G., O'Donoghue, J., and C. Wallace, "The Entity Attestation Token (EAT)", Work in Progress, Internet-Draft, draft-ietf-rats-eat-25, 15 January 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-rats-eat-25>.

[I-D.irtf-cfrg-det-sigs-with-noise]
Mattsson, J. P., Thormarker, E., and S. Ruohomaa, "Deterministic ECDSA and EdDSA Signatures with Additional Randomness", Work in Progress, Internet-Draft, draft-irtf-cfrg-det-sigs-with-noise-00, 8 August 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-det-sigs-with-noise-00>.

[I-D.selander-lake-authz]
Selander, G., Mattsson, J. P., Vuini, M., Richardson, M., and A. Schellenbaum, "Lightweight Authorization using Ephemeral Diffie-Hellman Over COSE", Work in Progress, Internet-Draft, draft-selander-lake-authz-03, 7 July 2023, <https://datatracker.ietf.org/doc/html/draft-selander-lake-authz-03>.

[Jacomme23]
Jacomme, C., Klein, E., Kremer, S., and M. Racouchot, "A comprehensive, formal and automated analysis of the EDHOC protocol", October 2022, <https://hal.inria.fr/hal-03810102/>.

[NISTPQC] "Post-Quantum Cryptography FAQs", August 2023, <https://csrc.nist.gov/Projects/post-quantum-cryptography/faqs>.

[Noise] Perrin, T., "The Noise Protocol Framework, Revision 34", July 2018, <https://noiseprotocol.org/noise.html>.

[Norrman20] Norrman, K., Sundararajan, V., and A. Bruni, "Formal Analysis of EDHOC Key Establishment for Constrained IoT Devices", September 2020, <https://arxiv.org/abs/2007.11427>.

[RFC2986] Nystrom, M. and B. Kaliski, "PKCS #10: Certification Request Syntax Specification Version 1.7", RFC 2986, DOI 10.17487/RFC2986, November 2000, <https://www.rfc-editor.org/info/rfc2986>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC6194] Polk, T., Chen, L., Turner, S., and P. Hoffman, "Security Considerations for the SHA-0 and SHA-1 Message-Digest Algorithms", RFC 6194, DOI 10.17487/RFC6194, March 2011, <https://www.rfc-editor.org/info/rfc6194>.

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

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

[RFC7296] Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., and T. Kivinen, "Internet Key Exchange Protocol Version 2 (IKEv2)", STD 79, RFC 7296, DOI 10.17487/RFC7296, October 2014, <https://www.rfc-editor.org/info/rfc7296>.

[RFC7624] Barnes, R., Schneier, B., Jennings, C., Hardie, T., Trammell, B., Huitema, C., and D. Borkmann, "Confidentiality in the Face of Pervasive Surveillance: A Threat Model and Problem Statement", RFC 7624, DOI 10.17487/RFC7624, August 2015, <https://www.rfc-editor.org/info/rfc7624>.

[RFC8366] Watsen, K., Richardson, M., Pritikin, M., and T. Eckert, "A Voucher Artifact for Bootstrapping Protocols", RFC 8366, DOI 10.17487/RFC8366, May 2018, <https://www.rfc-editor.org/info/rfc8366>.

[RFC8376] Farrell, S., Ed., "Low-Power Wide Area Network (LPWAN) Overview", RFC 8376, DOI 10.17487/RFC8376, May 2018, <https://www.rfc-editor.org/info/rfc8376>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC8937] Cremers, C., Garratt, L., Smyshlyaev, S., Sullivan, N., and C. Wood, "Randomness Improvements for Security Protocols", RFC 8937, DOI 10.17487/RFC8937, October 2020, <https://www.rfc-editor.org/info/rfc8937>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

[RFC9147] Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", RFC 9147, DOI 10.17487/RFC9147, April 2022, <https://www.rfc-editor.org/info/rfc9147>.

[RFC9176] Amsüss, C., Ed., Shelby, Z., Koster, M., Bormann, C., and P. van der Stok, "Constrained RESTful Environments (CoRE) Resource Directory", RFC 9176, DOI 10.17487/RFC9176, April 2022, <https://www.rfc-editor.org/info/rfc9176>.

[RFC9397] Pei, M., Tschofenig, H., Thaler, D., and D. Wheeler, "Trusted Execution Environment Provisioning (TEEP) Architecture", RFC 9397, DOI 10.17487/RFC9397, July 2023, <https://www.rfc-editor.org/info/rfc9397>.

[SECG] "Standards for Efficient Cryptography 1 (SEC 1)", May 2009, <https://www.secg.org/sec1-v2.pdf>.

[SIGMA] Krawczyk, H., "SIGMA - The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE-Protocols", June 2003, <https://www.iacr.org/cryptodb/archive/2003/CRYPTO/1495/1495.pdf>.

[SP-800-108]
Chen, L., "Recommendation for Key Derivation Using Pseudorandom Functions", NIST Special Publication 800-108 Revision 1, August 2022, <https://doi.org/10.6028/NIST.SP.800-108r1>.

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

[SP800-185]
John Kelsey, Shu-jen Chang, and Ray Perlner, "SHA-3 Derived Functions cSHAKE, KMAC, TupleHash and ParallelHash", NIST Special Publication 800-185, December 2016, <https://doi.org/10.6028/NIST.SP.800-185>.

[Thormarker21]
Thormarker, E., "On using the same key pair for Ed25519 and an X25519 based KEM", April 2021, <https://eprint.iacr.org/2021/509.pdf>.

Appendix A. Use with OSCORE and Transfer over CoAP

This appendix describes how to derive an OSCORE security context when EDHOC is used to key OSCORE, and how to transfer EDHOC messages over CoAP. The use of CoAP or OSCORE with EDHOC is optional, but if you are using CoAP or OSCORE, then certain normative requirements apply as detailed in the subsections.

A.1. Deriving the OSCORE Security Context

This section specifies how to use EDHOC output to derive the OSCORE security context.

After successful processing of EDHOC message_3, Client and Server derive Security Context parameters for OSCORE as follows (see Section 3.2 of [RFC8613]):

* The Master Secret and Master Salt SHALL be derived by using the EDHOC_Exporter interface, see Section 4.2.1:
  - The EDHOC Exporter Labels for deriving the OSCORE Master Secret and the OSCORE Master Salt, are the uints 0 and 1, respectively.

- The context parameter is h'' (0x40), the empty CBOR byte string.

- By default, oscore_key_length is the key length (in bytes) of the application AEAD Algorithm of the selected cipher suite for the EDHOC session. Also by default, oscore_salt_length has value 8. The Initiator and Responder MAY agree out-of-band on a longer oscore_key_length than the default, and on shorter or longer than the default oscore_salt_length.

```
Master Secret = EDHOC_Exporter( 0, h'', oscore_key_length )
Master Salt   = EDHOC_Exporter( 1, h'', oscore_salt_length )
```

* The AEAD Algorithm SHALL be the application AEAD algorithm of the selected cipher suite for the EDHOC session.

* The HKDF Algorithm SHALL be the one based on the application hash algorithm of the selected cipher suite for the EDHOC session. For example, if SHA-256 is the application hash algorithm of the selected cipher suite, HKDF SHA-256 is used as HKDF Algorithm in the OSCORE Security Context.

* The relationship between identifiers in OSCORE and EDHOC is specified in Section 3.3.3. The OSCORE Sender ID and Recipient ID SHALL be determined by the EDHOC connection identifiers C_R and C_I for the EDHOC session as shown in Figure 17.

	OSCORE Sender ID	OSCORE Recipient ID
EDHOC Initiator	C_R	C_I
EDHOC Responder	C_I	C_R

Figure 17: Usage of connection identifiers in OSCORE.

Client and Server SHALL use the parameters above to establish an OSCORE Security Context, as per Section 3.2.1 of [RFC8613].

From then on, Client and Server retrieve the OSCORE protocol state using the Recipient ID, and optionally other transport information such as the 5-tuple.

A.2. Transferring EDHOC over CoAP

This section specifies how EDHOC can be transferred as an exchange of CoAP [RFC7252] messages. CoAP provides a reliable transport that can preserve packet ordering, provides flow and congestion control, and handles message duplication. CoAP can also perform fragmentation and mitigate certain denial-of-service attacks. The underlying CoAP transport should be used in reliable mode, in particular when fragmentation is used, to avoid, e.g., situations with hanging endpoints waiting for each other.

EDHOC may run with the Initiator either being CoAP client or CoAP server. We denote the former by the "forward message flow" (see Appendix A.2.1) and the latter by the "reverse message flow" (see Appendix A.2.2). By default, we assume the forward message flow, but the roles SHOULD be chosen to protect the most sensitive identity, see Section 9.

According to this specification, EDHOC is transferred in POST requests to the Uri-Path: "/.well-known/edhoc" (see Section 10.7), and 2.04 (Changed) responses. An application may define its own path that can be discovered, e.g., using a resource directory [RFC9176]. Client applications can use the resource type "core.edhoc" to discover a server's EDHOC resource, i.e., where to send a request for executing the EDHOC protocol, see Section 10.10. An alternative transfer of the forward message flow is specified in [I-D.ietf-core-oscore-edhoc].

In order for the server to correlate a message received from a client to a message previously sent in the same EDHOC session over CoAP, messages sent by the client SHALL be prepended with the CBOR serialization of the connection identifier which the server has selected, see Section 3.4.1. This applies both to the forward and the reverse message flows. To indicate a new EDHOC session in the forward message flow, message_1 SHALL be prepended with the CBOR simple value true (0xf5). Even if CoAP is carried over a reliable transport protocol such as TCP, the prepending of identifiers specified here SHALL be practiced to enable interoperability independent of how CoAP is transported.

The prepended identifiers are encoded in CBOR and thus self-delimiting. The representation of identifiers described in Section 3.3.2 SHALL be used. They are sent in front of the actual EDHOC message to keep track of messages in an EDHOC session, and only the part of the body following the identifier is used for EDHOC processing. In particular, the connection identifiers within the EDHOC messages are not impacted by the prepended identifiers.

An EDHOC message has media type application/edhoc+cbor-seq, whereas an EDHOC message prepended by a connection identifier has media type application/cid-edhoc+cbor-seq, see Section 10.9.

To mitigate certain denial-of-service attacks, the CoAP server MAY respond to the first POST request with a 4.01 (Unauthorized) containing an Echo option [RFC9175]. This forces the Initiator to demonstrate reachability at its apparent network address. If message fragmentation is needed, the EDHOC messages may be fragmented using the CoAP Block-Wise Transfer mechanism [RFC7959].

EDHOC error messages need to be transported in response to a message that failed (see Section 6). EDHOC error messages transported with CoAP are carried in the payload.

Note that the transport over CoAP can serve as a blueprint for other client-server protocols:

* The client prepends the connection identifier selected by the server (or, for message_1, the CBOR simple value true) to any request message it sends.

* The server does not send any such indicator, as responses are matched to request by the client-server protocol design.

A.2.1. The Forward Message Flow

In the forward message flow the CoAP client is the Initiator and the CoAP server is the Responder. This flow protects the client identity against active attackers and the server identity against passive attackers.

In the forward message flow, the CoAP Token enables correlation on the Initiator (client) side, and the prepended C_R enables correlation on the Responder (server) side.

* EDHOC message_1 is sent in the payload of a POST request from the client to the server's resource for EDHOC, prepended with the identifier true (0xf5) indicating a new EDHOC session.

* EDHOC message_2 or the EDHOC error message is sent from the server to the client in the payload of the response, in the former case with response code 2.04 (Changed), in the latter with response code as specified in Appendix A.2.3.

* EDHOC message_3 or the EDHOC error message is sent from the client to the server's resource in the payload of a POST request, prepended with the connection identifier C_R.

* If EDHOC message_4 is used, or in case of an error message, it is sent from the server to the client in the payload of the response, with response codes analogously to message_2. In case of an error message sent in response to message_4, it is sent analogously to error message sent in response to message_2.

An example of a completed EDHOC session over CoAP in the forward message flow is shown in Figure 18.

Figure 18: Example of the forward message flow.

The forward message flow of EDHOC can be combined with an OSCORE exchange in a total of two round-trips, see [I-D.ietf-core-oscore-edhoc].

A.2.2. The Reverse Message Flow

In the reverse message flow the CoAP client is the Responder and the CoAP server is the Initiator. This flow protects the server identity against active attackers and the client identity against passive attackers.

In the reverse message flow, the CoAP Token enables correlation on the Responder (client) side, and the prepended C_I enables correlation on the Initiator (server) side.

* To trigger a new EDHOC session, the client makes an empty POST request to the server's resource for EDHOC.

* EDHOC message_1 is sent from the server to the client in the payload of the response with response code 2.04 (Changed).

* EDHOC message_2 or the EDHOC error message is sent from the client to the server's resource in the payload of a POST request, prepended with the connection identifier C_I.

* EDHOC message_3 or the EDHOC error message is sent from the server to the client in the payload of the response, in the former case with response code 2.04 (Changed), in the latter with response code as specified in Appendix A.2.3.

* If EDHOC message_4 is used, or in case of an error message, it is sent from the client to the server's resource in the payload of a POST request, prepended with the connection identifier C_I. In case of an error message sent in response to message_4, it is sent analogously to an error message sent in response to message_2.

An example of a completed EDHOC session over CoAP in the reverse message flow is shown in Figure 19.

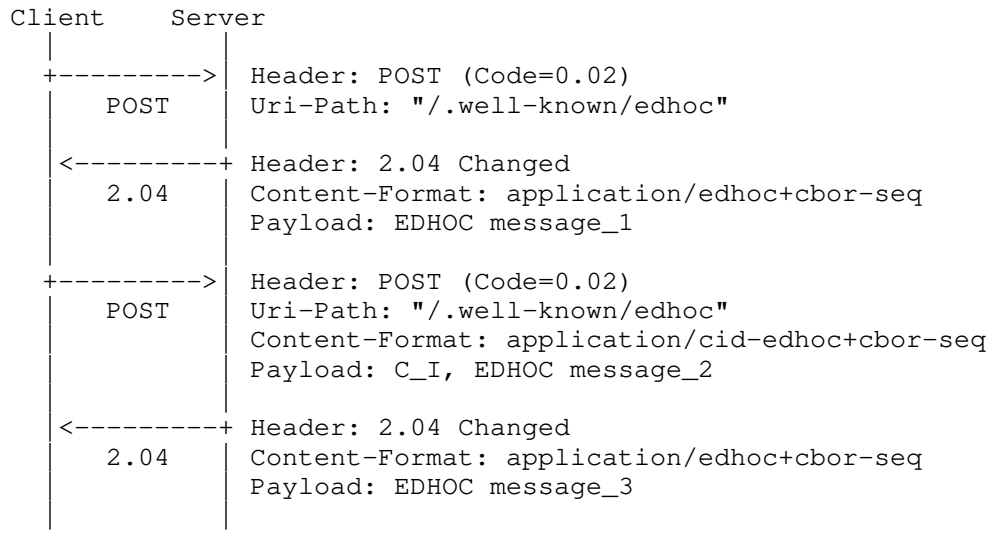


Figure 19: Example of the reverse message flow.

A.2.3. Errors in EDHOC over CoAP

When using EDHOC over CoAP, EDHOC error messages sent as CoAP responses MUST be sent in the payload of error responses, i.e., they MUST specify a CoAP error response code. In particular, it is RECOMMENDED that such error responses have response code either 4.00 (Bad Request) in case of client error (e.g., due to a malformed EDHOC message), or 5.00 (Internal Server Error) in case of server error (e.g., due to failure in deriving EDHOC keying material). The Content-Format of the error response MUST be set to application/edhoc+cbor-seq, see Section 10.9.

Appendix B. Compact Representation

This section defines a format for compact representation based on the Elliptic-Curve-Point-to-Octet-String Conversion defined in Section 2.3.3 of [SECG].

As described in Section 4.2 of [RFC6090] the x-coordinate of an elliptic curve public key is a suitable representative for the entire point whenever scalar multiplication is used as a one-way function. One example is ECDH with compact output, where only the x-coordinate of the computed value is used as the shared secret.

In EDHOC, compact representation is used for the ephemeral public keys (G_X and G_Y), see Section 3.7. Using the notation from [SECG], the output is an octet string of length ceil((log2 q) / 8), where ceil(x) is the smallest integer not less than x. See [SECG] for a definition of q, M, X, xp, and ~yp. The steps in Section 2.3.3 of [SECG] are replaced by:

1. Convert the field element xp to an octet string X of length ceil((log2 q) / 8) octets using the conversion routine specified in Section 2.3.5 of [SECG].

2. Output M = X

The encoding of the point at infinity is not supported.

Compact representation does not change any requirements on validation, see Section 9.2. Using compact representation has some security benefits. An implementation does not need to check that the point is not the point at infinity (the identity element). Similarly, as not even the sign of the y-coordinate is encoded, compact representation trivially avoids so-called "benign malleability" attacks where an attacker changes the sign, see [SECG].

The following may be needed for validation or compatibility with APIs that do not support compact representation or do not support the full [SECG] format:

* If a compressed y-coordinate is required, then the value ~yp set to zero can be used. The compact representation described above can in such a case be transformed into the SECG point compressed format by prepending it with the single byte 0x02 (i.e., M = 0x02 || X).

* If an uncompressed y-coordinate is required, then a y-coordinate has to be calculated following Section 2.3.4 of [SECG] or Appendix C of [RFC6090]. Any of the square roots (see [SECG] or [RFC6090]) can be used. The uncompressed SECG format is M = 0x04 || X || Y.

For example: The curve P-256 has the parameters (using the notation in [RFC6090])

* p = 2^256 2^224 + 2^192 + 2^96 1

* a = -3

* b = 41058363725152142129326129780047268409114441015993725554835256314039467401291

Given an example x:

* x = 11579208918339630209554680715474055844340679510865333639897069777227887997665 25

we can calculate y as the square root w = (x^3 + a x + b)^((p + 1)/4) (mod p)

* y = 83438718007019280682007586491862600528145125996401575416632522940595860276856

Note that this does not guarantee that (x, y) is on the correct elliptic curve. A full validation according to Section 5.6.2.3.3 of [SP-800-56A] can be achieved by also checking that 0 x < p and that y^2 x^3 + a x + b (mod p).

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

This Appendix is intended to help implementors not familiar with CBOR [RFC8949], CDDL [RFC8610], COSE [RFC9052], and HKDF [RFC5869].

C.1. CBOR and CDDL

The Concise Binary Object Representation (CBOR) [RFC8949] is a data format designed for small code size and small message size. CBOR builds on the JSON data model but extends it by e.g., encoding binary data directly without base64 conversion. In addition to the binary CBOR encoding, CBOR also has a diagnostic notation that is readable and editable by humans. The Concise Data Definition Language (CDDL) [RFC8610] provides a way to express structures for protocol messages and APIs that use CBOR. [RFC8610] also extends the diagnostic notation.

CBOR data items are encoded to or decoded from byte strings using a type-length-value encoding scheme, where the three highest order bits of the initial byte contain information about the major type. CBOR supports several types of data items, in addition to integers (int, uint), simple values, byte strings (bstr), and text strings (tstr), CBOR also supports arrays [] of data items, maps {} of pairs of data items, and sequences [RFC8742] of data items. Some examples are given below.

The EDHOC specification sometimes use CDDL names in CBOR diagnostic notation as in e.g., << ID_CRED_R, ? EAD_2 >>. This means that EAD_2 is optional and that ID_CRED_R and EAD_2 should be substituted with their values before evaluation. I.e., if ID_CRED_R = { 4 : h'' } and EAD_2 is omitted then << ID_CRED_R, ? EAD_2 >> = << { 4 : h'' } >>, which encodes to 0x43a10440. We also make use of the occurrence symbol "*", like in e.g., 2* int, meaning two or more CBOR integers.

For a complete specification and more examples, see [RFC8949] and [RFC8610]. We recommend implementors get used to CBOR by using the CBOR playground [CborMe].

Diagnostic	Encoded	Type
1	0x01	unsigned integer
24	0x1818	unsigned integer
-24	0x37	negative integer
-25	0x3818	negative integer
true	0xf5	simple value
h''	0x40	byte string
h'12cd'	0x4212cd	byte string
'12cd'	0x4431326364	byte string
"12cd"	0x6431326364	text string
{ 4 : h'cd' }	0xa10441cd	map
<< 1, 2, true >>	0x430102f5	byte string
[1, 2, true]	0x830102f5	array
(1, 2, true)	0x0102f5	sequence
1, 2, true	0x0102f5	sequence

Figure 20: Examples of use of CBOR and CDDL.

C.2. CDDL Definitions

This section compiles the CDDL definitions for ease of reference.

```
suites = [ 2* int ] / int

ead = (
  ead_label : int,
  ? ead_value : bstr,
)

EAD_1 = 1* ead
EAD_2 = 1* ead
EAD_3 = 1* ead
EAD_4 = 1* ead

message_1 = (
  METHOD : int,
  SUITES_I : suites,
  G_X : bstr,
  C_I : bstr / -24..23,
  ? EAD_1,
)

message_2 = (
  G_Y_CIPHERTEXT_2 : bstr,
)
```

```
PLAINTEXT_2 = (
  C_R,
  ID_CRED_R : map / bstr / -24..23,
  Signature_or_MAC_2 : bstr,
  ? EAD_2,
)

message_3 = (
  CIPHERTEXT_3 : bstr,
)

PLAINTEXT_3 = (
  ID_CRED_I : map / bstr / -24..23,
  Signature_or_MAC_3 : bstr,
  ? EAD_3,
)

message_4 = (
  CIPHERTEXT_4 : bstr,
)

PLAINTEXT_4 = (
  ? EAD_4,
)

error = (
  ERR_CODE : int,
  ERR_INFO : any,
)

info = (
  info_label : int,
  context : bstr,
  length : uint,
)
```

C.3. COSE

CBOR Object Signing and Encryption (COSE) [RFC9052] describes how to create and process signatures, message authentication codes, and encryption using CBOR. COSE builds on JOSE, but is adapted to allow more efficient processing in constrained devices. EDHOC makes use of COSE_Key, COSE_Encrypt0, and COSE_Sign1 objects in the message processing:

* ECDH ephemeral public keys of type EC2 or OKP in message_1 and message_2 consist of the COSE_Key parameter named 'x', see Section 7.1 and 7.2 of [RFC9053]

* The ciphertexts in message_3 and message_4 consist of a subset of the single recipient encrypted data object COSE_Encrypt0, which is described in Sections 5.2-5.3 of [RFC9052]. The ciphertext is computed over the plaintext and associated data, using an encryption key and an initialization vector. The associated data is an Enc_structure consisting of protected headers and externally supplied data (external_aad). COSE constructs the input to the AEAD [RFC5116] for message_i (i = 3 or 4, see Section 5.4 and Section 5.5, respectively) as follows:

  - Secret key K = K_i

  - Nonce N = IV_i

  - Plaintext P for message_i

  - Associated Data A = ["Encrypt0", h'', TH_i]

* Signatures in message_2 of method 0 and 2, and in message_3 of method 0 and 1, consist of a subset of the single signer data object COSE_Sign1, which is described in Sections 4.2-4.4 of [RFC9052]. The signature is computed over a Sig_structure containing payload, protected headers and externally supplied data (external_aad) using a private signature key and verified using the corresponding public signature key. For COSE_Sign1, the message to be signed is:

  ["Signature1", protected, external_aad, payload]

  where protected, external_aad and payload are specified in Section 5.3 and Section 5.4.

Different header parameters to identify X.509 or C509 certificates by reference are defined in [RFC9360] and [I-D.ietf-cose-cbor-encoded-cert]:

* by a hash value with the 'x5t' or 'c5t' parameters, respectively:

  - ID_CRED_x = { 34 : COSE_CertHash }, for x = I or R,

  - ID_CRED_x = { TBD3 : COSE_CertHash }, for x = I or R;

* or by a URI with the 'x5u' or 'c5u' parameters, respectively:

  - ID_CRED_x = { 35 : uri }, for x = I or R,

  - ID_CRED_x = { TBD4 : uri }, for x = I or R.

When ID_CRED_x does not contain the actual credential, it may be very short, e.g., if the endpoints have agreed to use a key identifier parameter 'kid':

* ID_CRED_x = { 4 : kid_x }, where kid_x : kid, for x = I or R. For further optimization, see Section 3.5.3.

Note that ID_CRED_x can contain several header parameters, for example { x5u, x5t } or { kid, kid_context }.

ID_CRED_x MAY also identify the credential by value. For example, a certificate chain can be transported in an ID_CRED field with COSE header parameter c5c or x5chain, defined in [I-D.ietf-cose-cbor-encoded-cert] and [RFC9360] and credentials of type CWT and CCS can be transported with the COSE header parameters registered in Section 10.6.

Appendix D. Authentication Related Verifications

EDHOC performs certain authentication related operations, see Section 3.5, but in general it is necessary to make additional verifications beyond EDHOC message processing. Which verifications that are needed depend on the deployment, in particular the trust model and the security policies, but most commonly it can be expressed in terms of verifications of credential content.

EDHOC assumes the existence of mechanisms (certification authority or other trusted third party, pre-provisioning, etc.) for generating and distributing authentication credentials and other credentials, as well as the existence of trust anchors (CA certificates, trusted public keys, etc.). For example, a public key certificate or CWT may rely on a trusted third party whose public key is pre-provisioned, whereas a CCS or a self-signed certificate/CWT may be used when trust in the public key can be achieved by other means, or in the case of Trust on first use, see Appendix D.5.

In this section we provide some examples of such verifications. These verifications are the responsibility of the application but may be implemented as part of an EDHOC library.

D.1. Validating the Authentication Credential

The authentication credential may contain, in addition to the authentication key, other parameters that needs to be verified. For example:

* In X.509 and C509 certificates, signature keys typically have key usage "digitalSignature" and Diffie-Hellman public keys typically have key usage "keyAgreement" [RFC3279][RFC8410].

* In X.509 and C509 certificates validity is expressed using Not After and Not Before. In CWT and CCS, the exp and nbf claims have similar meanings.

D.2. Identities

The application must decide on allowing a connection or not depending on the intended endpoint, and in particular whether it is a specific identity or in a set of identities. To prevent misbinding attacks, the identity of the endpoint is included in a MAC verified through the protocol. More details and examples are provided in this section.

Policies for what connections to allow are typically set based on the identity of the other endpoint, and endpoints typically only allow connections from a specific identity or a small restricted set of identities. For example, in the case of a device connecting to a network, the network may only allow connections from devices which authenticate with certificates having a particular range of serial numbers and signed by a particular CA. Conversely, a device may only be allowed to connect to a network which authenticates with a particular public key.

* When a Public Key Infrastructure (PKI) is used with certificates, the identity is the subject whose unique name, e.g., a domain name, a Network Access Identifier (NAI), or an Extended Unique Identifier (EUI), is included in the endpoint's certificate.

* Similarly, when a PKI is used with CWTs, the identity is the subject identified by the relevant claim(s), such as 'sub' (subject).

* When PKI is not used (e.g., CCS, self-signed certificate/CWT) the identity is typically directly associated with the authentication key of the other party. For example, if identities can be expressed in the form of unique subject names assigned to public keys, then a binding to identity is achieved by including both public key and associated subject name in the authentication credential: CRED_I or CRED_R may be a self-signed certificate/CWT or CCS containing the authentication key and the subject name, see Section 3.5.2. Each endpoint thus needs to know the specific authentication key/unique associated subject name, or set of public authentication keys/unique associated subject names, which it is allowed to communicate with.

To prevent misbinding attacks in systems where an attacker can register public keys without proving knowledge of the private key, SIGMA [SIGMA] enforces a MAC to be calculated over the "identity". EDHOC follows SIGMA by calculating a MAC over the whole authentication credential, which in case of an X.509 or C509 certificate includes the "subject" and "subjectAltName" fields, and in the case of CWT or CCS includes the "sub" claim.

(While the SIGMA paper only focuses on the identity, the same principle is true for other information such as policies associated with the public key.)

D.3. Certification Path and Trust Anchors

When a Public Key Infrastructure (PKI) is used with certificates, the trust anchor is a Certification Authority (CA) certificate. Each party needs at least one CA public key certificate, or just the CA public key. The certification path contains proof that the subject of the certificate owns the public key in the certificate. Only validated public-key certificates are to be accepted.

Similarly, when a PKI is used with CWTs, each party needs to have at least one trusted third party public key as trust anchor to verify the end entity CWTs. The trusted third party public key can, e.g., be stored in a self-signed CWT or in a CCS.

The signature of the authentication credential needs to be verified with the public key of the issuer. X.509 and C509 certificates includes the Issuer field. In CWT and CCS, the iss claim has a similar meaning. The public key is either a trust anchor or the public key in another valid and trusted credential in a certification path from trust anchor to authentication credential.

Similar verifications as made with the authentication credential (see Appendix D.1) are also needed for the other credentials in the certification path.

When PKI is not used (CCS, self-signed certificate/CWT), the trust anchor is the authentication key of the other party, in which case there is no certification path.

D.4. Revocation Status

The application may need to verify that the credentials are not revoked, see Section 9.8. Some use cases may be served by short-lived credentials, for example, where the validity of the credential is on par with the interval between revocation checks. But, in general, credential lifetime and revocation checking are complementary measures to control credential status. Revocation information may be transported as External Authentication Data (EAD), see Appendix E.

D.5. Unauthenticated Operation

EDHOC might be used without authentication by allowing the Initiator or Responder to communicate with any identity except its own. Note that EDHOC without mutual authentication is vulnerable to active on-path attacks and therefore unsafe for general use. However, it is possible to later establish a trust relationship with an unknown or not-yet-trusted endpoint. Some examples:

* The EDHOC authentication credential can be verified out-of-band at a later stage.

* The EDHOC session key can be bound to an identity out-of-band at a later stage.

* Trust on first use (TOFU) can be used to verify that several EDHOC connections are made to the same identity. TOFU combined with proximity is a common IoT deployment model which provides good security if done correctly. Note that secure proximity based on short range wireless technology requires very low signal strength or very low latency.

Appendix E. Use of External Authorization Data

In order to reduce the number of messages and round trips, or to simplify processing, external security applications may be integrated into EDHOC by transporting related external authorization data (EAD) in the messages.

The EAD format is specified in Section 3.8, this section contains examples and further details of how EAD may be used with an appropriate accompanying specification.

* One example is third party assisted authorization, requested with EAD_1, and an authorization artifact (voucher, cf. [RFC8366]) returned in EAD_2, see [I-D.selander-lake-authz].

* Another example is remote attestation, requested in EAD_2, and an Entity Attestation Token (EAT, [I-D.ietf-rats-eat]) returned in EAD_3.

* A third example is certificate enrolment, where a Certificate Signing Request (CSR, [RFC2986]) is included EAD_3, and the issued public key certificate (X.509 [RFC5280], C509 [I-D.ietf-cose-cbor-encoded-cert]) or a reference thereof is returned in EAD_4.

External authorization data should be considered unprotected by EDHOC, and the protection of EAD is the responsibility of the security application (third party authorization, remote attestation, certificate enrolment, etc.). The security properties of the EAD fields (after EDHOC processing) are discussed in Section 9.1.

The content of the EAD field may be used in the EDHOC processing of the message in which they are contained. For example, authentication related information like assertions and revocation information, transported in EAD fields may provide input about trust anchors or validity of credentials relevant to the authentication processing. The EAD fields (like ID_CRED fields) are therefore made available to the application before the message is verified, see details of message processing in Section 5. In the first example above, a voucher in EAD_2 made available to the application can enable the Initiator to verify the identity or public key of the Responder before verifying the signature. An application allowing EAD fields containing authentication information thus may need to handle authentication related verifications associated with EAD processing.

Conversely, the security application may need to wait for EDHOC message verification to complete. In the third example above, the validation of a CSR carried in EAD_3 is not started by the Responder before EDHOC has successfully verified message_3 and proven the possession of the private key of the Initiator.

The security application may reuse EDHOC protocol fields which therefore need to be available to the application. For example, the security application may use the same crypto algorithms as in the EDHOC session and therefore needs access to the selected cipher suite (or the whole SUITES_I). The application may use the ephemeral public keys G_X and G_Y, as ephemeral keys or as nonces, see [I-D.selander-lake-authz].

The processing of the EAD item (ead_label, ? ead_value) by the security application needs to be described in the specification where the ead_label is registered, see Section 10.5, including the optional ead_value for each message and actions in case of errors. An

application may support multiple security applications that make use of EAD, which may result in multiple EAD items in one EAD field, see Section 3.8. Any dependencies on security applications with previously registered EAD items needs to be documented, and the processing needs to consider their simultaneous use.

Since data carried in EAD may not be protected, or be processed by the application before the EDHOC message is verified, special considerations need to be made such that it does not violate security and privacy requirements of the service which uses this data, see Section 9.5. The content in an EAD item may impact the security properties provided by EDHOC. Security applications making use of the EAD items must perform the necessary security analysis.

Appendix F. Application Profile Example

This appendix contains a rudimentary example of an application profile, see Section 3.9.

For use of EDHOC with application X the following assumptions are made:

1. Transfer in CoAP as specified in Appendix A.2 with requests expected by the CoAP server (= Responder) at /app1-edh, no Content-Format needed.

2. METHOD = 1 (I uses signature key, R uses static DH key.)

3. CRED_I is an IEEE 802.1AR IDevID encoded as a C509 certificate of type 0 [I-D.ietf-cose-cbor-encoded-cert].

   * R acquires CRED_I out-of-band, indicated in EAD_1.

   * ID_CRED_I = {4: h''} is a 'kid' with value the empty CBOR byte string.

4. CRED_R is a CCS of type OKP as specified in Section 3.5.2.

   * The CBOR map has parameters 1 (kty), -1 (crv), and -2 (x-coordinate).

   * ID_CRED_R is {TBD2 : CCS}. Editor's note: TBD2 is the COSE header parameter value of 'kccs', see Section 10.6

5. External authorization data is defined and processed as specified in [I-D.selander-lake-authz].

6. EUI-64 is used as the identity of the endpoint (see example in Section 3.5.2).

7. No use of message_4: the application sends protected messages from R to I.

Appendix G. Long PLAINTEXT_2

By the definition of encryption of PLAINTEXT_2 with KEYSTREAM_2, it is limited to lengths of PLAINTEXT_2 not exceeding the output of EDHOC_KDF, see Section 4.1.2. If the EDHOC hash algorithm is SHA-2 then HKDF-Expand is used, which limits the length of the EDHOC_KDF output to 255 hash_length, where hash_length is the length of the output of the EDHOC hash algorithm given by the cipher suite. For example, with SHA-256 as EDHOC hash algorithm, the length of the hash output is 32 bytes and the maximum length of PLAINTEXT_2 is 255 32 = 8160 bytes.

While PLAINTEXT_2 is expected to be much shorter than 8 kB for the intended use cases, it seems nevertheless prudent to specify a solution for the event that this should turn out to be a limitation.

A potential work-around is to use a cipher suite with a different hash function. In particular, the use of KMAC removes all practical limitations in this respect.

This section specifies a solution which works with any hash function, by making use of multiple invocations of HKDF-Expand and negative values of info_label.

Consider the PLAINTEXT_2 partitioned in parts P(i) of length equal to M = 255 hash_length, except possibly the last part P(last) which has 0 < length M.

PLAINTEXT_2 = P(0) | P(1) | ... | P(last)

where | indicates concatenation.

The object is to define a matching KEYSTREAM_2 of the same length and perform the encryption in the same way as defined in Section 5.3.2:

CIPHERTEXT_2 = PLAINTEXT_2 XOR KEYSTREAM_2

Define the keystream as:

KEYSTREAM_2 = OKM(0) | OKM(1) | ... | OKM(last)

where

OKM(i) = EDHOC_KDF(PRK_2e, -i, TH_2, length(P(i)))

Note that if length(PLAINTEXT_2) M then P(0) = PLAINTEXT_2 and the definition of KEYSTREAM_2 = OKM(0) coincides with Figure 8.

This describes the processing of the Responder when sending message_2. The Initiator makes the same calculations when receiving message_2, but interchanging PLAINTEXT_2 and CIPHERTEXT_2.

An application profile may specify if it supports or not the method described in this appendix.

Appendix H. EDHOC_KeyUpdate

To provide forward secrecy in an even more efficient way than re-running EDHOC, this section specifies the optional function EDHOC_KeyUpdate in terms of EDHOC_KDF and PRK_out.

When EDHOC_KeyUpdate is called, a new PRK_out is calculated as a "hash" of the old PRK_out using the EDHOC_Expand function as illustrated by the following pseudocode. The change of PRK_out causes a change to PRK_exporter which enables the derivation of new application keys superseding the old ones, using EDHOC_Exporter, see Section 4.2.1.

```
EDHOC_KeyUpdate( context ):
   new PRK_out = EDHOC_KDF( old PRK_out, 11, context, hash_length )
   new PRK_exporter = EDHOC_KDF( new PRK_out, 10, h'', hash_length )
```

where hash_length denotes the output size in bytes of the EDHOC hash algorithm of the selected cipher suite.

The EDHOC_KeyUpdate takes a context as input to enable binding of the updated PRK_out to some event that triggered the key update. The Initiator and the Responder need to agree on the context, which can, e.g., be a counter, a pseudorandom number, or a hash. To provide forward secrecy the old PRK_out and keys derived from it (old PRK_exporter and old application keys) must be deleted as soon as they are not needed. When to delete the old keys and how to verify that they are not needed is up to the application.

An application using EDHOC_KeyUpdate needs to store PRK_out. Compromise of PRK_out leads to compromise of all keying material derived with the EDHOC_Exporter since the last invocation of the EDHOC_KeyUpdate function.

While this key update method provides forward secrecy it does not give as strong security properties as re-running EDHOC. EDHOC_KeyUpdate can be used to meet cryptographic limits and provide partial protection against key leakage, but it provides significantly weaker security properties than re-running EDHOC with ephemeral Diffie-Hellman. Even with frequent use of EDHOC_KeyUpdate, compromise of one session key compromises all future session keys, and an attacker therefore only needs to perform static key exfiltration [RFC7624], which is less complicated and has a lower risk profile than the dynamic case, see Section 9.1.

A similar method to do key update for OSCORE is KUDOS, see [I-D.ietf-core-oscore-key-update].

Appendix I. Example Protocol State Machine

This appendix describes an example protocol state machine for the Initiator and for the Responder. States are denoted in all capitals and parentheses denote actions taken only in some circumstances.

Note that this state machine is just an example, and that details of processing are omitted, for example:

* When error messages are being sent (with one exception)

* How credentials and EAD are processed by EDHOC and the application in the RCVD state

* What verifications are made, which includes not only MACs and signatures

I.1. Initiator State Machine

The Initiator sends message_1, triggering the state machine to transition from START to WAIT_M2, and waits for message_2.

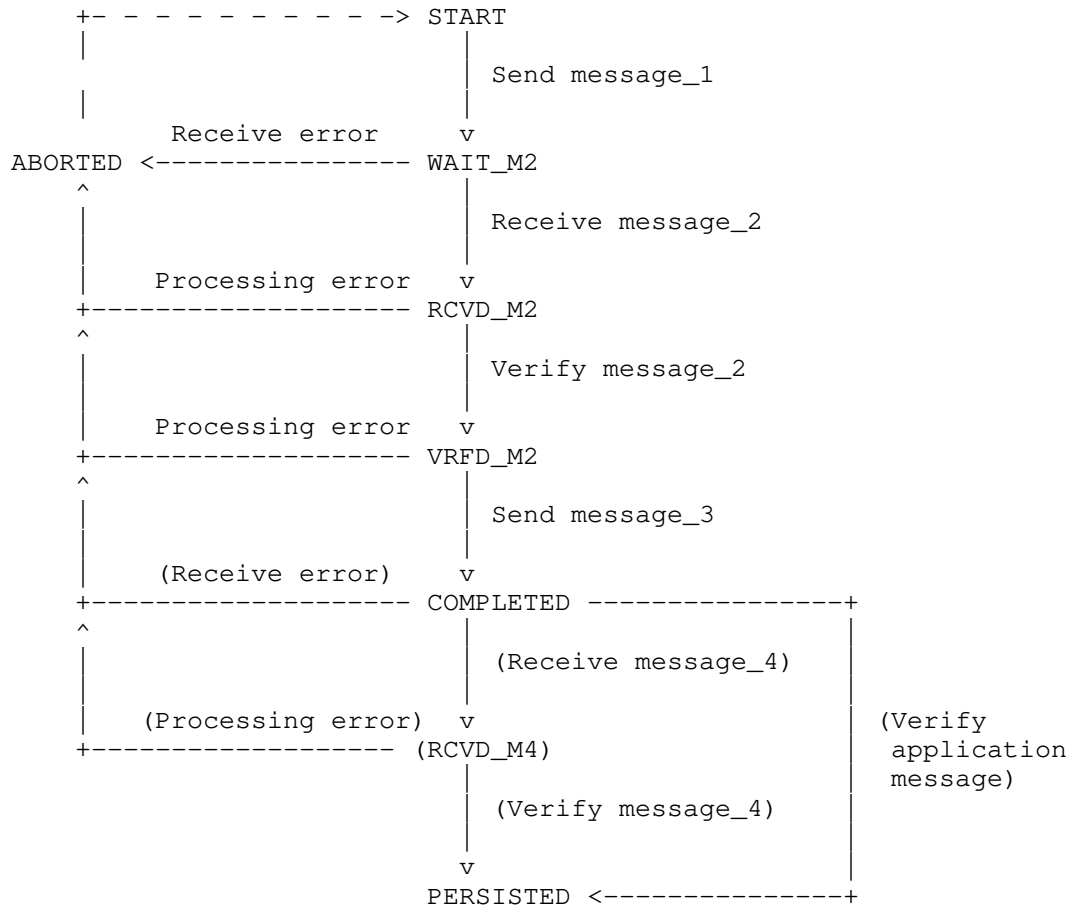
If the incoming message is an error message then the Initiator transitions from WAIT_M2 to ABORTED. In case of error code 2 (Wrong Selected Cipher Suite), the Initiator remembers the supported cipher suites for this particular Responder and transitions from ABORTED to START. The message_1 that the Initiator subsequently sends takes into account the cipher suites supported by the Responder.

Upon receiving a non-error message, the Initiator transitions from WAIT_M2 to RCVD_M2 and processes the message. If a processing error occurs on message_2, then the Initiator transitions from RCVD_M2 to ABORTED. In case of successful processing of message_2, the Initiator transitions from RCVD_M2 to VRFD_M2.

The Initiator prepares and processes message_3 for sending. If any processing error is encountered, the Initiator transitions from VRFD_M2 to ABORTED. If message_3 is successfully sent, the Initiator transitions from VRFD_M2 to COMPLETED.

If the application profile includes message_4, then the Initiator waits for message_4. If the incoming message is an error message then the Initiator transitions from COMPLETED to ABORTED. Upon receiving a non-error message, the Initiator transitions from COMPLETED (="WAIT_M4") to RCVD_M4 and processes the message. If a processing error occurs on message_4, then the Initiator transitions from RCVD_M4 to ABORTED. In case of successful processing of message_4, the Initiator transitions from RCVD_M4 to PERSISTED (="VRFD_M4").

If the application profile does not include message_4, then the Initiator waits for an incoming application message. If the decryption and verification of the application message is successful, then the Initiator transitions from COMPLETED to PERSISTED.



I.2. Responder State Machine

Upon receiving message_1, the Responder transitions from START to RCVD_M1.

If a processing error occurs on message_1, the Responder transitions from RCVD_M1 to ABORTED. This includes sending error message with error code 2 (Wrong Selected Cipher Suite) if the selected cipher suite in message_1 is not supported. In case of successful processing of message_1, the Responder transitions from RCVD_M1 to VRFD_M1.

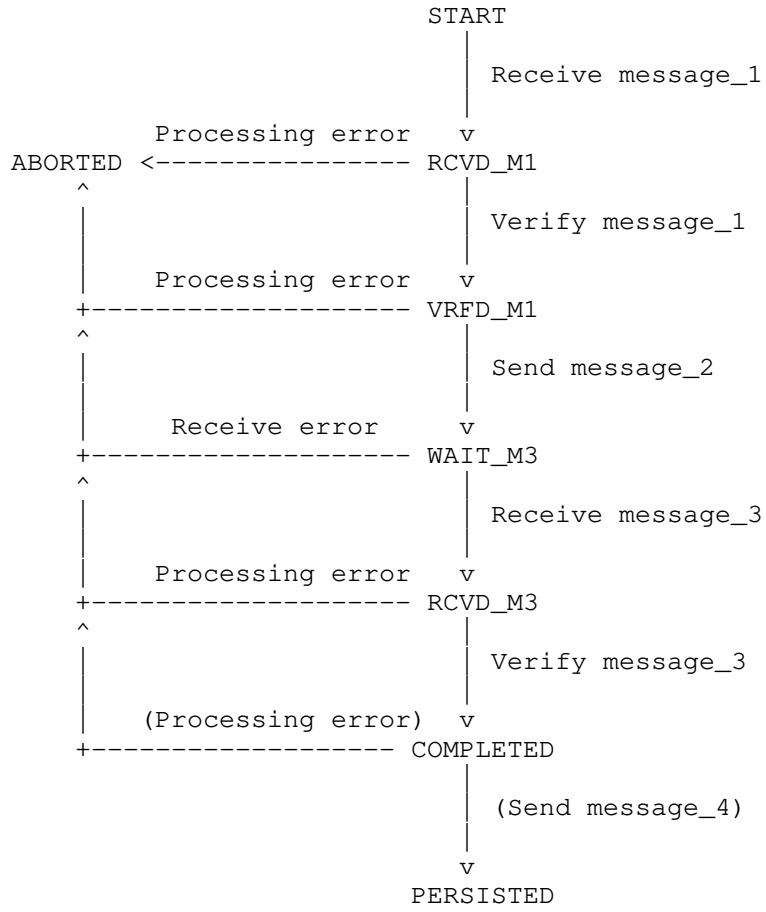
The Responder prepares and processes message_2 for sending. If any processing error is encountered, the Responder transitions from VRFD_M1 to ABORTED. If message_2 is successfully sent, the Initiator transitions from VRFD_M2 to WAIT_M3, and waits for message_3.

If the incoming message is an error message then the Responder transitions from WAIT_M3 to ABORTED.

Upon receiving message_3, the Responder transitions from WAIT_M3 to RCVD_M3. If a processing error occurs on message_3, the Responder transitions from RCVD_M3 to ABORTED. In case of successful processing of message_3, the Responder transitions from RCVD_M3 to COMPLETED (="VRFD_M3").

If the application profile includes message_4, the Responder prepares and processes message_4 for sending. If any processing error is encountered, the Responder transitions from COMPLETED to ABORTED.

If message_4 is successfully sent, or if the application profile does not include message_4, the Responder transitions from COMPLETED to PERSISTED.



Appendix J. Change Log

RFC Editor: Please remove this appendix.

* From -21 to -22
  - Normative text on transport capabilities.
* From -20 to -21
  - Recommendation to use chain instead of bag
  - Improved text about
    - denial-of-service
    - deriving secret and non-secret randomness from the same KDF instance
    - practical security against quantum computers
  - Clarifications, including
    - several updates section 3.4. Transport
    - descriptions in COSE IANA registration
    - encoding in Figure 5, reading of Figure 17
  - Removed term "dummy"
  - Harmonizing captions
  - Updated references
  - Acknowledgments
* From -19 to -20
  - C_R encrypted in message_2
  - C_R removed from TH_2
  - Error code for unknown referenced credential
  - Error code 0 (success) explicitly reserved
  - Message deduplication section moved from appendix to body

- Terminology
  - discontinued -> aborted
  - protocol run / exchange -> session
- Clarifications, in particular
  - when to derive application keys
  - the role of the application for authentication
- Security considerations for kccs and kcwt
- Updated references

* From -18 to -19
  - Clarifications:
    - Relation to SIGMA
    - Role of Static DH
    - Initiator and Responder roles
    - Transport properties
    - Construction of SUITES_I
    - Message correlation, new subsection 3.4.1, replacing former appendix H
    - Role of description about long PLAINTEXT_2
    - ead_label and ead_value
    - Message processing (Section 5)
    - Padding
    - Cipher suite negotiation example
  - Other updates:
    - Improved and stricter normative text in Appendix A

- Naming and separate sections for the two message flows in Appendix A: Forward/Reverse message flow,
- Table index style captions
- Aligning with COSE terminology: header map -> header_map
- Aligning terminology, use of "_" instead of "-"
- Prefixing "EDHOC_" to functions
- Updated list of security analysis papers
- New appendix with example state machine
- Acknowledgements
- Language improvements by native English speakers
- Updated IANA section with registration procedures
- New and updated references
- Removed appendix H

* From -17 to -18
  - Padding realised as EAD with ead_label = 0, PAD fields removed
  - Revised EAD syntax; ead is now EAD item; ead_value is now optional
  - Clarifications of
    - Identifier representation
    - Authentication credentials
    - RPK
    - Encoding of ID_CRED with kid
    - Representation of public keys, y-coordinate of ephemeral keys and validation
    - Processing after completed protocol
    - Making verifications available to the application

    - o Relation between EDHOC and OSCORE identifiers

  - Terminology alignment in particular session / protocol; discontinue / terminate

  - Updated CDDL

  - Additional unicode encodings

  - Large number of nits from WGLC

* From -16 to -17

  - EDHOC-KeyUpdate moved to appendix

  - Updated peer awareness properties based on SIGMA

  - Clarify use of random connection identifiers

  - Editorials related to appendix about messages with long PLAINTEXT_2

  - Updated acknowledgments (have we forgotten someone else? please send email)

* From -15 to -16

  - TH_2 used as salt in the derivation of PRK_2e

  - CRED_R/CRED_I included in TH_3/TH_4

  - Distinguish label used in info, exporter or elsewhere

  - New appendix for optional handling arbitrarily large message_2

    - o info_label type changed to int to support this

  - Updated security considerations

  - Implementation note about identifiers which are bstr/int

  - Clarifications, especifically about compact representation

  - Type bug fix in CDDL section

* From -14 to -15

  - Connection identifiers and key identifiers are now byte strings

    - Represented as CBOR bstr in the EDHOC message
      + Unless they happen to encode a one-byte CBOR int
    - More examples
  - EAD updates and details
    - Definition of EAD item
    - Definition of critical / non-critical EAD item
  - New section in Appendix D: Unauthenticated Operation
  - Clarifications
    - Lengths used in EDHOC-KDF
    - Key derivation from PRK_out
      + EDHOC-KeyUpdate and EDHOC-Exporter
    - Padding
  - Security considerations
    - When a change in a message is detected
    - Confidentiality in case of active attacks
    - Connection identifiers should be unpredictable
    - Maximum length of message_2
  - Minor bugs
* From -13 to -14
  - Merge of section 1.1 and 1.2
  - Connection and key identifiers restricted to be byte strings
  - Representation of byte strings as one-byte CBOR ints (-24..23)
  - Simplified mapping between EDHOC and OSCORE identifiers
  - Rewrite of 3.5

  - Clarification of authentication related operations performed by EDHOC
  - Authentication related verifications, including old section 3.5.1, moved to new appendix D

- Rewrite of 3.8
  - Move content about use of EAD to new appendix E
  - ead_value changed to bstr

- EDHOC-KDF updated
  - transcript_hash argument removed
  - TH included in context argument
  - label argument is now type uint, all labels replaced

- Key schedule updated
  - New salts derived to avoid reuse of same key with expand and extract
  - PRK_4x3m renamed PRK_4e3m
  - K_4 and IV_4 derived from PRK_4e3m
  - New PRK: PRK_out derived from PRK_4e3m and TH_4
  - Clarified main output of EDHOC is the shared secret PRK_out
  - Exporter defined by EDHOC-KDF and new PRK PRK_exporter derived from PRK_out
  - Key update defined by Expand instead of Extract

- All applications of EDHOC-KDF in one place

- Update of processing
  - EAD and ID_CRED passed to application when available
  - identity verification and credential retrieval omitted in protocol description

    - Transcript hash defined by plaintext messages instead of ciphertext
    - Changed order of input to TH_2
    - Removed general G_X checking against selfie-attacks
  - Support for padding of plaintext
  - Updated compliance requirements
  - Updated security considerations
    - Updated and more clear requirements on MAC length
    - Clarification of key confirmation
    - Forbid use of same key for signature and static DH
  - Updated appendix on message deduplication
  - Clarifications of
    - connection identifiers
    - cipher suites, including negotiation
    - EAD
    - Error messages
  - Updated media types
  - Applicability template renamed application profile
  - Editorials
* From -12 to -13
  - no changes
* From -12:
  - Shortened labels to derive OSCORE key and salt
  - ead_value changed to bstr
  - Removed general G_X checking against selfie-attacks

- Updated and more clear requirements on MAC length
- Clarifications from Kathleen, Stephen, Marco, Sean, Stefan,
- Authentication Related Verifications moved to appendix
- Updated MTI section and cipher suite
- Updated security considerations

* From -11 to -12:
  - Clarified applicability to KEMs
  - Clarified use of COSE header parameters
  - Updates on MTI
  - Updated security considerations
  - New section on PQC
  - Removed duplicate definition of cipher suites
  - Explanations of use of COSE moved to Appendix C.3
  - Updated internal references
* From -10 to -11:
  - Restructured section on authentication parameters
  - Changed UCCS to CCS
  - Changed names and description of COSE header parameters for CWT/CCS
  - Changed several of the KDF and Exporter labels
  - Removed edhoc_aead_id from info (already in transcript_hash)
  - Added MTI section
  - EAD: changed CDDL names and added value type to registry
  - Updated Figures 1, 2, and 3
  - Some correction and clarifications

- Added core.edhoc to CoRE Resource Type registry

* From -09 to -10:

  - SUITES_I simplified to only contain the selected and more preferred suites

  - Info is a CBOR sequence and context is a bstr

  - Added kid to UCCS example

  - Separate header parameters for CWT and UCCS

  - CWT Confirmation Method kid extended to bstr / int

* From -08 to -09:

  - G_Y and CIPHERTEXT_2 are now included in one CBOR bstr

  - MAC_2 and MAC_3 are now generated with EDHOC-KDF

  - Info field context is now general and explicit in EDHOC-KDF

  - Restructured Section 4, Key Derivation

  - Added EDHOC MAC length to cipher suite for use with static DH

  - More details on the use of CWT and UCCS

  - Restructured and clarified Section 3.5, Authentication Parameters

  - Replaced 'kid2' with extension of 'kid'

  - EAD encoding now supports multiple ead types in one message

  - Clarified EAD type

  - Updated message sizes

  - Replaced perfect forward secrecy with forward secrecy

  - Updated security considerations

  - Replaced prepended 'null' with 'true' in the CoAP transport of message_1

  - Updated CDDL definitions

- Expanded on the use of COSE

* From -07 to -08:

  - Prepended C_x moved from the EDHOC protocol itself to the transport mapping

  - METHOD_CORR renamed to METHOD, corr removed

  - Removed bstr_identifier and use bstr / int instead; C_x can now be int without any implied bstr semantics

  - Defined COSE header parameter 'kid2' with value type bstr / int for use with ID_CRED_x

  - Updated message sizes

  - New cipher suites with AES-GCM and ChaCha20 / Poly1305

  - Changed from one- to two-byte identifier of CNSA compliant suite

  - Separate sections on transport and connection id with further sub-structure

  - Moved back key derivation for OSCORE from draft-ietf-core-oscore-edhoc

  - OSCORE and CoAP specific processing moved to new appendix

  - Message 4 section moved to message processing section

* From -06 to -07:

  - Changed transcript hash definition for TH_2 and TH_3

  - Removed "EDHOC signature algorithm curve" from cipher suite

  - New IANA registry "EDHOC Exporter Label"

  - New application defined parameter "context" in EDHOC-Exporter

  - Changed normative language for failure from MUST to SHOULD send error

  - Made error codes non-negative and 0 for success

  - Added detail on success error code

- Aligned terminology "protocol instance" -> "session"
- New appendix on compact EC point representation
- Added detail on use of ephemeral public keys
- Moved key derivation for OSCORE to draft-ietf-core-oscore-edhoc
- Additional security considerations
- Renamed "Auxililary Data" as "External Authorization Data"
- Added encrypted EAD_4 to message_4

* From -05 to -06:
  - New section 5.2 "Message Processing Outline"
  - Optional inital byte C_1 = null in message_1
  - New format of error messages, table of error codes, IANA registry
  - Change of recommendation transport of error in CoAP
  - Merge of content in 3.7 and appendix C into new section 3.7 "Applicability Statement"
  - Requiring use of deterministic CBOR
  - New section on message deduplication
  - New appendix containin all CDDL definitions
  - New appendix with change log
  - Removed section "Other Documents Referencing EDHOC"
  - Clarifications based on review comments
* From -04 to -05:
  - EDHOC-Rekey-FS -> EDHOC-KeyUpdate
  - Clarification of cipher suite negotiation
  - Updated security considerations

- Updated test vectors
- Updated applicability statement template

* From -03 to -04:
  - Restructure of section 1
  - Added references to C509 Certificates
  - Change in CIPHERTEXT_2 -> plaintext XOR KEYSTREAM_2 (test vector not updated)
  - "K_2e", "IV_2e" -> KEYSTREAM_2
  - Specified optional message 4
  - EDHOC-Exporter-FS -> EDHOC-Rekey-FS
  - Less constrained devices SHOULD implement both suite 0 and 2
  - Clarification of error message
  - Added exporter interface test vector
* From -02 to -03:
  - Rearrangements of section 3 and beginning of section 4
  - Key derivation new section 4
  - Cipher suites 4 and 5 added
  - EDHOC-EXPORTER-FS - generate a new PRK_4x3m from an old one
  - Change in CIPHERTEXT_2 -> COSE_Encrypt0 without tag (no change to test vector)
  - Clarification of error message
  - New appendix C applicability statement
* From -01 to -02:
  - New section 1.2 Use of EDHOC
  - Clarification of identities

- New section 4.3 clarifying bstr_identifier
- Updated security considerations
- Updated text on cipher suite negotiation and key confirmation
- Test vector for static DH

* From -00 to -01:
  - Removed PSK method
  - Removed references to certificate by value

Acknowledgments

The authors want to thank Christian Amsüss, Alessandro Bruni, Karthikeyan Bhargavan, Carsten Bormann, Timothy Claeys, Baptiste Cottier, Roman Danyliw, Martin Disch, Martin Duke, Donald Eastlake, Lars Eggert, Stephen Farrell, Loïc Ferreira, Theis Grønbech Petersen, Felix Günther, Dan Harkins, Klaus Hartke, Russ Housley, Stefan Hristozov, Marc Ilunga, Charlie Jacomme, Elise Klein, Erik Kline, Steve Kremer, Alexandros Krontiris, Ilari Liusvaara, Rafa Marín-López, Kathleen Moriarty, David Navarro, Karl Norrman, Salvador Pérez, Radia Perlman, David Pointcheval, Maïwenn Racouchot, Eric Rescorla, Michael Richardson, Thorvald Sahl Jørgensen, Zaheduzzaman Sarker, Jim Schaad, Michael Scharf, Carsten Schürmann, John Scudder, Ludwig Seitz, Brian Sipos, Stanislav Smyshlyaev, Valery Smyslov, Peter van der Stok, Rene Struik, Vaishnavi Sundararajan, Erik Thormarker, Marco Tiloca, Sean Turner, Michel Veillette, Malia Vuini, Paul Wouters, and Lei Yan for reviewing and commenting on intermediate versions of the draft. We are especially indebted to the late Jim Schaad for his continuous reviewing and implementation of early versions of this and other drafts.

Work on this document has in part been supported by the H2020 project SIFIS-Home (grant agreement 952652).

Authors' Addresses

Göran Selander
Ericsson AB
SE-164 80 Stockholm
Sweden
Email: goran.selander@ericsson.com

John Preuß Mattsson
Ericsson AB
SE-164 80 Stockholm
Sweden
Email: john.mattsson@ericsson.com

Francesca Palombini
Ericsson AB
SE-164 80 Stockholm
Sweden
Email: francesca.palombini@ericsson.com

LAKE Working Group
Internet-Draft
Intended status: Informational
Expires: 30 July 2024

G. Selander
J. Preuß Mattsson
Ericsson
M. Serafin
ASSA ABLOY
M. Tiloca
RISE
M. Vuini
Inria

27 January 2024

Traces of EDHOC
draft-ietf-lake-traces-09

Abstract

This document contains some example traces of Ephemeral Diffie-Hellman Over COSE (EDHOC).

Status of This Memo

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

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

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

This Internet-Draft will expire on 30 July 2024.

Copyright Notice

Table of Contents

1. Introduction

EDHOC [I-D.ietf-lake-edhoc] is a lightweight authenticated key exchange protocol designed for highly constrained settings. This document contains annotated traces of EDHOC sessions, with input, output, and intermediate processing results to simplify testing of implementations. The traces have been verified by two independent implementations.

1.1. Setup

EDHOC is run between an Initiator (I) and a Responder (R). The private/public key pairs and credentials of the Initiator and the Responder required to produce the protocol messages are shown in the traces when needed for the calculations.

EDHOC messages and intermediate results are encoded in CBOR [RFC8949] and can therefore be displayed in CBOR diagnostic notation using, e.g., the CBOR playground [CborMe], which makes them easy to parse for humans. Credentials can also be encoded in CBOR, e.g. CBOR Web Tokens (CWT) [RFC8392].

The document contains two traces:

* Section 2 - Authentication with signature keys identified by the hash value of the X.509 certificates (provided in Section 2.8). The endpoints use EdDSA [RFC8032] for authentication and X25519 [RFC7748] for ephemeral-ephemeral Diffie-Hellman key exchange.

* Section 3 - Authentication with static Diffie-Hellman keys identified by short key identifiers labelling CWT Claim Sets (CCSs) [RFC8392]. The endpoints use NIST P-256 [SP-800-186] for both ephemeral-ephemeral and static-ephemeral Diffie-Hellman key exchange. This trace also illustrates the cipher suite negotiation, and provides an example of low protocol overhead, with messages sizes of (39, 45, 19) bytes.

Examples of invalid EDHOC messages are found in Section 4.

NOTE 1. The same name is used for hexadecimal byte strings and their CBOR encodings. The traces contain both the raw byte strings and the corresponding CBOR encoded data items.

NOTE 2. If not clear from the context, remember that CBOR sequences and CBOR arrays assume CBOR encoded data items as elements.

NOTE 3. When the protocol transporting EDHOC messages does not inherently provide correlation across all messages, like CoAP [RFC7252], then some messages typically are prepended with connection identifiers and potentially a message_1 indicator (see Sections 3.4.1 and A.2 of [I-D.ietf-lake-edhoc]). Those bytes are not included in the traces in this document.

1.2. Terminology and Requirements Language

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

2. Authentication with Signatures, X.509 Certificates Identified by 'x5t'

In this example the Initiator (I) and Responder (R) are authenticated with digital signatures (METHOD = 0). Both the Initiator and the Responder support cipher suite 0, which determines the algorithms:

* EDHOC AEAD algorithm = AES-CCM-16-64-128
* EDHOC hash algorithm = SHA-256
* EDHOC MAC length in bytes (Static DH) = 8
* EDHOC key exchange algorithm (ECDH curve) = X25519
* EDHOC signature algorithm = EdDSA
* Application AEAD algorithm = AES-CCM-16-64-128
* Application hash algorithm = SHA-256

The public keys are represented with X.509 certificates identified by the COSE header parameter 'x5t'.

2.1. message_1

Both endpoints are authenticated with signatures, i.e., METHOD = 0:

METHOD (CBOR Data Item) (1 byte)
00

The Initiator selects cipher suite 0. A single cipher suite is encoded as an int:

SUITES_I (CBOR Data Item) (1 byte)
00

The Initiator creates an ephemeral key pair for use with the EDHOC key exchange algorithm:

Initiator's ephemeral private key
X (Raw Value) (32 bytes)
89 2e c2 8e 5c b6 66 91 08 47 05 39 50 0b 70 5e 60 d0 08 d3 47 c5 81 7e e9 f3 32 7c 8a 87 bb 03

Initiator's ephemeral public key
G_X (Raw Value) (32 bytes)
31 f8 2c 7b 5b 9c bb f0 f1 94 d9 13 cc 12 ef 15 32 d3 28 ef 32 63 2a 48 81 a1 c0 70 1e 23 7f 04

Initiator's ephemeral public key
G_X (CBOR Data Item) (34 bytes)
58 20 31 f8 2c 7b 5b 9c bb f0 f1 94 d9 13 cc 12 ef 15 32 d3 28 ef 32 63 2a 48 81 a1 c0 70 1e 23 7f 04

The Initiator selects its connection identifier C_I to be the byte string 0x2d, which since it is represented by the 1-byte CBOR int -14 is encoded as 0x2d:

Connection identifier chosen by Initiator
C_I (Raw Value) (1 byte)
2d

Connection identifier chosen by Initiator
C_I (CBOR Data Item) (1 byte)
2d

No external authorization data:

EAD_1 (CBOR Sequence) (0 bytes)

The Initiator constructs message_1:

```
message_1 =
(
 0,
 0,
 h'31f82c7b5b9cbbf0f194d913cc12ef1532d328ef32632a48
   81a1c0701e237f04',
 -14
)
```

message_1 (CBOR Sequence) (37 bytes)
00 00 58 20 31 f8 2c 7b 5b 9c bb f0 f1 94 d9 13 cc 12 ef 15 32 d3 28
ef 32 63 2a 48 81 a1 c0 70 1e 23 7f 04 2d

2.2. message_2

The Responder supports the most preferred and selected cipher suite 0, so SUITES_I is acceptable.

The Responder creates an ephemeral key pair for use with the EDHOC key exchange algorithm:

Responder’s ephemeral private key
Y (Raw Value) (32 bytes)
e6 9c 23 fb f8 1b c4 35 94 24 46 83 7f e8 27 bf 20 6c 8f a1 0a 39 db
47 44 9e 5a 81 34 21 e1 e8

Responder’s ephemeral public key
G_Y (Raw Value) (32 bytes)
dc 88 d2 d5 1d a5 ed 67 fc 46 16 35 6b c8 ca 74 ef 9e be 8b 38 7e 62
3a 36 0b a4 80 b9 b2 9d 1c

Responder’s ephemeral public key
G_Y (CBOR Data Item) (34 bytes)
58 20 dc 88 d2 d5 1d a5 ed 67 fc 46 16 35 6b c8 ca 74 ef 9e be 8b 38
7e 62 3a 36 0b a4 80 b9 b2 9d 1c

The Responder selects its connection identifier C_R to be the byte string 0x18, which since it is not represented as a 1-byte CBOR int is encoded as h’18’ = 0x4118:

Connection identifier chosen by Responder
C_R (Raw Value) (1 byte)
18

Connection identifier chosen by Responder
C_R (CBOR Data Item) (2 bytes)
41 18

The transcript hash TH_2 is calculated using the EDHOC hash algorithm:

TH_2 = H(G_Y, H(message_1))

H(message_1) (Raw Value) (32 bytes)
c1 65 d6 a9 9d 1b ca fa ac 8d bf 2b 35 2a 6f 7d 71 a3 0b 43 9c 9d 64
d3 49 a2 38 48 03 8e d1 6b

H(message_1) (CBOR Data Item) (34 bytes)
58 20 c1 65 d6 a9 9d 1b ca fa ac 8d bf 2b 35 2a 6f 7d 71 a3 0b 43 9c
9d 64 d3 49 a2 38 48 03 8e d1 6b

The input to calculate TH_2 is the CBOR sequence:

G_Y, H(message_1)

Input to calculate TH_2 (CBOR Sequence) (68 bytes)
58 20 dc 88 d2 d5 1d a5 ed 67 fc 46 16 35 6b c8 ca 74 ef 9e be 8b 38
7e 62 3a 36 0b a4 80 b9 b2 9d 1c 58 20 c1 65 d6 a9 9d 1b ca fa ac 8d
bf 2b 35 2a 6f 7d 71 a3 0b 43 9c 9d 64 d3 49 a2 38 48 03 8e d1 6b

TH_2 (Raw Value) (32 bytes)
c6 40 5c 15 4c 56 74 66 ab 1d f2 03 69 50 0e 54 0e 9f 14 bd 3a 79 6a
06 52 ca e6 6c 90 61 68 8d

TH_2 (CBOR Data Item) (34 bytes)
58 20 c6 40 5c 15 4c 56 74 66 ab 1d f2 03 69 50 0e 54 0e 9f 14 bd 3a
79 6a 06 52 ca e6 6c 90 61 68 8d

PRK_2e is specified in Section 4.1.1.1 of [I-D.ietf-lake-edhoc].

First, the ECDH shared secret G_XY is computed from G_X and Y, or G_Y
and X:

G_XY (Raw Value) (ECDH shared secret) (32 bytes)
e5 cd f3 a9 86 cd ac 5b 7b f0 46 91 e2 b0 7c 08 e7 1f 53 99 8d 8f 84
2b 7c 3f b4 d8 39 cf 7b 28

Then, PRK_2e is calculated using EDHOC_Extract() determined by the
EDHOC hash algorithm:

PRK_2e = EDHOC_Extract(salt, G_XY) =
       = HMAC-SHA-256(salt, G_XY)

where salt is TH_2:

salt (Raw Value) (32 bytes)
c6 40 5c 15 4c 56 74 66 ab 1d f2 03 69 50 0e 54 0e 9f 14 bd 3a 79 6a
06 52 ca e6 6c 90 61 68 8d

PRK_2e (Raw Value) (32 bytes)
d5 84 ac 2e 5d ad 5a 77 d1 4b 53 eb e7 2e f1 d5 da a8 86 0d 39 93 73
bf 2c 24 0a fa 7b a8 04 da

Since METHOD = 0, the Responder authenticates using signatures. Since the selected cipher suite is 0, the EDHOC signature algorithm is EdDSA.

The Responder's signature key pair using EdDSA:

Responder's private authentication key
SK_R (Raw Value) (32 bytes)
ef 14 0f f9 00 b0 ab 03 f0 c0 8d 87 9c bb d4 b3 1e a7 1e 6e 7e e7 ff
cb 7e 79 55 77 7a 33 27 99

Responder's public authentication key
PK_R (Raw Value) (32 bytes)
a1 db 47 b9 51 84 85 4a d1 2a 0c 1a 35 4e 41 8a ac e3 3a a0 f2 c6 62
c0 0b 3a c5 5d e9 2f 93 59

PRK_3e2m is specified in Section 4.1.1.2 of [I-D.ietf-lake-edhoc].

Since the Responder authenticates with signatures PRK_3e2m = PRK_2e.

PRK_3e2m (Raw Value) (32 bytes)
d5 84 ac 2e 5d ad 5a 77 d1 4b 53 eb e7 2e f1 d5 da a8 86 0d 39 93 73
bf 2c 24 0a fa 7b a8 04 da

The Responder constructs the remaining input needed to calculate MAC_2:

MAC_2 = EDHOC_KDF(PRK_3e2m, 2, context_2, mac_length_2)

context_2 = << C_R, ID_CRED_R, TH_2, CRED_R, ? EAD_2 >>

CRED_R is identified by a 64-bit hash:

ID_CRED_R =
{
  34 : [-15, h'79f2a41b510c1f9b']
}

where the COSE header value 34 ('x5t') indicates a hash of an X.509 certficate, and the COSE algorithm -15 indicates the hash algorithm SHA-256 truncated to 64 bits.

ID_CRED_R (CBOR Data Item) (14 bytes)
a1 18 22 82 2e 48 79 f2 a4 1b 51 0c 1f 9b

CRED_R is a CBOR byte string of the DER encoding of the X.509 certificate in Section 2.8.1:

CRED_R (Raw Value) (241 bytes)

```
30 81 ee 30 81 a1 a0 03 02 01 02 02 04 62 31 9e c4 30 05 06 03 2b 65
70 30 1d 31 1b 30 19 06 03 55 04 03 0c 12 45 44 48 4f 43 20 52 6f 6f
74 20 45 64 32 35 35 31 39 30 1e 17 0d 32 32 30 33 31 36 30 38 32 34
33 36 5a 17 0d 32 39 31 32 33 31 32 33 30 30 30 30 5a 30 22 31 20 30
1e 06 03 55 04 03 0c 17 45 44 48 4f 43 20 52 65 73 70 6f 6e 64 65 72
20 45 64 32 35 35 31 39 30 2a 30 05 06 03 2b 65 70 03 21 00 a1 db 47
b9 51 84 85 4a d1 2a 0c 1a 35 4e 41 8a ac e3 3a a0 f2 c6 62 c0 0b 3a
c5 5d e9 2f 93 59 30 05 06 03 2b 65 70 03 41 00 b7 23 bc 01 ea b0 92
8e 8b 2b 6c 98 de 19 cc 38 23 d4 6e 7d 69 87 b0 32 47 8f ec fa f1 45
37 a1 af 14 cc 8b e8 29 c6 b7 30 44 10 18 37 eb 4a bc 94 95 65 d8 6d
ce 51 cf ae 52 ab 82 c1 52 cb 02
```

CRED_R (CBOR Data Item) (243 bytes)

```
58 f1 30 81 ee 30 81 a1 a0 03 02 01 02 02 04 62 31 9e c4 30 05 06 03
2b 65 70 30 1d 31 1b 30 19 06 03 55 04 03 0c 12 45 44 48 4f 43 20 52
6f 6f 74 20 45 64 32 35 35 31 39 30 1e 17 0d 32 32 30 33 31 36 30 38
32 34 33 36 5a 17 0d 32 39 31 32 33 31 32 33 30 30 30 30 5a 30 22 31
20 30 1e 06 03 55 04 03 0c 17 45 44 48 4f 43 20 52 65 73 70 6f 6e 64
65 72 20 45 64 32 35 35 31 39 30 2a 30 05 06 03 2b 65 70 03 21 00 a1
db 47 b9 51 84 85 4a d1 2a 0c 1a 35 4e 41 8a ac e3 3a a0 f2 c6 62 c0
0b 3a c5 5d e9 2f 93 59 30 05 06 03 2b 65 70 03 41 00 b7 23 bc 01 ea
b0 92 8e 8b 2b 6c 98 de 19 cc 38 23 d4 6e 7d 69 87 b0 32 47 8f ec fa
f1 45 37 a1 af 14 cc 8b e8 29 c6 b7 30 44 10 18 37 eb 4a bc 94 95 65
d8 6d ce 51 cf ae 52 ab 82 c1 52 cb 02
```

No external authorization data:

EAD_2 (CBOR Sequence) (0 bytes)

context_2 = << C_R, ID_CRED_R, TH_2, CRED_R, ? EAD_2 >>

context_2 (CBOR Sequence) (293 bytes)

```
41 18 a1 18 22 82 2e 48 79 f2 a4 1b 51 0c 1f 9b 58 20 c6 40 5c 15 4c
56 74 66 ab 1d f2 03 69 50 0e 54 0e 9f 14 bd 3a 79 6a 06 52 ca e6 6c
90 61 68 8d 58 f1 30 81 ee 30 81 a1 a0 03 02 01 02 02 04 62 31 9e c4
30 05 06 03 2b 65 70 30 1d 31 1b 30 19 06 03 55 04 03 0c 12 45 44 48
4f 43 20 52 6f 6f 74 20 45 64 32 35 35 31 39 30 1e 17 0d 32 32 30 33
31 36 30 38 32 34 33 36 5a 17 0d 32 39 31 32 33 31 32 33 30 30 30 30
5a 30 22 31 20 30 1e 06 03 55 04 03 0c 17 45 44 48 4f 43 20 52 65 73
70 6f 6e 64 65 72 20 45 64 32 35 35 31 39 30 2a 30 05 06 03 2b 65 70
03 21 00 a1 db 47 b9 51 84 85 4a d1 2a 0c 1a 35 4e 41 8a ac e3 3a a0
f2 c6 62 c0 0b 3a c5 5d e9 2f 93 59 30 05 06 03 2b 65 70 03 41 00 b7
23 bc 01 ea b0 92 8e 8b 2b 6c 98 de 19 cc 38 23 d4 6e 7d 69 87 b0 32
47 8f ec fa f1 45 37 a1 af 14 cc 8b e8 29 c6 b7 30 44 10 18 37 eb 4a
bc 94 95 65 d8 6d ce 51 cf ae 52 ab 82 c1 52 cb 02
```

```
context_2 (CBOR byte string) (296 bytes)
59 01 25 41 18 a1 18 22 82 2e 48 79 f2 a4 1b 51 0c 1f 9b 58 20 c6 40
5c 15 4c 56 74 66 ab 1d f2 03 69 50 0e 54 0e 9f 14 bd 3a 79 6a 06 52
ca e6 6c 90 61 68 8d 58 f1 30 81 ee 30 81 a1 a0 03 02 01 02 02 04 62
31 9e c4 30 05 06 03 2b 65 70 30 1d 31 1b 30 19 06 03 55 04 03 0c 12
45 44 48 4f 43 20 52 6f 6f 74 20 45 64 32 35 35 31 39 30 1e 17 0d 32
32 30 33 31 36 30 38 32 34 33 36 5a 17 0d 32 39 31 32 33 31 32 33 30
30 30 30 5a 30 22 31 20 30 1e 06 03 55 04 03 0c 17 45 44 48 4f 43 20
52 65 73 70 6f 6e 64 65 72 20 45 64 32 35 35 31 39 30 2a 30 05 06 03
2b 65 70 03 21 00 a1 db 47 b9 51 84 85 4a d1 2a 0c 1a 35 4e 41 8a ac
e3 3a a0 f2 c6 62 c0 0b 3a c5 5d e9 2f 93 59 30 05 06 03 2b 65 70 03
41 00 b7 23 bc 01 ea b0 92 8e 8b 2b 6c 98 de 19 cc 38 23 d4 6e 7d 69
87 b0 32 47 8f ec fa f1 45 37 a1 af 14 cc 8b e8 29 c6 b7 30 44 10 18
37 eb 4a bc 94 95 65 d8 6d ce 51 cf ae 52 ab 82 c1 52 cb 02
```

MAC_2 is computed through EDHOC_Expand() using the EDHOC hash algorithm, see Section 4.1.2 of [I-D.ietf-lake-edhoc]:

MAC_2 = HKDF-Expand(PRK_3e2m, info, mac_length_2), where

info = (2, context_2, mac_length_2)

Since METHOD = 0, mac_length_2 is given by the EDHOC hash algorithm.

info for MAC_2 is:

```
info =
(
 2,
 h'4118a11822822e4879f2a41b510c1f9b5820c6405c154c56
   7466ab1df20369500e540e9f14bd3a796a0652cae66c9061
   688d58f13081ee3081a1a003020102020462319ec4300506
   032b6570301d311b30190603550403 0c124544484f432052
   6f6f74204564323535313930 1e170d323230333136303832
   3433365a170d323933323331323330303030 5a3022312030
   1e06035504030c174544484f4320526573706f6e64657220
   45643235353139302a300506032b6570032100a1db47b951
   84854ad12a0c1a354e418aace33aa0f2c662c00b3ac55de9
   2f9359300506032b6570034100b723bc01eab0928e8b2b6c
   98de19cc3823d46e7d6987b032478fecfaf14537a1af14cc
   8be829c6b73044101837eb4abc949565d86dce51cfae52ab
   82c152cb02',
 32
)
```

where the last value is the output size of the EDHOC hash algorithm in bytes.

info for MAC_2 (CBOR Sequence) (299 bytes)
02 59 01 25 41 18 a1 18 22 82 2e 48 79 f2 a4 1b 51 0c 1f 9b 58 20 c6
40 5c 15 4c 56 74 66 ab 1d f2 03 69 50 0e 54 0e 9f 14 bd 3a 79 6a 06
52 ca e6 6c 90 61 68 8d 58 f1 30 81 ee 30 81 a1 a0 03 02 01 02 02 04
62 31 9e c4 30 05 06 03 2b 65 70 30 1d 31 1b 30 19 06 03 55 04 03 0c
12 45 44 48 4f 43 20 52 6f 6f 74 20 45 64 32 35 35 31 39 30 1e 17 0d
32 32 30 33 31 36 30 38 32 34 33 36 5a 17 0d 32 39 31 32 33 31 32 33
30 30 30 30 5a 30 22 31 20 30 1e 06 03 55 04 03 0c 17 45 44 48 4f 43
20 52 65 73 70 6f 6e 64 65 72 20 45 64 32 35 35 31 39 30 2a 30 05 06
03 2b 65 70 03 21 00 a1 db 47 b9 51 84 85 4a d1 2a 0c 1a 35 4e 41 8a
ac e3 3a a0 f2 c6 62 c0 0b 3a c5 5d e9 2f 93 59 30 05 06 03 2b 65 70
03 41 00 b7 23 bc 01 ea b0 92 8e 8b 2b 6c 98 de 19 cc 38 23 d4 6e 7d
69 87 b0 32 47 8f ec fa f1 45 37 a1 af 14 cc 8b e8 29 c6 b7 30 44 10
18 37 eb 4a bc 94 95 65 d8 6d ce 51 cf ae 52 ab 82 c1 52 cb 02 18 20

MAC_2 (Raw Value) (32 bytes)
86 2a 7e 5e f1 47 f9 a5 f4 c5 12 e1 b6 62 3c d6 6c d1 7a 72 72 07 2b
fe 5b 60 2f fe 30 7e e0 e9

MAC_2 (CBOR Data Item) (34 bytes)
58 20 86 2a 7e 5e f1 47 f9 a5 f4 c5 12 e1 b6 62 3c d6 6c d1 7a 72 72
07 2b fe 5b 60 2f fe 30 7e e0 e9

Since METHOD = 0, Signature_or_MAC_2 is the 'signature' of the COSE_Sign1 object.

The Responder constructs the message to be signed:

```
[ "Signature1", << ID_CRED_R >>,
 << TH_2, CRED_R, ? EAD_2 >>, MAC_2 ] =

[
 "Signature1",
 h'a11822822e4879f2a41b510c1f9b',
 h'5820c6405c154c567466ab1df20369500e540e9f14bd3a79
   6a0652cae66c9061688d58f13081ee3081a1a00302010202
   0462319ec4300506032b6570301d311b301906035504030c
   124544484f4320526f6f742045643235353139301e170d32
   32303331363038323433365a170d32393132333132333030
   30305a30223120301e06035504030c174544484f43205265
   73706f6e646572204564323535313930 2a300506032b6570
   032100a1db47b95184854ad12a0c1a354e418aace33aa0f2
   c662c00b3ac55de92f9359300506032b6570034100b723bc
   01eab0928e8b2b6c98de19cc3823d46e7d6987b032478fec
   faf14537a1af14cc8be829c6b73044101837eb4abc949565
   d86dce51cfae52ab82c152cb02',
 h'862a7e5ef147f9a5f4c512e1b6623cd66cd17a7272072bfe
   5b602ffe307ee0e9'
]
```

Message to be signed 2 (CBOR Data Item) (341 bytes)

```
84 6a 53 69 67 6e 61 74 75 72 65 31 4e a1 18 22 82 2e 48 79 f2 a4 1b
51 0c 1f 9b 59 01 15 58 20 c6 40 5c 15 4c 56 74 66 ab 1d f2 03 69 50
0e 54 0e 9f 14 bd 3a 79 6a 06 52 ca e6 6c 90 61 68 8d 58 f1 30 81 ee
30 81 a1 a0 03 02 01 02 02 04 62 31 9e c4 30 05 06 03 2b 65 70 30 1d
31 1b 30 19 06 03 55 04 03 0c 12 45 44 48 4f 43 20 52 6f 6f 74 20 45
64 32 35 35 31 39 30 1e 17 0d 32 32 30 33 31 36 30 38 32 34 33 36 5a
17 0d 32 39 31 32 33 31 32 33 30 30 30 30 5a 30 22 31 20 30 1e 06 03
55 04 03 0c 17 45 44 48 4f 43 20 52 65 73 70 6f 6e 64 65 72 20 45 64
32 35 35 31 39 30 2a 30 05 06 03 2b 65 70 03 21 00 a1 db 47 b9 51 84
85 4a d1 2a 0c 1a 35 4e 41 8a ac e3 3a a0 f2 c6 62 c0 0b 3a c5 5d e9
2f 93 59 30 05 06 03 2b 65 70 03 41 00 b7 23 bc 01 ea b0 92 8e 8b 2b
6c 98 de 19 cc 38 23 d4 6e 7d 69 87 b0 32 47 8f ec fa f1 45 37 a1 af
14 cc 8b e8 29 c6 b7 30 44 10 18 37 eb 4a bc 94 95 65 d8 6d ce 51 cf
ae 52 ab 82 c1 52 cb 02 58 20 86 2a 7e 5e f1 47 f9 a5 f4 c5 12 e1 b6
62 3c d6 6c d1 7a 72 72 07 2b fe 5b 60 2f fe 30 7e e0 e9
```

The Responder signs using the private authentication key SK_R

Signature_or_MAC_2 (Raw Value) (64 bytes)

```
c3 b5 bd 44 d1 e4 4a 08 5c 03 d3 ae de 4e 1e 6c 11 c5 72 a1 96 8c c3
62 9b 50 5f 98 c6 81 60 8d 3d 1d e7 93 d1 c4 0e b5 dd 5d 89 ac f1 96
6a ea 07 02 2b 48 cd c9 98 70 eb c4 03 74 e8 fa 6e 09
```

Signature_or_MAC_2 (CBOR Data Item) (66 bytes)
58 40 c3 b5 bd 44 d1 e4 4a 08 5c 03 d3 ae de 4e 1e 6c 11 c5 72 a1 96
8c c3 62 9b 50 5f 98 c6 81 60 8d 3d 1d e7 93 d1 c4 0e b5 dd 5d 89 ac
f1 96 6a ea 07 02 2b 48 cd c9 98 70 eb c4 03 74 e8 fa 6e 09

The Responder constructs PLAINTEXT_2:

```
PLAINTEXT_2 =
(
 C_R,
 ID_CRED_R / bstr / -24..23,
 Signature_or_MAC_2,
 ? EAD_2
)
```

PLAINTEXT_2 (CBOR Sequence) (82 bytes)
41 18 a1 18 22 82 2e 48 79 f2 a4 1b 51 0c 1f 9b 58 40 c3 b5 bd 44 d1
e4 4a 08 5c 03 d3 ae de 4e 1e 6c 11 c5 72 a1 96 8c c3 62 9b 50 5f 98
c6 81 60 8d 3d 1d e7 93 d1 c4 0e b5 dd 5d 89 ac f1 96 6a ea 07 02 2b
48 cd c9 98 70 eb c4 03 74 e8 fa 6e 09

The input needed to calculate KEYSTREAM_2 is defined in Section 4.1.2 of [I-D.ietf-lake-edhoc], using EDHOC_Expand() with the EDHOC hash algorithm:

```
KEYSTREAM_2 = EDHOC_KDF( PRK_2e, 0, TH_2, plaintext_length ) =
            = HKDF-Expand( PRK_2e, info, plaintext_length )
```

where plaintext_length is the length in bytes of PLAINTEXT_2 in bytes, and info for KEYSTREAM_2 is:

```
info =
(
 0,
 h'c6405c154c567466ab1df20369500e540e9f14bd3a796a06
   52cae66c9061688d',
 82
)
```

where the last value is the length in bytes of PLAINTEXT_2.

info for KEYSTREAM_2 (CBOR Sequence) (37 bytes)
00 58 20 c6 40 5c 15 4c 56 74 66 ab 1d f2 03 69 50 0e 54 0e 9f 14 bd
3a 79 6a 06 52 ca e6 6c 90 61 68 8d 18 52

KEYSTREAM_2 (Raw Value) (82 bytes)
fd 3e 7c 3f 2d 6b ee 64 3d 3c 9d 2f 28 47 03 5d 73 e2 ec b0 f8 db 5c
d1 c6 85 4e 24 89 6a f2 11 88 b2 c4 34 4e 68 9e c2 98 42 83 d9 fb c6
9c e1 c5 db 10 dc ff f2 4d f9 a4 9a 04 a9 40 58 27 7b c7 fa 9a d6 c6
b1 94 ab 32 8b 44 5e b0 80 49 0c d7 86

The Responder calculates CIPHERTEXT_2 as XOR between PLAINTEXT_2 and KEYSTREAM_2:

CIPHERTEXT_2 (Raw Value) (82 bytes)
bc 26 dd 27 0f e9 c0 2c 44 ce 39 34 79 4b 1c c6 2b a2 2f 05 45 9f 8d
35 8c 8d 12 27 5a c4 2c 5f 96 de d5 f1 3c c9 08 4e 5b 20 18 89 a4 5e
5a 60 a5 56 2d c1 18 61 9c 3d aa 2f d9 f4 c9 f4 d6 ed ad 10 9d d4 ed
f9 59 62 aa fb af 9a b3 f4 a1 f6 b9 8f

The Responder constructs message_2:

```
message_2 =
(
 G_Y_CIPHERTEXT_2
)
```

where G_Y_CIPHERTEXT_2 is the bstr encoding of the concatenation of the raw values of G_Y and CIPHERTEXT_2.

message_2 (CBOR Sequence) (116 bytes)
58 72 dc 88 d2 d5 1d a5 ed 67 fc 46 16 35 6b c8 ca 74 ef 9e be 8b 38
7e 62 3a 36 0b a4 80 b9 b2 9d 1c bc 26 dd 27 0f e9 c0 2c 44 ce 39 34
79 4b 1c c6 2b a2 2f 05 45 9f 8d 35 8c 8d 12 27 5a c4 2c 5f 96 de d5
f1 3c c9 08 4e 5b 20 18 89 a4 5e 5a 60 a5 56 2d c1 18 61 9c 3d aa 2f
d9 f4 c9 f4 d6 ed ad 10 9d d4 ed f9 59 62 aa fb af 9a b3 f4 a1 f6 b9
8f

2.3. message_3

Since METHOD = 0, the Initiator authenticates using signatures. Since the selected cipher suite is 0, the EDHOC signature algorithm is EdDSA.

The Initiator’s signature key pair using EdDSA:

Initiator’s private authentication key
SK_I (Raw Value) (32 bytes)
4c 5b 25 87 8f 50 7c 6b 9d ae 68 fb d4 fd 3f f9 97 53 3d b0 af 00 b2
5d 32 4e a2 8e 6c 21 3b c8

Initiator's public authentication key
PK_I (Raw Value) (32 bytes)
ed 06 a8 ae 61 a8 29 ba 5f a5 45 25 c9 d0 7f 48 dd 44 a3 02 f4 3e 0f
23 d8 cc 20 b7 30 85 14 1e

PRK_4e3m is specified in Section 4.1.1.3 of [I-D.ietf-lake-edhoc].

Since the Initiator authenticates with signatures PRK_4e3m =
PRK_3e2m.

PRK_4e3m (Raw Value) (32 bytes)
d5 84 ac 2e 5d ad 5a 77 d1 4b 53 eb e7 2e f1 d5 da a8 86 0d 39 93 73
bf 2c 24 0a fa 7b a8 04 da

The transcript hash TH_3 is calculated using the EDHOC hash
algorithm:

TH_3 = H(TH_2, PLAINTEXT_2, CRED_R)

Input to calculate TH_3 (CBOR Sequence) (359 bytes)
58 20 c6 40 5c 15 4c 56 74 66 ab 1d f2 03 69 50 0e 54 0e 9f 14 bd 3a
79 6a 06 52 ca e6 6c 90 61 68 8d 41 18 a1 18 22 82 2e 48 79 f2 a4 1b
51 0c 1f 9b 58 40 c3 b5 bd 44 d1 e4 4a 08 5c 03 d3 ae de 4e 1e 6c 11
c5 72 a1 96 8c c3 62 9b 50 5f 98 c6 81 60 8d 3d 1d e7 93 d1 c4 0e b5
dd 5d 89 ac f1 96 6a ea 07 02 2b 48 cd c9 98 70 eb c4 03 74 e8 fa 6e
09 58 f1 30 81 ee 30 81 a1 a0 03 02 01 02 02 04 62 31 9e c4 30 05 06
03 2b 65 70 30 1d 31 1b 30 19 06 03 55 04 03 0c 12 45 44 48 4f 43 20
52 6f 6f 74 20 45 64 32 35 35 31 39 30 1e 17 0d 32 32 30 33 31 36 30
38 32 34 33 36 5a 17 0d 32 39 31 32 33 31 32 33 30 30 30 30 5a 30 22
31 20 30 1e 06 03 55 04 03 0c 17 45 44 48 4f 43 20 52 65 73 70 6f 6e
64 65 72 20 45 64 32 35 35 31 39 30 2a 30 05 06 03 2b 65 70 03 21 00
a1 db 47 b9 51 84 85 4a d1 2a 0c 1a 35 4e 41 8a ac e3 3a a0 f2 c6 62
c0 0b 3a c5 5d e9 2f 93 59 30 05 06 03 2b 65 70 03 41 00 b7 23 bc 01
ea b0 92 8e 8b 2b 6c 98 de 19 cc 38 23 d4 6e 7d 69 87 b0 32 47 8f ec
fa f1 45 37 a1 af 14 cc 8b e8 29 c6 b7 30 44 10 18 37 eb 4a bc 94 95
65 d8 6d ce 51 cf ae 52 ab 82 c1 52 cb 02

TH_3 (Raw Value) (32 bytes)
5b 7d f9 b4 f5 8f 24 0c e0 41 8e 48 19 1b 5f ff 3a 22 b5 ca 57 f6 69
b1 67 77 99 65 92 e9 28 bc

TH_3 (CBOR Data Item) (34 bytes)
58 20 5b 7d f9 b4 f5 8f 24 0c e0 41 8e 48 19 1b 5f ff 3a 22 b5 ca 57
f6 69 b1 67 77 99 65 92 e9 28 bc

The Initiator constructs the remaining input needed to calculate
MAC_3:

```
MAC_3 = EDHOC_KDF( PRK_4e3m, 6, context_3, mac_length_3 )
```

where

```
context_3 = << ID_CRED_I, TH_3, CRED_I, ? EAD_3 >>
```

CRED_I is identified by a 64-bit hash:

```
ID_CRED_I =
{
 34 : [-15, h'c24ab2fd7643c79f']
}
```

where the COSE header value 34 ('x5t') indicates a hash of an X.509 certficate, and the COSE algorithm -15 indicates the hash algorithm SHA-256 truncated to 64 bits.

ID_CRED_I (CBOR Data Item) (14 bytes)

```
a1 18 22 82 2e 48 c2 4a b2 fd 76 43 c7 9f
```

CRED_I is a CBOR byte string of the DER encoding of the X.509 certificate in Section 2.8.2:

CRED_I (Raw Value) (241 bytes)

```
30 81 ee 30 81 a1 a0 03 02 01 02 02 04 62 31 9e a0 30 05 06 03 2b 65
70 30 1d 31 1b 30 19 06 03 55 04 03 0c 12 45 44 48 4f 43 20 52 6f 6f
74 20 45 64 32 35 35 31 39 30 1e 17 0d 32 32 30 33 31 36 30 38 32 34
30 30 5a 17 0d 32 39 31 32 33 31 32 33 30 30 30 30 5a 30 22 31 20 30
1e 06 03 55 04 03 0c 17 45 44 48 4f 43 20 49 6e 69 74 69 61 74 6f 72
20 45 64 32 35 35 31 39 30 2a 30 05 06 03 2b 65 70 03 21 00 ed 06 a8
ae 61 a8 29 ba 5f a5 45 25 c9 d0 7f 48 dd 44 a3 02 f4 3e 0f 23 d8 cc
20 b7 30 85 14 1e 30 05 06 03 2b 65 70 03 41 00 52 12 41 d8 b3 a7 70
99 6b cf c9 b9 ea d4 e7 e0 a1 c0 db 35 3a 3b df 29 10 b3 92 75 ae 48
b7 56 01 59 81 85 0d 27 db 67 34 e3 7f 67 21 22 67 dd 05 ee ff 27 b9
e7 a8 13 fa 57 4b 72 a0 0b 43 0b
```

CRED_I (CBOR Data Item) (243 bytes)

```
58 f1 30 81 ee 30 81 a1 a0 03 02 01 02 02 04 62 31 9e a0 30 05 06 03
2b 65 70 30 1d 31 1b 30 19 06 03 55 04 03 0c 12 45 44 48 4f 43 20 52
6f 6f 74 20 45 64 32 35 35 31 39 30 1e 17 0d 32 32 30 33 31 36 30 38
32 34 30 30 5a 17 0d 32 39 31 32 33 31 32 33 30 30 30 30 5a 30 22 31
20 30 1e 06 03 55 04 03 0c 17 45 44 48 4f 43 20 49 6e 69 74 69 61 74
6f 72 20 45 64 32 35 35 31 39 30 2a 30 05 06 03 2b 65 70 03 21 00 ed
06 a8 ae 61 a8 29 ba 5f a5 45 25 c9 d0 7f 48 dd 44 a3 02 f4 3e 0f 23
d8 cc 20 b7 30 85 14 1e 30 05 06 03 2b 65 70 03 41 00 52 12 41 d8 b3
a7 70 99 6b cf c9 b9 ea d4 e7 e0 a1 c0 db 35 3a 3b df 29 10 b3 92 75
ae 48 b7 56 01 59 81 85 0d 27 db 67 34 e3 7f 67 21 22 67 dd 05 ee ff
27 b9 e7 a8 13 fa 57 4b 72 a0 0b 43 0b
```

No external authorization data:

EAD_3 (CBOR Sequence) (0 bytes)

context_3 = << ID_CRED_I, TH_3, CRED_I, ? EAD_3 >>

context_3 (CBOR Sequence) (291 bytes)
```
a1 18 22 82 2e 48 c2 4a b2 fd 76 43 c7 9f 58 20 5b 7d f9 b4 f5 8f 24
0c e0 41 8e 48 19 1b 5f ff 3a 22 b5 ca 57 f6 69 b1 67 77 99 65 92 e9
28 bc 58 f1 30 81 ee 30 81 a1 a0 03 02 01 02 02 04 62 31 9e a0 30 05
06 03 2b 65 70 30 1d 31 1b 30 19 06 03 55 04 03 0c 12 45 44 48 4f 43
20 52 6f 6f 74 20 45 64 32 35 35 31 39 30 1e 17 0d 32 32 30 33 31 36
30 38 32 34 30 30 5a 17 0d 32 39 31 32 33 31 32 33 30 30 30 30 5a 30
22 31 20 30 1e 06 03 55 04 03 0c 17 45 44 48 4f 43 20 49 6e 69 74 69
61 74 6f 72 20 45 64 32 35 35 31 39 30 2a 30 05 06 03 2b 65 70 03 21
00 ed 06 a8 ae 61 a8 29 ba 5f a5 45 25 c9 d0 7f 48 dd 44 a3 02 f4 3e
0f 23 d8 cc 20 b7 30 85 14 1e 30 05 06 03 2b 65 70 03 41 00 52 12 41
d8 b3 a7 70 99 6b cf c9 b9 ea d4 e7 e0 a1 c0 db 35 3a 3b df 29 10 b3
92 75 ae 48 b7 56 01 59 81 85 0d 27 db 67 34 e3 7f 67 21 22 67 dd 05
ee ff 27 b9 e7 a8 13 fa 57 4b 72 a0 0b 43 0b
```

context_3 (CBOR byte string) (294 bytes)
```
59 01 23 a1 18 22 82 2e 48 c2 4a b2 fd 76 43 c7 9f 58 20 5b 7d f9 b4
f5 8f 24 0c e0 41 8e 48 19 1b 5f ff 3a 22 b5 ca 57 f6 69 b1 67 77 99
65 92 e9 28 bc 58 f1 30 81 ee 30 81 a1 a0 03 02 01 02 02 04 62 31 9e
a0 30 05 06 03 2b 65 70 30 1d 31 1b 30 19 06 03 55 04 03 0c 12 45 44
48 4f 43 20 52 6f 6f 74 20 45 64 32 35 35 31 39 30 1e 17 0d 32 32 30
33 31 36 30 38 32 34 30 30 5a 17 0d 32 39 31 32 33 31 32 33 30 30 30
30 5a 30 22 31 20 30 1e 06 03 55 04 03 0c 17 45 44 48 4f 43 20 49 6e
69 74 69 61 74 6f 72 20 45 64 32 35 35 31 39 30 2a 30 05 06 03 2b 65
70 03 21 00 ed 06 a8 ae 61 a8 29 ba 5f a5 45 25 c9 d0 7f 48 dd 44 a3
02 f4 3e 0f 23 d8 cc 20 b7 30 85 14 1e 30 05 06 03 2b 65 70 03 41 00
52 12 41 d8 b3 a7 70 99 6b cf c9 b9 ea d4 e7 e0 a1 c0 db 35 3a 3b df
29 10 b3 92 75 ae 48 b7 56 01 59 81 85 0d 27 db 67 34 e3 7f 67 21 22
67 dd 05 ee ff 27 b9 e7 a8 13 fa 57 4b 72 a0 0b 43 0b
```

MAC_3 is computed through EDHOC_Expand() using the EDHOC hash algorithm, see Section 4.1.2 of [I-D.ietf-lake-edhoc]:

MAC_3 = HKDF-Expand(PRK_4e3m, info, mac_length_3), where

info = (6, context_3, mac_length_3)

where context_3 = << ID_CRED_I, TH_3, CRED_I, ? EAD_3 >>

Since METHOD = 0, mac_length_3 is given by the EDHOC hash algorithm.

info for MAC_3 is:

```
info =
(
 6,
 h'a11822822e48c24ab2fd7643c79f58205b7df9b4f58f240c
   e0418e48191b5fff3a22b5ca57f669b16777996592e928bc
   58f13081ee3081a1a003020102020462319ea0300506032b
   6570301d311b30190603550403 0c124544484f4320526f6f
   7420456432353531393 01e170d32323033331363038323430
   305a170d323933323331323333033030305a30223120301e06
   035504030c174544484f4320496e69746961746f72204564
   3235353139302a300506032b6570032100ed06a8ae61a829
   ba5fa54525c9d07f48dd44a302f43e0f23d8cc20b7308514
   1e300506032b657003410052124 1d8b3a770996bcfc9b9ea
   d4e7e0a1c0db353a3bdf2910b39275ae48b756015981850d
   27db6734e37f67212267dd05eeff27b9e7a813fa574b72a0
   0b430b',
 32
)
```

where the last value is the output size of the EDHOC hash algorithm in bytes.

info for MAC_3 (CBOR Sequence) (297 bytes)

```
06 59 01 23 a1 18 22 82 2e 48 c2 4a b2 fd 76 43 c7 9f 58 20 5b 7d f9
b4 f5 8f 24 0c e0 41 8e 48 19 1b 5f ff 3a 22 b5 ca 57 f6 69 b1 67 77
99 65 92 e9 28 bc 58 f1 30 81 ee 30 81 a1 a0 03 02 01 02 02 04 62 31
9e a0 30 05 06 03 2b 65 70 30 1d 31 1b 30 19 06 03 55 04 03 0c 12 45
44 48 4f 43 20 52 6f 6f 74 20 45 64 32 35 35 31 39 30 1e 17 0d 32 32
30 33 31 36 30 38 32 34 30 30 5a 17 0d 32 39 31 32 33 31 32 33 30 30
30 30 5a 30 22 31 20 30 1e 06 03 55 04 03 0c 17 45 44 48 4f 43 20 49
6e 69 74 69 61 74 6f 72 20 45 64 32 35 35 31 39 30 2a 30 05 06 03 2b
65 70 03 21 00 ed 06 a8 ae 61 a8 29 ba 5f a5 45 25 c9 d0 7f 48 dd 44
a3 02 f4 3e 0f 23 d8 cc 20 b7 30 85 14 1e 30 05 06 03 2b 65 70 03 41
00 52 12 41 d8 b3 a7 70 99 6b cf c9 b9 ea d4 e7 e0 a1 c0 db 35 3a 3b
df 29 10 b3 92 75 ae 48 b7 56 01 59 81 85 0d 27 db 67 34 e3 7f 67 21
22 67 dd 05 ee ff 27 b9 e7 a8 13 fa 57 4b 72 a0 0b 43 0b 18 20
```

MAC_3 (Raw Value) (32 bytes)

```
39 b1 27 c1 30 12 9a fa 30 61 8c 75 13 29 e6 37 cc 37 34 27 0d 4b 01
25 84 45 a8 ee 02 da a3 bd
```

MAC_3 (CBOR Data Item) (34 bytes)

```
58 20 39 b1 27 c1 30 12 9a fa 30 61 8c 75 13 29 e6 37 cc 37 34 27 0d 4b
01 25 84 45 a8 ee 02 da a3 bd
```

Since METHOD = 0, Signature_or_MAC_3 is the 'signature' of the COSE_Sign1 object.

The Initiator constructs the message to be signed:

```
[ "Signature1", << ID_CRED_I >>,
 << TH_3, CRED_I, ? EAD_3 >>, MAC_3 ] =

[
 "Signature1",
 h'a11822822e48c24ab2fd7643c79f',
 h'58205b7df9b4f58f240ce0418e48191b5fff3a22b5ca57f6
   69b16777996592e928bc58f13081ee3081a1a00302010202
   0462319ea0300506032b6570301d311b301906035504030c
   124544484f4320526f6f74204564323535313930 1e170d32
   323033313630383234303035a170d32393132333132333030
   30305a30223120301e06035504030c174544484f4320496e
   69746961746f722045643235353139302a300506032b6570
   032100ed06a8ae61a829ba5fa54525c9d07f48dd44a302f4
   3e0f23d8cc20b73085141e300506032b6570034100521241
   d8b3a770996bcfc9b9ead4e7e0a1c0db353a3bdf2910b392
   75ae48b756015981850d27db6734e37f67212267dd05eeff
   27b9e7a813fa574b72a00b430b',
 h'39b127c130129afa30618c751329e637cc3734270d4b0125
   8445a8ee02daa3bd'
]
```

Message to be signed 3 (CBOR Data Item) (341 bytes)

```
84 6a 53 69 67 6e 61 74 75 72 65 31 4e a1 18 22 82 2e 48 c2 4a b2 fd
76 43 c7 9f 59 01 15 58 20 5b 7d f9 b4 f5 8f 24 0c e0 41 8e 48 19 1b
5f ff 3a 22 b5 ca 57 f6 69 b1 67 77 99 65 92 e9 28 bc 58 f1 30 81 ee
30 81 a1 a0 03 02 01 02 02 04 62 31 9e a0 30 05 06 03 2b 65 70 30 1d
31 1b 30 19 06 03 55 04 03 0c 12 45 44 48 4f 43 20 52 6f 6f 74 20 45
64 32 35 35 31 39 30 1e 17 0d 32 32 30 33 31 36 30 38 32 34 30 30 5a
17 0d 32 39 31 32 33 31 32 33 30 30 30 30 5a 30 22 31 20 30 1e 06 03
55 04 03 0c 17 45 44 48 4f 43 20 49 6e 69 74 69 61 74 6f 72 20 45 64
32 35 35 31 39 30 2a 30 05 06 03 2b 65 70 03 21 00 ed 06 a8 ae 61 a8
29 ba 5f a5 45 25 c9 d0 7f 48 dd 44 a3 02 f4 3e 0f 23 d8 cc 20 b7 30
85 14 1e 30 05 06 03 2b 65 70 03 41 00 52 12 41 d8 b3 a7 70 99 6b cf
c9 b9 ea d4 e7 e0 a1 c0 db 35 3a 3b df 29 10 b3 92 75 ae 48 b7 56 01
59 81 85 0d 27 db 67 34 e3 7f 67 21 22 67 dd 05 ee ff 27 b9 e7 a8 13
fa 57 4b 72 a0 0b 43 0b 58 20 39 b1 27 c1 30 12 9a fa 30 61 8c 75 13
29 e6 37 cc 37 34 27 0d 4b 01 25 84 45 a8 ee 02 da a3 bd
```

The Initiator signs using the private authentication key SK_I:

Signature_or_MAC_3 (Raw Value) (64 bytes)

```
96 e1 cd 5f ce ad fa c1 b5 af 81 94 43 f7 09 24 f5 71 99 55 95 7f d0
26 55 be b4 77 5e 1a 73 18 6a 0d 1d 3e a6 83 f0 8f 8d 03 dc ec b9 cf
15 4e 1c 6f 55 5a 1e 12 ca 11 8c e4 2b db a6 87 89 07
```

Signature_or_MAC_3 (CBOR Data Item) (66 bytes)
58 40 96 e1 cd 5f ce ad fa c1 b5 af 81 94 43 f7 09 24 f5 71 99 55 95
7f d0 26 55 be b4 77 5e 1a 73 18 6a 0d 1d 3e a6 83 f0 8f 8d 03 dc ec
b9 cf 15 4e 1c 6f 55 5a 1e 12 ca 11 8c e4 2b db a6 87 89 07

The Initiator constructs PLAINTEXT_3:

```
PLAINTEXT_3 =
(
 ID_CRED_I / bstr / -24..23,
 Signature_or_MAC_3,
 ? EAD_3
)
```

PLAINTEXT_3 (CBOR Sequence) (80 bytes)
a1 18 22 82 2e 48 c2 4a b2 fd 76 43 c7 9f 58 40 96 e1 cd 5f ce ad fa
c1 b5 af 81 94 43 f7 09 24 f5 71 99 55 95 7f d0 26 55 be b4 77 5e 1a
73 18 6a 0d 1d 3e a6 83 f0 8f 8d 03 dc ec b9 cf 15 4e 1c 6f 55 5a 1e
12 ca 11 8c e4 2b db a6 87 89 07

The Initiator constructs the associated data for message_3:

```
A_3 =
[
 "Encrypt0",
 h'',
 h'5b7df9b4f58f240ce0418e48191b5fff3a22b5ca57f669b1
   6777996592e928bc'
]
```

A_3 (CBOR Data Item) (45 bytes)
83 68 45 6e 63 72 79 70 74 30 40 58 20 5b 7d f9 b4 f5 8f 24 0c e0 41
8e 48 19 1b 5f ff 3a 22 b5 ca 57 f6 69 b1 67 77 99 65 92 e9 28 bc

The Initiator constructs the input needed to derive the key K_3, see Section 4.1.2 of [I-D.ietf-lake-edhoc], using the EDHOC hash algorithm:

```
K_3 = EDHOC_KDF( PRK_3e2m, 3, TH_3, key_length )
    = HKDF-Expand( PRK_3e2m, info, key_length ),
```

where key_length is the key length in bytes for the EDHOC AEAD algorithm, and info for K_3 is:

```
info =
(
 3,
 h'5b7df9b4f58f240ce0418e48191b5fff3a22b5ca57f669b1
   6777996592e928bc',
 16
)
```

where the last value is the key length in bytes for the EDHOC AEAD algorithm.

info for K_3 (CBOR Sequence) (36 bytes)
03 58 20 5b 7d f9 b4 f5 8f 24 0c e0 41 8e 48 19 1b 5f ff 3a 22 b5 ca
57 f6 69 b1 67 77 99 65 92 e9 28 bc 10

K_3 (Raw Value) (16 bytes)
da 19 5e 5f 64 8a c6 3b 0e 8f b0 c4 55 20 51 39

The Initiator constructs the input needed to derive the nonce IV_3, see Section 4.1.2 of [I-D.ietf-lake-edhoc], using the EDHOC hash algorithm:

```
IV_3 = EDHOC_KDF( PRK_3e2m, 4, TH_3, iv_length )
     = HKDF-Expand( PRK_3e2m, info, iv_length ),
```

where iv_length is the nonce length in bytes for the EDHOC AEAD algorithm, and info for IV_3 is:

```
info =
(
 4,
 h'5b7df9b4f58f240ce0418e48191b5fff3a22b5ca57f669b1
   6777996592e928bc',
 13
)
```

where the last value is the nonce length in bytes for the EDHOC AEAD algorithm.

info for IV_3 (CBOR Sequence) (36 bytes)
04 58 20 5b 7d f9 b4 f5 8f 24 0c e0 41 8e 48 19 1b 5f ff 3a 22 b5 ca
57 f6 69 b1 67 77 99 65 92 e9 28 bc 0d

IV_3 (Raw Value) (13 bytes)
38 d8 c6 4c 56 25 5a ff a4 49 f4 be d7

The Initiator calculates CIPHERTEXT_3 as 'ciphertext' of COSE_Encrypt0 applied using the EDHOC AEAD algorithm with plaintext PLAINTEXT_3, additional data A_3, key K_3 and nonce IV_3.

CIPHERTEXT_3 (Raw Value) (88 bytes)
```
25 c3 45 88 4a aa eb 22 c5 27 f9 b1 d2 b6 78 72 07 e0 16 3c 69 b6 2a
0d 43 92 81 50 42 72 03 c3 16 74 e4 51 4e a6 e3 83 b5 66 eb 29 76 3e
fe b0 af a5 18 77 6a e1 c6 5f 85 6d 84 bf 32 af 3a 78 36 97 04 66 dc
b7 1f 76 74 5d 39 d3 02 5e 77 03 e0 c0 32 eb ad 51 94 7c
```

message_3 is the CBOR bstr encoding of CIPHERTEXT_3:

message_3 (CBOR Sequence) (90 bytes)
```
58 58 25 c3 45 88 4a aa eb 22 c5 27 f9 b1 d2 b6 78 72 07 e0 16 3c 69
b6 2a 0d 43 92 81 50 42 72 03 c3 16 74 e4 51 4e a6 e3 83 b5 66 eb 29
76 3e fe b0 af a5 18 77 6a e1 c6 5f 85 6d 84 bf 32 af 3a 78 36 97 04
66 dc b7 1f 76 74 5d 39 d3 02 5e 77 03 e0 c0 32 eb ad 51 94 7c
```

The transcript hash TH_4 is calculated using the EDHOC hash algorithm:

TH_4 = H(TH_3, PLAINTEXT_3, CRED_I)

Input to calculate TH_4 (CBOR Sequence) (357 bytes)
```
58 20 5b 7d f9 b4 f5 8f 24 0c e0 41 8e 48 19 1b 5f ff 3a 22 b5 ca 57
f6 69 b1 67 77 99 65 92 e9 28 bc a1 18 22 82 2e 48 c2 4a b2 fd 76 43
c7 9f 58 40 96 e1 cd 5f ce ad fa c1 b5 af 81 94 43 f7 09 24 f5 71 99
55 95 7f d0 26 55 be b4 77 5e 1a 73 18 6a 0d 1d 3e a6 83 f0 8f 8d 03
dc ec b9 cf 15 4e 1c 6f 55 5a 1e 12 ca 11 8c e4 2b db a6 87 89 07 58
f1 30 81 ee 30 81 a1 a0 03 02 01 02 02 04 62 31 9e a0 30 05 06 03 2b
65 70 30 1d 31 1b 30 19 06 03 55 04 03 0c 12 45 44 48 4f 43 20 52 6f
6f 74 20 45 64 32 35 35 31 39 30 1e 17 0d 32 32 30 33 31 36 30 38 32
34 30 30 5a 17 0d 32 39 31 32 33 31 32 33 30 30 30 30 5a 30 22 31 20
30 1e 06 03 55 04 03 0c 17 45 44 48 4f 43 20 49 6e 69 74 69 61 74 6f
72 20 45 64 32 35 35 31 39 30 2a 30 05 06 03 2b 65 70 03 21 00 ed 06
a8 ae 61 a8 29 ba 5f a5 45 25 c9 d0 7f 48 dd 44 a3 02 f4 3e 0f 23 d8
cc 20 b7 30 85 14 1e 30 05 06 03 2b 65 70 03 41 00 52 12 41 d8 b3 a7
70 99 6b cf c9 b9 ea d4 e7 e0 a1 c0 db 35 3a 3b df 29 10 b3 92 75 ae
48 b7 56 01 59 81 85 0d 27 db 67 34 e3 7f 67 21 22 67 dd 05 ee ff 27
b9 e7 a8 13 fa 57 4b 72 a0 0b 43 0b
```

TH_4 (Raw Value) (32 bytes)
```
0e b8 68 f2 63 cf 35 55 dc cd 39 6d d8 de c2 9d 37 50 d5 99 be 42 d5
a4 1a 5a 37 c8 96 f2 94 ac
```

TH_4 (CBOR Data Item) (34 bytes)
```
58 20 0e b8 68 f2 63 cf 35 55 dc cd 39 6d d8 de c2 9d 37 50 d5 99 be
42 d5 a4 1a 5a 37 c8 96 f2 94 ac
```

2.4. message_4

No external authorization data:

EAD_4 (CBOR Sequence) (0 bytes)

The Responder constructs PLAINTEXT_4:

```
PLAINTEXT_4 =
(
 ? EAD_4
)
```

PLAINTEXT_4 (CBOR Sequence) (0 bytes)

The Responder constructs the associated data for message_4:

```
A_4 =
[
 "Encrypt0",
 h'',
 h'0eb868f263cf3555dccd396dd8dec29d3750d599be42d5a4
   1a5a37c896f294ac'
]
```

A_4 (CBOR Data Item) (45 bytes)

```
83 68 45 6e 63 72 79 70 74 30 40 58 20 0e b8 68 f2 63 cf 35 55 dc cd
39 6d d8 de c2 9d 37 50 d5 99 be 42 d5 a4 1a 5a 37 c8 96 f2 94 ac
```

The Responder constructs the input needed to derive the EDHOC message_4 key, see Section 4.1.2 of [I-D.ietf-lake-edhoc], using the EDHOC hash algorithm:

```
K_4   = EDHOC_KDF( PRK_4e3m, 8, TH_4, key_length )
      = HKDF-Expand( PRK_4x3m, info, key_length )
```

where key_length is the key length in bytes for the EDHOC AEAD algorithm, and info for K_4 is:

```
info =
(
 8,
 h'0eb868f263cf3555dccd396dd8dec29d3750d599be42d5a4
   1a5a37c896f294ac',
 16
)
```

where the last value is the key length in bytes for the EDHOC AEAD algorithm.

info for K_4 (CBOR Sequence) (36 bytes)
08 58 20 0e b8 68 f2 63 cf 35 55 dc cd 39 6d d8 de c2 9d 37 50 d5 99 be 42 d5 a4 1a 5a 37 c8 96 f2 94 ac 10

K_4 (Raw Value) (16 bytes)
df 8c b5 86 1e 1f df ed d3 b2 30 15 a3 9d 1e 2e

The Responder constructs the input needed to derive the EDHOC message_4 nonce, see Section 4.1.2 of [I-D.ietf-lake-edhoc], using the EDHOC hash algorithm:

```
IV_4 = EDHOC_KDF( PRK_4e3m, 9, TH_4, iv_length )
     = HKDF-Expand( PRK_4x3m, info, iv_length )
```

where length is the nonce length in bytes for the EDHOC AEAD algorithm, and info for IV_4 is:

```
info =
(
 9,
 h'0eb868f263cf3555dccd396dd8dec29d3750d599be42d5a4
   1a5a37c896f294ac',
 13
)
```

where the last value is the nonce length in bytes for the EDHOC AEAD algorithm.

info for IV_4 (CBOR Sequence) (36 bytes)
09 58 20 0e b8 68 f2 63 cf 35 55 dc cd 39 6d d8 de c2 9d 37 50 d5 99 be 42 d5 a4 1a 5a 37 c8 96 f2 94 ac 0d

IV_4 (Raw Value) (13 bytes)
12 8e c6 58 d9 70 d7 38 0f 74 fc 6c 27

The Responder calculates CIPHERTEXT_4 as 'ciphertext' of COSE_Encrypt0 applied using the EDHOC AEAD algorithm with plaintext PLAINTEXT_4, additional data A_4, key K_4 and nonce IV_4.

CIPHERTEXT_4 (8 bytes)
4f 0e de e3 66 e5 c8 83

message_4 is the CBOR bstr encoding of CIPHERTEXT_4:

message_4 (CBOR Sequence) (9 bytes)
48 4f 0e de e3 66 e5 c8 83

2.5. PRK_out and PRK_exporter

PRK_out is specified in Section 4.1.3 of [I-D.ietf-lake-edhoc].

```
PRK_out = EDHOC_KDF( PRK_4e3m, 7, TH_4, hash_length ) =
        = HKDF-Expand( PRK_4e3m, info,  hash_length )
```

where hash_length is the length in bytes of the output of the EDHOC hash algorithm, and info for PRK_out is:

```
info =
(
 7,
 h'0eb868f263cf3555dccd396dd8dec29d3750d599be42d5a4
   1a5a37c896f294ac',
 32
)
```

where the last value is the length in bytes of the output of the EDHOC hash algorithm.

info for PRK_out (CBOR Sequence) (37 bytes)
07 58 20 0e b8 68 f2 63 cf 35 55 dc cd 39 6d d8 de c2 9d 37 50 d5 99 be 42 d5 a4 1a 5a 37 c8 96 f2 94 ac 18 20

PRK_out (Raw Value) (32 bytes)
b7 44 cb 7d 8a 87 cc 04 47 c3 35 0e 16 5b 25 0d ab 12 ec 45 33 25 ab b9 22 b3 03 07 e5 c3 68 f0

The OSCORE Master Secret and OSCORE Master Salt are derived with the EDHOC_Exporter as specified in Section 4.2.1 of [I-D.ietf-lake-edhoc].

```
EDHOC_Exporter( label, context, length )
= EDHOC_KDF( PRK_exporter, label, context, length )
```

where PRK_exporter is derived from PRK_out:

```
 PRK_exporter = EDHOC_KDF( PRK_out, 10, h'', hash_length ) =
              = HKDF-Expand( PRK_out, info,  hash_length )
```

where hash_length is the length in bytes of the output of the EDHOC hash algorithm, and info for the PRK_exporter is:

```
info =
(
 10,
 h'',
 32
)
```

where the last value is the length in bytes of the output of the EDHOC hash algorithm.

info for PRK_exporter (CBOR Sequence) (4 bytes)
0a 40 18 20

PRK_exporter (Raw Value) (32 bytes)
2a ae c8 fc 4a b3 bc 32 95 de f6 b5 51 05 1a 2f a5 61 42 4d b3 01 fa 84 f6 42 f5 57 8a 6d f5 1a

2.6. OSCORE Parameters

The derivation of OSCORE parameters is specified in Appendix A.1 of [I-D.ietf-lake-edhoc].

The AEAD and Hash algorithms to use in OSCORE are given by the selected cipher suite:

Application AEAD Algorithm (int)
10

Application Hash Algorithm (int)
-16

The mapping from EDHOC connection identifiers to OSCORE Sender/ Recipient IDs is defined in Section 3.3.3 of [I-D.ietf-lake-edhoc].

C_R is mapped to the Recipient ID of the server, i.e., the Sender ID of the client. The byte string 0x18, which as C_R is encoded as the CBOR byte string 0x4118, is converted to the server Recipient ID 0x18.

Client's OSCORE Sender ID (Raw Value) (1 byte)
18

C_I is mapped to the Recipient ID of the client, i.e., the Sender ID of the server. The byte string 0x2d, which as C_I is encoded as the CBOR integer 0x2d is converted to the client Recipient ID 0x2d.

Server's OSCORE Sender ID (Raw Value) (1 byte)
2d

The OSCORE Master Secret is computed through EDHOC_Expand() using the Application hash algorithm, see Appendix A.1 of [I-D.ietf-lake-edhoc]:

```
OSCORE Master Secret = EDHOC_Exporter( 0, h'', oscore_key_length )
= EDHOC_KDF( PRK_exporter, 0, h'',  oscore_key_length )
= HKDF-Expand( PRK_exporter, info,  oscore_key_length )
```

where oscore_key_length is by default the key length in bytes for the Application AEAD algorithm, and info for the OSCORE Master Secret is:

```
info =
(
 0,
 h'',
 16
)
```

where the last value is the key length in bytes for the Application AEAD algorithm.

```
info for OSCORE Master Secret (CBOR Sequence) (3 bytes)
00 40 10
```

```
OSCORE Master Secret (Raw Value) (16 bytes)
1e 1c 6b ea c3 a8 a1 ca c4 35 de 7e 2f 9a e7 ff
```

The OSCORE Master Salt is computed through EDHOC_Expand() using the Application hash algorithm, see Section 4.2 of [I-D.ietf-lake-edhoc]:

```
OSCORE Master Salt = EDHOC_Exporter( 1, h'', oscore_salt_length )
= EDHOC_KDF( PRK_exporter, 1, h'', oscore_salt_length )
= HKDF-Expand( PRK_4x3m, info, oscore_salt_length )
```

where oscore_salt_length is the length in bytes of the OSCORE Master Salt, and info for the OSCORE Master Salt is:

```
info =
(
 1,
 h'',
 8
)
```

where the last value is the length in bytes of the OSCORE Master Salt.

info for OSCORE Master Salt (CBOR Sequence) (3 bytes)
01 40 08

OSCORE Master Salt (Raw Value) (8 bytes)
ce 7a b8 44 c0 10 6d 73

2.7. Key Update

Key update is defined in Appendix H of [I-D.ietf-lake-edhoc].

```
EDHOC_KeyUpdate( context ):
PRK_out = EDHOC_KDF( PRK_out, 11, context, hash_length )
        = HKDF-Expand( PRK_out, info, hash_length )
```

where hash_length is the length in bytes of the output of the EDHOC hash function, and context for KeyUpdate is

context for KeyUpdate (Raw Value) (16 bytes)
d6 be 16 96 02 b8 bc ea a0 11 58 fd b8 20 89 0c

context for KeyUpdate (CBOR Data Item) (17 bytes)
50 d6 be 16 96 02 b8 bc ea a0 11 58 fd b8 20 89 0c

and where info for key update is:

```
info =
(
 11,
 h'd6be169602b8bceaa01158fdb820890c',
 32
)
```

PRK_out after KeyUpdate (Raw Value) (32 bytes)
da 6e ac d9 a9 85 f4 fb a9 ae c2 a9 29 90 22 97 6b 25 b1 4e 89 fa 15
97 94 f2 8d 82 fa f2 da ad

After key update, the PRK_exporter needs to be derived anew:

```
 PRK_exporter = EDHOC_KDF( PRK_out, 10, h'', hash_length ) =
              = HKDF-Expand( PRK_out, info,  hash_length )
```

where info and hash_length are unchanged as in Section 2.5.

PRK_exporter after KeyUpdate (Raw Value) (32 bytes)
00 14 d2 52 5e e0 d8 e2 13 ea 59 08 02 8e 9a 1c e9 a0 1c 30 54 6f 09
30 c0 44 d3 8d b5 36 2c 05

The OSCORE Master Secret is derived with the updated PRK_exporter:

OSCORE Master Secret =
= HKDF-Expand(PRK_exporter, info, oscore_key_length)

where info and key_length are unchanged as in Section 2.6.

OSCORE Master Secret after KeyUpdate (Raw Value) (16 bytes)
ee 0f f5 42 c4 7e b0 e0 9c 69 30 76 49 bd bb e5

The OSCORE Master Salt is derived with the updated PRK_exporter:

OSCORE Master Salt = HKDF-Expand(PRK_exporter, info, salt_length)

where info and salt_length are unchanged as in Section 2.6.

OSCORE Master Salt after KeyUpdate (Raw Value) (8 bytes)
80 ce de 2a 1e 5a ab 48

2.8. Certificates

2.8.1. Responder Certificate

```
Version: 3 (0x2)
Serial Number: 1647419076 (0x62319ec4)
Signature Algorithm: ED25519
Issuer: CN = EDHOC Root Ed25519
Validity
    Not Before: Mar 16 08:24:36 2022 GMT
    Not After : Dec 31 23:00:00 2029 GMT
Subject: CN = EDHOC Responder Ed25519
Subject Public Key Info:
    Public Key Algorithm: ED25519
        ED25519 Public-Key:
        pub:
            a1 db 47 b9 51 84 85 4a d1 2a 0c 1a 35 4e 41
            8a ac e3 3a a0 f2 c6 62 c0 0b 3a c5 5d e9 2f
            93 59
Signature Algorithm: ED25519
Signature Value:
    b7 23 bc 01 ea b0 92 8e 8b 2b 6c 98 de 19 cc 38 23 d4
    6e 7d 69 87 b0 32 47 8f ec fa f1 45 37 a1 af 14 cc 8b
    e8 29 c6 b7 30 44 10 18 37 eb 4a bc 94 95 65 d8 6d ce
    51 cf ae 52 ab 82 c1 52 cb 02
```

2.8.2. Initiator Certificate

```
Version: 3 (0x2)
Serial Number: 1647419040 (0x62319ea0)
Signature Algorithm: ED25519
Issuer: CN = EDHOC Root Ed25519
Validity
    Not Before: Mar 16 08:24:00 2022 GMT
    Not After : Dec 31 23:00:00 2029 GMT
Subject: CN = EDHOC Initiator Ed25519
Subject Public Key Info:
    Public Key Algorithm: ED25519
        ED25519 Public-Key:
        pub:
            ed 06 a8 ae 61 a8 29 ba 5f a5 45 25 c9 d0 7f
            48 dd 44 a3 02 f4 3e 0f 23 d8 cc 20 b7 30 85
            14 1e
Signature Algorithm: ED25519
Signature Value:
    52 12 41 d8 b3 a7 70 99 6b cf c9 b9 ea d4 e7 e0 a1 c0
    db 35 3a 3b df 29 10 b3 92 75 ae 48 b7 56 01 59 81 85
    0d 27 db 67 34 e3 7f 67 21 22 67 dd 05 ee ff 27 b9 e7
    a8 13 fa 57 4b 72 a0 0b 43 0b
```

2.8.3. Common Root Certificate

```
Version: 3 (0x2)
Serial Number: 1647418996 (0x62319e74)
Signature Algorithm: ED25519
Issuer: CN = EDHOC Root Ed25519
Validity
    Not Before: Mar 16 08:23:16 2022 GMT
    Not After : Dec 31 23:00:00 2029 GMT
Subject: CN = EDHOC Root Ed25519
Subject Public Key Info:
    Public Key Algorithm: ED25519
        ED25519 Public-Key:
        pub:
            2b 7b 3e 80 57 c8 64 29 44 d0 6a fe 7a 71 d1
            c9 bf 96 1b 62 92 ba c4 b0 4f 91 66 9b bb 71
            3b e4
X509v3 extensions:
    X509v3 Key Usage: critical
        Certificate Sign
    X509v3 Basic Constraints: critical
        CA:TRUE
Signature Algorithm: ED25519
Signature Value:
    4b b5 2b bf 15 39 b7 1a 4a af 42 97 78 f2 9e da 7e 81
    46 80 69 8f 16 c4 8f 2a 6f a4 db e8 25 41 c5 82 07 ba
    1b c9 cd b0 c2 fa 94 7f fb f0 f0 ec 0e e9 1a 7f f3 7a
    94 d9 25 1f a5 cd f1 e6 7a 0f
```

3. Authentication with Static DH, CCS Identified by 'kid'

In this example the Initiator and the Responder are authenticated with ephemeral-static Diffie-Hellman (METHOD = 3). The Initiator supports cipher suites 6 and 2 (in order of preference) and the Responder only supports cipher suite 2. After an initial negotiation message exchange, cipher suite 2 is used, which determines the algorithms:

* EDHOC AEAD algorithm = AES-CCM-16-64-128

* EDHOC hash algorithm = SHA-256

* EDHOC MAC length in bytes (Static DH) = 8

* EDHOC key exchange algorithm (ECDH curve) = P-256

* EDHOC signature algorithm = ES256

* Application AEAD algorithm = AES-CCM-16-64-128

* Application hash algorithm = SHA-256

The public keys are represented as raw public keys (RPK), encoded in a CWT Claims Set (CCS) and identified by the COSE header parameter 'kid'.

3.1. message_1 (first time)

Both endpoints are authenticated with static DH, i.e., METHOD = 3:

METHOD (CBOR Data Item) (1 byte)
03

The Initiator selects its preferred cipher suite 6. A single cipher suite is encoded as an int:

SUITES_I (CBOR Data Item) (1 byte)
06

The Initiator creates an ephemeral key pair for use with the EDHOC key exchange algorithm:

Initiator's ephemeral private key
X (Raw Value) (32 bytes)
5c 41 72 ac a8 b8 2b 5a 62 e6 6f 72 22 16 f5 a1 0f 72 aa 69 f4 2c 1d 1c d3 cc d7 bf d2 9c a4 e9

Initiator's ephemeral public key, 'x'-coordinate
G_X (Raw Value) (32 bytes)
74 1a 13 d7 ba 04 8f bb 61 5e 94 38 6a a3 b6 1b ea 5b 3d 8f 65 f3 26 20 b7 49 be e8 d2 78 ef a9

Initiator's ephemeral public key, 'x'-coordinate
G_X (CBOR Data Item) (34 bytes)
58 20 74 1a 13 d7 ba 04 8f bb 61 5e 94 38 6a a3 b6 1b ea 5b 3d 8f 65 f3 26 20 b7 49 be e8 d2 78 ef a9

The Initiator selects its connection identifier C_I to be the byte string 0x0e, which since it is represented by the 1-byte CBOR int 14 is encoded as 0x0e:

Connection identifier chosen by Initiator
C_I (Raw Value) (1 byte)
0e

Connection identifier chosen by Initiator
C_I (CBOR Data Item) (1 byte)
0e

No external authorization data:

EAD_1 (CBOR Sequence) (0 bytes)

The Initiator constructs message_1:

```
message_1 =
(
 3,
 6,
 h'741a13d7ba048fbb615e94386aa3b61bea5b3d8f65f32620
   b749bee8d278efa9',
 14
)
```

message_1 (CBOR Sequence) (37 bytes)
03 06 58 20 74 1a 13 d7 ba 04 8f bb 61 5e 94 38 6a a3 b6 1b ea 5b 3d
8f 65 f3 26 20 b7 49 be e8 d2 78 ef a9 0e

3.2. error

The Responder does not support cipher suite 6 and sends an error with ERR_CODE 2 containing SUITES_R as ERR_INFO. The Responder proposes cipher suite 2, a single cipher suite thus encoded as an int.

SUITES_R
02

error (CBOR Sequence) (2 bytes)
02 02

3.3. message_1 (second time)

Same steps are performed as for message_1 the first time, Section 3.1, but with updated SUITES_I.

Both endpoints are authenticated with static DH, i.e., METHOD = 3:

METHOD (CBOR Data Item) (1 byte)
03

The Initiator selects cipher suite 2 and indicates the more preferred cipher suite(s), in this case 6, all encoded as the array [6, 2]:

SUITES_I (CBOR Data Item) (3 bytes)
82 06 02

The Initiator creates an ephemeral key pair for use with the EDHOC key exchange algorithm:

Initiator's ephemeral private key
X (Raw Value) (32 bytes)
36 8e c1 f6 9a eb 65 9b a3 7d 5a 8d 45 b2 1b dc 02 99 dc ea a8 ef 23
5f 3c a4 2c e3 53 0f 95 25

Initiator's ephemeral public key, 'x'-coordinate
G_X (Raw Value) (32 bytes)
8a f6 f4 30 eb e1 8d 34 18 40 17 a9 a1 1b f5 11 c8 df f8 f8 34 73 0b
96 c1 b7 c8 db ca 2f c3 b6

Initiator's ephemeral public key, one 'y'-coordinate
(Raw Value) (32 bytes)
51 e8 af 6c 6e db 78 16 01 ad 1d 9c 5f a8 bf 7a a1 57 16 c7 c0 6a 5d
03 85 03 c6 14 ff 80 c9 b3

Initiator's ephemeral public key, 'x'-coordinate
G_X (CBOR Data Item) (34 bytes)
58 20 8a f6 f4 30 eb e1 8d 34 18 40 17 a9 a1 1b f5 11 c8 df f8 f8 34
73 0b 96 c1 b7 c8 db ca 2f c3 b6

The Initiator selects its connection identifier C_I to be the byte string 0x37, which since it is represented by the 1-byte CBOR int -24 is encoded as 0x37:

Connection identifier chosen by Initiator
C_I (Raw Value) (1 byte)
37

Connection identifier chosen by Initiator
C_I (CBOR Data Item) (1 byte)
37

No external authorization data:

EAD_1 (CBOR Sequence) (0 bytes)

The Initiator constructs message_1:

```
message_1 =
(
 3,
 [6, 2],
 h'8af6f430ebe18d34184017a9a11bf511c8dff8f834730b96
   c1b7c8dbca2fc3b6',
 -24
)
```

```
message_1 (CBOR Sequence) (39 bytes)
03 82 06 02 58 20 8a f6 f4 30 eb e1 8d 34 18 40 17 a9 a1 1b f5 11 c8
df f8 f8 34 73 0b 96 c1 b7 c8 db ca 2f c3 b6 37
```

3.4. message_2

The Responder supports the selected cipher suite 2 and not the by the Initiator more preferred cipher suite(s) 6, so SUITES_I is acceptable.

The Responder creates an ephemeral key pair for use with the EDHOC key exchange algorithm:

```
Responder's ephemeral private key
Y (Raw Value) (32 bytes)
e2 f4 12 67 77 20 5e 85 3b 43 7d 6e ac a1 e1 f7 53 cd cc 3e 2c 69 fa
88 4b 0a 1a 64 09 77 e4 18
```

```
Responder's ephemeral public key, 'x'-coordinate
G_Y (Raw Value) (32 bytes)
41 97 01 d7 f0 0a 26 c2 dc 58 7a 36 dd 75 25 49 f3 37 63 c8 93 42 2c
8e a0 f9 55 a1 3a 4f f5 d5
```

```
Responder's ephemeral public key, one 'y'-coordinate
(Raw Value) (32 bytes)
5e 4f 0d d8 a3 da 0b aa 16 b9 d3 ad 56 a0 c1 86 0a 94 0a f8 59 14 91
5e 25 01 9b 40 24 17 e9 9d
```

```
Responder's ephemeral public key, 'x'-coordinate
G_Y (CBOR Data Item) (34 bytes)
58 20 41 97 01 d7 f0 0a 26 c2 dc 58 7a 36 dd 75 25 49 f3 37 63 c8 93
42 2c 8e a0 f9 55 a1 3a 4f f5 d5
```

The Responder selects its connection identifier C_R to be the byte string 0x27, which since it is represented by the 1-byte CBOR int -8 is encoded as 0x27:

Connection identifier chosen by Responder
C_R (raw value) (1 byte)
27

Connection identifier chosen by Responder
C_R (CBOR Data Item) (1 byte)
27

The transcript hash TH_2 is calculated using the EDHOC hash algorithm:

TH_2 = H(G_Y, H(message_1))

H(message_1) (Raw Value) (32 bytes)
ca 02 ca bd a5 a8 90 27 49 b4 2f 71 10 50 bb 4d bd 52 15 3e 87 52 75
94 b3 9f 50 cd f0 19 88 8c

H(message_1) (CBOR Data Item) (34 bytes)
58 20 ca 02 ca bd a5 a8 90 27 49 b4 2f 71 10 50 bb 4d bd 52 15 3e 87
52 75 94 b3 9f 50 cd f0 19 88 8c

The input to calculate TH_2 is the CBOR sequence:

G_Y, H(message_1)

Input to calculate TH_2 (CBOR Sequence) (68 bytes)
58 20 41 97 01 d7 f0 0a 26 c2 dc 58 7a 36 dd 75 25 49 f3 37 63 c8 93
42 2c 8e a0 f9 55 a1 3a 4f f5 d5 58 20 ca 02 ca bd a5 a8 90 27 49 b4
2f 71 10 50 bb 4d bd 52 15 3e 87 52 75 94 b3 9f 50 cd f0 19 88 8c

TH_2 (Raw Value) (32 bytes)
35 6e fd 53 77 14 25 e0 08 f3 fe 3a 86 c8 3f f4 c6 b1 6e 57 02 8f f3
9d 52 36 c1 82 b2 02 08 4b

TH_2 (CBOR Data Item) (34 bytes)
58 20 35 6e fd 53 77 14 25 e0 08 f3 fe 3a 86 c8 3f f4 c6 b1 6e 57 02
8f f3 9d 52 36 c1 82 b2 02 08 4b

PRK_2e is specified in Section 4.1.1.1 of [I-D.ietf-lake-edhoc].

First, the ECDH shared secret G_XY is computed from G_X and Y, or G_Y and X:

G_XY (Raw Value) (ECDH shared secret) (32 bytes)
2f 0c b7 e8 60 ba 53 8f bf 5c 8b de d0 09 f6 25 9b 4b 62 8f e1 eb 7d
be 93 78 e5 ec f7 a8 24 ba

Then, PRK_2e is calculated using EDHOC_Extract() determined by the EDHOC hash algorithm:

```
PRK_2e = EDHOC_Extract( salt, G_XY ) =
       = HMAC-SHA-256( salt, G_XY )
```

where salt is TH_2:

```
salt (Raw Value) (32 bytes)
35 6e fd 53 77 14 25 e0 08 f3 fe 3a 86 c8 3f f4 c6 b1 6e 57 02 8f f3
9d 52 36 c1 82 b2 02 08 4b
```

```
PRK_2e (Raw Value) (32 bytes)
5a a0 d6 9f 3e 3d 1e 0c 47 9f 0b 8a 48 66 90 c9 80 26 30 c3 46 6b 1d
c9 23 71 c9 82 56 31 70 b5
```

Since METHOD = 3, the Responder authenticates using static DH. The EDHOC key exchange algorithm is based on the same curve as for the ephemeral keys, which is P-256, since the selected cipher suite is 2.

The Responder's static Diffie-Hellman P-256 key pair:

```
Responder's private authentication key
SK_R (Raw Value) (32 bytes)
72 cc 47 61 db d4 c7 8f 75 89 31 aa 58 9d 34 8d 1e f8 74 a7 e3 03 ed
e2 f1 40 dc f3 e6 aa 4a ac
```

```
Responder's public authentication key, 'x'-coordinate
(Raw Value) (32 bytes)
bb c3 49 60 52 6e a4 d3 2e 94 0c ad 2a 23 41 48 dd c2 17 91 a1 2a fb
cb ac 93 62 20 46 dd 44 f0
```

```
Responder's public authentication key, 'y'-coordinate
(Raw Value) (32 bytes)
45 19 e2 57 23 6b 2a 0c e2 02 3f 09 31 f1 f3 86 ca 7a fd a6 4f cd e0
10 8c 22 4c 51 ea bf 60 72
```

Since the Responder authenticates with static DH (METHOD = 3), PRK_3e2m is derived from SALT_3e2m and G_RX.

The input needed to calculate SALT_3e2m is defined in Section 4.1.2 of [I-D.ietf-lake-edhoc], using EDHOC_Expand() with the EDHOC hash algorithm:

```
SALT_3e2m  = EDHOC_KDF( PRK_2e, 1, TH_2, hash_length ) =
           = HKDF-Expand( PRK_2e, info, hash_length )
```

where hash_length is the length in bytes of the output of the EDHOC hash algorithm, and info for SALT_3e2m is:

```
info =
(
 1,
 h'356efd53771425e008f3fe3a86c83ff4c6b16e57028ff39d
   5236c182b202084b',
 32
)
```

info for SALT_3e2m (CBOR Sequence) (37 bytes)
01 58 20 35 6e fd 53 77 14 25 e0 08 f3 fe 3a 86 c8 3f f4 c6 b1 6e 57
02 8f f3 9d 52 36 c1 82 b2 02 08 4b 18 20

SALT_3e2m (Raw Value) (32 bytes)
af 4e 10 3a 47 cb 3c f3 25 70 d5 c2 5a d2 77 32 bd 8d 81 78 e9 a6 9d
06 1c 31 a2 7f 8e 3c a9 26

PRK_3e2m is specified in Section 4.1.1.2 of [I-D.ietf-lake-edhoc].

PRK_3e2m is derived from G_RX using EDHOC_Extract() with the EDHOC hash algorithm:

```
PRK_3e2m = EDHOC_Extract( SALT_3e2m, G_RX ) =
         = HMAC-SHA-256( SALT_3e2m, G_RX )
```

where G_RX is the ECDH shared secret calculated from G_X and R, or G_R and X.

G_RX (Raw Value) (ECDH shared secret) (32 bytes)
f2 b6 ee a0 22 20 b9 5e ee 5a 0b c7 01 f0 74 e0 0a 84 3e a0 24 22 f6
08 25 fb 26 9b 3e 16 14 23

PRK_3e2m (Raw Value) (32 bytes)
0c a3 d3 39 82 96 b3 c0 39 00 98 76 20 c1 1f 6f ce 70 78 1c 1d 12 19
72 0f 9e c0 8c 12 2d 84 34

The Responder constructs the remaining input needed to calculate MAC_2:

```
MAC_2 = EDHOC_KDF( PRK_3e2m, 2, context_2, mac_length_2 )

context_2 = << C_R, ID_CRED_R, TH_2, CRED_R, ? EAD_2 >>
```

CRED_R is identified by a 'kid' with byte string value 0x32:

```
ID_CRED_R =
{
 4 : h'32'
}

ID_CRED_R (CBOR Data Item) (4 bytes)
a1 04 41 32

CRED_R is an RPK encoded as a CCS:

{                                                  /CCS/
  2 : "example.edu",                               /sub/
  8 : {                                            /cnf/
    1 : {                                          /COSE_Key/
      1 : 2,                                       /kty/
      2 : h'32',                                   /kid/
     -1 : 1,                                       /crv/
     -2 : h'BBC34960526EA4D32E940CAD2A234148
             DDC21791A12AFBCBAC93622046DD44F0', /x/
     -3 : h'4519E257236B2A0CE2023F0931F1F386
             CA7AFDA64FCDE0108C224C51EABF6072'  /y/
    }
  }
}

CRED_R (CBOR Data Item) (95 bytes)
a2 02 6b 65 78 61 6d 70 6c 65 2e 65 64 75 08 a1 01 a5 01 02 02 41 32
20 01 21 58 20 bb c3 49 60 52 6e a4 d3 2e 94 0c ad 2a 23 41 48 dd c2
17 91 a1 2a fb cb ac 93 62 20 46 dd 44 f0 22 58 20 45 19 e2 57 23 6b
2a 0c e2 02 3f 09 31 f1 f3 86 ca 7a fd a6 4f cd e0 10 8c 22 4c 51 ea
bf 60 72

No external authorization data:

EAD_2 (CBOR Sequence) (0 bytes)

context_2 = << C_R, ID_CRED_R, TH_2, CRED_R, ? EAD_2 >>

context_2 (CBOR Sequence) (134 bytes)
27 a1 04 41 32 58 20 35 6e fd 53 77 14 25 e0 08 f3 fe 3a 86 c8 3f f4
c6 b1 6e 57 02 8f f3 9d 52 36 c1 82 b2 02 08 4b a2 02 6b 65 78 61 6d
70 6c 65 2e 65 64 75 08 a1 01 a5 01 02 02 41 32 20 01 21 58 20 bb c3
49 60 52 6e a4 d3 2e 94 0c ad 2a 23 41 48 dd c2 17 91 a1 2a fb cb ac
93 62 20 46 dd 44 f0 22 58 20 45 19 e2 57 23 6b 2a 0c e2 02 3f 09 31
f1 f3 86 ca 7a fd a6 4f cd e0 10 8c 22 4c 51 ea bf 60 72
```

context_2 (CBOR byte string) (136 bytes)
58 86 27 a1 04 41 32 58 20 35 6e fd 53 77 14 25 e0 08 f3 fe 3a 86 c8
3f f4 c6 b1 6e 57 02 8f f3 9d 52 36 c1 82 b2 02 08 4b a2 02 6b 65 78
61 6d 70 6c 65 2e 65 64 75 08 a1 01 a5 01 02 02 41 32 20 01 21 58 20
bb c3 49 60 52 6e a4 d3 2e 94 0c ad 2a 23 41 48 dd c2 17 91 a1 2a fb
cb ac 93 62 20 46 dd 44 f0 22 58 20 45 19 e2 57 23 6b 2a 0c e2 02 3f
09 31 f1 f3 86 ca 7a fd a6 4f cd e0 10 8c 22 4c 51 ea bf 60 72

MAC_2 is computed through EDHOC_Expand() using the EDHOC hash algorithm, see Section 4.1.2 of [I-D.ietf-lake-edhoc]:

MAC_2 = HKDF-Expand(PRK_3e2m, info, mac_length_2), where

info = (2, context_2, mac_length_2)

Since METHOD = 3, mac_length_2 is given by the EDHOC MAC length.

info for MAC_2 is:

```
info =
(
 2,
 h'27a10441325820356efd53771425e008f3fe3a86c83ff4c6
   b16e57028ff39d5236c182b202084ba2026b6578616d706c
   652e65647508a101a501020241322001215820bbc3496052
   6ea4d32e940cad2a234148ddc21791a12afbcbac93622046
   dd44f02258204519e257236b2a0ce2023f0931f1f386ca7a
   fda64fcde0108c224c51eabf6072',
 8
)
```

where the last value is the EDHOC MAC length in bytes.

info for MAC_2 (CBOR Sequence) (138 bytes)
02 58 86 27 a1 04 41 32 58 20 35 6e fd 53 77 14 25 e0 08 f3 fe 3a 86
c8 3f f4 c6 b1 6e 57 02 8f f3 9d 52 36 c1 82 b2 02 08 4b a2 02 6b 65
78 61 6d 70 6c 65 2e 65 64 75 08 a1 01 a5 01 02 02 41 32 20 01 21 58
20 bb c3 49 60 52 6e a4 d3 2e 94 0c ad 2a 23 41 48 dd c2 17 91 a1 2a
fb cb ac 93 62 20 46 dd 44 f0 22 58 20 45 19 e2 57 23 6b 2a 0c e2 02
3f 09 31 f1 f3 86 ca 7a fd a6 4f cd e0 10 8c 22 4c 51 ea bf 60 72 08

MAC_2 (Raw Value) (8 bytes)
09 43 30 5c 89 9f 5c 54

MAC_2 (CBOR Data Item) (9 bytes)
48 09 43 30 5c 89 9f 5c 54

Since METHOD = 3, Signature_or_MAC_2 is MAC_2:

Signature_or_MAC_2 (Raw Value) (8 bytes)
09 43 30 5c 89 9f 5c 54

Signature_or_MAC_2 (CBOR Data Item) (9 bytes)
48 09 43 30 5c 89 9f 5c 54

The Responder constructs PLAINTEXT_2:

```
PLAINTEXT_2 =
(
 C_R,
 ID_CRED_R / bstr / -24..23,
 Signature_or_MAC_2,
 ? EAD_2
)
```

Since ID_CRED_R contains a single 'kid' parameter, only the byte string value is included in the plaintext, represented as described in Section 3.3.2 of [I-D.ietf-lake-edhoc]. The CBOR map { 4 : h'32' } is thus replaced, not by the CBOR byte string 0x4132, but by the CBOR int 0x32, since that is a one byte encoding of a CBOR integer (-19).

PLAINTEXT_2 (CBOR Sequence) (11 bytes)
27 32 48 09 43 30 5c 89 9f 5c 54

The input needed to calculate KEYSTREAM_2 is defined in Section 4.1.2 of [I-D.ietf-lake-edhoc], using EDHOC_Expand() with the EDHOC hash algorithm:

```
KEYSTREAM_2 = EDHOC_KDF( PRK_2e, 0, TH_2, plaintext_length ) =
            = HKDF-Expand( PRK_2e, info, plaintext_length )
```

where plaintext_length is the length in bytes of PLAINTEXT_2, and info for KEYSTREAM_2 is:

```
info =
(
 0,
 h'356efd53771425e008f3fe3a86c83ff4c6b16e57028ff39d
   5236c182b202084b',
 11
)
```

where the last value is the length in bytes of PLAINTEXT_2.

info for KEYSTREAM_2 (CBOR Sequence) (36 bytes)
00 58 20 35 6e fd 53 77 14 25 e0 08 f3 fe 3a 86 c8 3f f4 c6 b1 6e 57
02 8f f3 9d 52 36 c1 82 b2 02 08 4b 0b

KEYSTREAM_2 (Raw Value) (11 bytes)
bf 50 e9 e7 ba d0 bb 68 17 33 99

The Responder calculates CIPHERTEXT_2 as XOR between PLAINTEXT_2 and KEYSTREAM_2:

CIPHERTEXT_2 (Raw Value) (11 bytes)
98 62 a1 ee f9 e0 e7 e1 88 6f cd

The Responder constructs message_2:

```
message_2 =
(
 G_Y_CIPHERTEXT_2,
)
```

where G_Y_CIPHERTEXT_2 is the bstr encoding of the concatenation of the raw values of G_Y and CIPHERTEXT_2.

message_2 (CBOR Sequence) (45 bytes)
58 2b 41 97 01 d7 f0 0a 26 c2 dc 58 7a 36 dd 75 25 49 f3 37 63 c8 93
42 2c 8e a0 f9 55 a1 3a 4f f5 d5 98 62 a1 ee f9 e0 e7 e1 88 6f cd

3.5. message_3

The transcript hash TH_3 is calculated using the EDHOC hash algorithm:

```
TH_3 = H( TH_2, PLAINTEXT_2, CRED_R )
```

Input to calculate TH_3 (CBOR Sequence) (140 bytes)
58 20 35 6e fd 53 77 14 25 e0 08 f3 fe 3a 86 c8 3f f4 c6 b1 6e 57 02
8f f3 9d 52 36 c1 82 b2 02 08 4b 27 32 48 09 43 30 5c 89 9f 5c 54 a2
02 6b 65 78 61 6d 70 6c 65 2e 65 64 75 08 a1 01 a5 01 02 02 41 32 20
01 21 58 20 bb c3 49 60 52 6e a4 d3 2e 94 0c ad 2a 23 41 48 dd c2 17
91 a1 2a fb cb ac 93 62 20 46 dd 44 f0 22 58 20 45 19 e2 57 23 6b 2a
0c e2 02 3f 09 31 f1 f3 86 ca 7a fd a6 4f cd e0 10 8c 22 4c 51 ea bf
60 72

TH_3 (Raw Value) (32 bytes)
ad af 67 a7 8a 4b cc 91 e0 18 f8 88 27 62 a7 22 00 0b 25 07 03 9d f0
bc 1b bf 0c 16 1b b3 15 5c

TH_3 (CBOR Data Item) (34 bytes)
58 20 ad af 67 a7 8a 4b cc 91 e0 18 f8 88 27 62 a7 22 00 0b 25 07 03
9d f0 bc 1b bf 0c 16 1b b3 15 5c

Since METHOD = 3, the Initiator authenticates using static DH. The EDHOC key exchange algorithm is based on the same curve as for the ephemeral keys, which is P-256, since the selected cipher suite is 2.

The Initiator's static Diffie-Hellman P-256 key pair:

Initiator's private authentication key
SK_I (Raw Value) (32 bytes)
fb 13 ad eb 65 18 ce e5 f8 84 17 66 08 41 14 2e 83 0a 81 fe 33 43 80
a9 53 40 6a 13 05 e8 70 6b

Initiator's public authentication key, 'x'-coordinate
(Raw Value) (32 bytes)
ac 75 e9 ec e3 e5 0b fc 8e d6 03 99 88 95 22 40 5c 47 bf 16 df 96 66
0a 41 29 8c b4 30 7f 7e b6

Initiator's public authentication key, 'y'-coordinate
(Raw Value) (32 bytes)
6e 5d e6 11 38 8a 4b 8a 82 11 33 4a c7 d3 7e cb 52 a3 87 d2 57 e6 db
3c 2a 93 df 21 ff 3a ff c8

Since I authenticates with static DH (METHOD = 3), PRK_4e3m is derived from SALT_4e3m and G_IY.

The input needed to calculate SALT_4e3m is defined in Section 4.1.2 of [I-D.ietf-lake-edhoc], using EDHOC_Expand() with the EDHOC hash algorithm:

```
SALT_4e3m  = EDHOC_KDF( PRK_3e2m, 5, TH_3, hash_length ) =
           = HKDF-Expand( PRK_3e2m, info, hash_length )
```

where hash_length is the length in bytes of the output of the EDHOC hash algorithm, and info for SALT_4e3m is:

```
info =
(
 5,
 h'adaf67a78a4bcc91e018f8882762a722000b2507039df0bc
   1bbf0c161bb3155c',
 32
)
```

info for SALT_4e3m (CBOR Sequence) (37 bytes)
05 58 20 ad af 67 a7 8a 4b cc 91 e0 18 f8 88 27 62 a7 22 00 0b 25 07
03 9d f0 bc 1b bf 0c 16 1b b3 15 5c 18 20

SALT_4e3m (Raw Value) (32 bytes)
cf dd f9 51 5a 7e 46 e7 b4 db ff 31 cb d5 6c d0 4b a3 32 25 0d e9 ea
5d e1 ca f9 f6 d1 39 14 a7

PRK_4e3m is specified in Section 4.1.1.3 of [I-D.ietf-lake-edhoc].

Since I authenticates with static DH (METHOD = 3), PRK_4e3m is derived from G_IY using EDHOC_Extract() with the EDHOC hash algorithm:

```
PRK_4e3m = EDHOC_Extract(SALT_4e3m, G_IY) =
         = HMAC-SHA-256(SALT_4e3m, G_IY)
```

where G_IY is the ECDH shared secret calculated from G_I and Y, or G_Y and I.

G_IY (Raw Value) (ECDH shared secret) (32 bytes)
08 0f 42 50 85 bc 62 49 08 9e ac 8f 10 8e a6 23 26 85 7e 12 ab 07 d7
20 28 ca 1b 5f 36 e0 04 b3

PRK_4e3m (Raw Value) (32 bytes)
81 cc 8a 29 8e 35 70 44 e3 c4 66 bb 5c 0a 1e 50 7e 01 d4 92 38 ae ba
13 8d f9 46 35 40 7c 0f f7

The Initiator constructs the remaining input needed to calculate MAC_3:

```
MAC_3 = EDHOC_KDF( PRK_4e3m, 6, context_3, mac_length_3 )

context_3 = << ID_CRED_I, TH_3, CRED_I, ? EAD_3 >>
```

CRED_I is identified by a 'kid' with byte string value 0x2b:

```
ID_CRED_I =
{
 4 : h'2b'
}
```

ID_CRED_I (CBOR Data Item) (4 bytes)
a1 04 41 2b

CRED_I is an RPK encoded as a CCS:

```
{                                                    /CCS/
  2 : "42-50-31-FF-EF-37-32-39",                     /sub/
  8 : {                                              /cnf/
    1 : {                                            /COSE_Key/
      1 : 2,                                         /kty/
      2 : h'2b',                                     /kid/
     -1 : 1,                                         /crv/
     -2 : h'AC75E9ECE3E50BFC8ED6039988952240
             5C47BF16DF96660A41298CB4307F7EB6'       /x/
     -3 : h'6E5DE611388A4B8A8211334AC7D37ECB
             52A387D257E6DB3C2A93DF21FF3AFFC8'       /y/
    }
  }
}
```

CRED_I (CBOR Data Item) (107 bytes)

```
a2 02 77 34 32 2d 35 30 2d 33 31 2d 46 46 2d 45 46 2d 33 37 2d 33 32
2d 33 39 08 a1 01 a5 01 02 02 41 2b 20 01 21 58 20 ac 75 e9 ec e3 e5
0b fc 8e d6 03 99 88 95 22 40 5c 47 bf 16 df 96 66 0a 41 29 8c b4 30
7f 7e b6 22 58 20 6e 5d e6 11 38 8a 4b 8a 82 11 33 4a c7 d3 7e cb 52
a3 87 d2 57 e6 db 3c 2a 93 df 21 ff 3a ff c8
```

No external authorization data:

EAD_3 (CBOR Sequence) (0 bytes)

context_3 = << ID_CRED_I, TH_3, CRED_I, ? EAD_3 >>

context_3 (CBOR Sequence) (145 bytes)

```
a1 04 41 2b 58 20 ad af 67 a7 8a 4b cc 91 e0 18 f8 88 27 62 a7 22 00
0b 25 07 03 9d f0 bc 1b bf 0c 16 1b b3 15 5c a2 02 77 34 32 2d 35 30
2d 33 31 2d 46 46 2d 45 46 2d 33 37 2d 33 32 2d 33 39 08 a1 01 a5 01
02 02 41 2b 20 01 21 58 20 ac 75 e9 ec e3 e5 0b fc 8e d6 03 99 88 95
22 40 5c 47 bf 16 df 96 66 0a 41 29 8c b4 30 7f 7e b6 22 58 20 6e 5d
e6 11 38 8a 4b 8a 82 11 33 4a c7 d3 7e cb 52 a3 87 d2 57 e6 db 3c 2a
93 df 21 ff 3a ff c8
```

context_3 (CBOR byte string) (147 bytes)

```
58 91 a1 04 41 2b 58 20 ad af 67 a7 8a 4b cc 91 e0 18 f8 88 27 62 a7
22 00 0b 25 07 03 9d f0 bc 1b bf 0c 16 1b b3 15 5c a2 02 77 34 32 2d
35 30 2d 33 31 2d 46 46 2d 45 46 2d 33 37 2d 33 32 2d 33 39 08 a1 01
a5 01 02 02 41 2b 20 01 21 58 20 ac 75 e9 ec e3 e5 0b fc 8e d6 03 99
88 95 22 40 5c 47 bf 16 df 96 66 0a 41 29 8c b4 30 7f 7e b6 22 58 20
6e 5d e6 11 38 8a 4b 8a 82 11 33 4a c7 d3 7e cb 52 a3 87 d2 57 e6 db
3c 2a 93 df 21 ff 3a ff c8
```

MAC_3 is computed through EDHOC_Expand() using the EDHOC hash algorithm, see Section 4.1.2 of [I-D.ietf-lake-edhoc]:

MAC_3 = HKDF-Expand(PRK_4e3m, info, mac_length_3), where

info = (6, context_3, mac_length_3)

Since METHOD = 3, mac_length_3 is given by the EDHOC MAC length.

info for MAC_3 is:

```
info =
(
 6,
 h'a104412b5820adaf67a78a4bcc91e018f8882762a722000b
   2507039df0bc1bbf0c161bb3155ca2027734322d35302d33
   312d46462d45462d33372d33322d333908a101a501020241
   2b2001215820ac75e9ece3e50bfc8ed60399889522405c47
   bf16df96660a41298cb4307f7eb62258206e5de611388a4b
   8a8211334ac7d37ecb52a387d257e6db3c2a93df21ff3aff
   c8',
 8
)
```

where the last value is the EDHOC MAC length in bytes.

info for MAC_3 (CBOR Sequence) (149 bytes)
06 58 91 a1 04 41 2b 58 20 ad af 67 a7 8a 4b cc 91 e0 18 f8 88 27 62
a7 22 00 0b 25 07 03 9d f0 bc 1b bf 0c 16 1b b3 15 5c a2 02 77 34 32
2d 35 30 2d 33 31 2d 46 46 2d 45 46 2d 33 37 2d 33 32 2d 33 39 08 a1
01 a5 01 02 02 41 2b 20 01 21 58 20 ac 75 e9 ec e3 e5 0b fc 8e d6 03
99 88 95 22 40 5c 47 bf 16 df 96 66 0a 41 29 8c b4 30 7f 7e b6 22 58
20 6e 5d e6 11 38 8a 4b 8a 82 11 33 4a c7 d3 7e cb 52 a3 87 d2 57 e6
db 3c 2a 93 df 21 ff 3a ff c8 08

MAC_3 (Raw Value) (8 bytes)
62 3c 91 df 41 e3 4c 2f

MAC_3 (CBOR Data Item) (9 bytes)
48 62 3c 91 df 41 e3 4c 2f

Since METHOD = 3, Signature_or_MAC_3 is MAC_3:

Signature_or_MAC_3 (Raw Value) (8 bytes)
62 3c 91 df 41 e3 4c 2f

Signature_or_MAC_3 (CBOR Data Item) (9 bytes)
48 62 3c 91 df 41 e3 4c 2f

The Initiator constructs PLAINTEXT_3:

```
PLAINTEXT_3 =
(
 ID_CRED_I / bstr / -24..23,
 Signature_or_MAC_3,
 ? EAD_3
)
```

Since ID_CRED_I contains a single 'kid' parameter, only the byte string value is included in the plaintext, represented as described in Section 3.3.2 of [I-D.ietf-lake-edhoc]. The CBOR map { 4 : h'2b' } is thus replaced, not by the CBOR byte string 0x412b, but by the CBOR int 0x2b, since that is a one byte encoding of a CBOR integer (-12).

```
PLAINTEXT_3 (CBOR Sequence) (10 bytes)
2b 48 62 3c 91 df 41 e3 4c 2f
```

The Initiator constructs the associated data for message_3:

```
A_3 =
[
 "Encrypt0",
 h'',
 h'adaf67a78a4bcc91e018f8882762a722000b2507039df0bc
   1bbf0c161bb3155c'
]
```

```
A_3 (CBOR Data Item) (45 bytes)
83 68 45 6e 63 72 79 70 74 30 40 58 20 ad af 67 a7 8a 4b cc 91 e0 18
f8 88 27 62 a7 22 00 0b 25 07 03 9d f0 bc 1b bf 0c 16 1b b3 15 5c
```

The Initiator constructs the input needed to derive the key K_3, see Section 4.1.2 of [I-D.ietf-lake-edhoc], using the EDHOC hash algorithm:

```
K_3 = EDHOC_KDF( PRK_3e2m, 3, TH_3, key_length )
    = HKDF-Expand( PRK_3e2m, info, key_length ),
```

where key_length is the key length in bytes for the EDHOC AEAD algorithm, and info for K_3 is:

```
info =
(
 3,
 h'adaf67a78a4bcc91e018f8882762a722000b2507039df0bc
   1bbf0c161bb3155c',
 16
)
```

where the last value is the key length in bytes for the EDHOC AEAD algorithm.

info for K_3 (CBOR Sequence) (36 bytes)
03 58 20 ad af 67 a7 8a 4b cc 91 e0 18 f8 88 27 62 a7 22 00 0b 25 07
03 9d f0 bc 1b bf 0c 16 1b b3 15 5c 10

K_3 (Raw Value) (16 bytes)
8e 7a 30 04 20 00 f7 90 0e 81 74 13 1f 75 f3 ed

The Initiator constructs the input needed to derive the nonce IV_3, see Section 4.1.2 of [I-D.ietf-lake-edhoc], using the EDHOC hash algorithm:

IV_3 = EDHOC_KDF(PRK_3e2m, 4, TH_3, iv_length)
= HKDF-Expand(PRK_3e2m, info, iv_length),

where iv_length is the nonce length in bytes for the EDHOC AEAD algorithm, and info for IV_3 is:

```
info =
(
 4,
 h'adaf67a78a4bcc91e018f8882762a722000b2507039df0bc
   1bbf0c161bb3155c',
 13
)
```

where the last value is the nonce length in bytes for the EDHOC AEAD algorithm.

info for IV_3 (CBOR Sequence) (36 bytes)
04 58 20 ad af 67 a7 8a 4b cc 91 e0 18 f8 88 27 62 a7 22 00 0b 25 07
03 9d f0 bc 1b bf 0c 16 1b b3 15 5c 0d

IV_3 (Raw Value) (13 bytes)
6d 83 00 c1 e2 3b 56 15 3a e7 0e e4 57

The Initiator calculates CIPHERTEXT_3 as 'ciphertext' of COSE_Encrypt0 applied using the EDHOC AEAD algorithm with plaintext PLAINTEXT_3, additional data A_3, key K_3 and nonce IV_3.

CIPHERTEXT_3 (Raw Value) (18 bytes)
e5 62 09 7b c4 17 dd 59 19 48 5a c7 89 1f fd 90 a9 fc

message_3 is the CBOR bstr encoding of CIPHERTEXT_3:

message_3 (CBOR Sequence) (19 bytes)
52 e5 62 09 7b c4 17 dd 59 19 48 5a c7 89 1f fd 90 a9 fc

The transcript hash TH_4 is calculated using the EDHOC hash algorithm:

TH_4 = H(TH_3, PLAINTEXT_3, CRED_I)

Input to calculate TH_4 (CBOR Sequence) (151 bytes)
58 20 ad af 67 a7 8a 4b cc 91 e0 18 f8 88 27 62 a7 22 00 0b 25 07 03
9d f0 bc 1b bf 0c 16 1b b3 15 5c 2b 48 62 3c 91 df 41 e3 4c 2f a2 02
77 34 32 2d 35 30 2d 33 31 2d 46 46 2d 45 46 2d 33 37 2d 33 32 2d 33
39 08 a1 01 a5 01 02 02 41 2b 20 01 21 58 20 ac 75 e9 ec e3 e5 0b fc
8e d6 03 99 88 95 22 40 5c 47 bf 16 df 96 66 0a 41 29 8c b4 30 7f 7e
b6 22 58 20 6e 5d e6 11 38 8a 4b 8a 82 11 33 4a c7 d3 7e cb 52 a3 87
d2 57 e6 db 3c 2a 93 df 21 ff 3a ff c8

TH_4 (Raw Value) (32 bytes)
c9 02 b1 e3 a4 32 6c 93 c5 55 1f 5f 3a a6 c5 ec c0 24 68 06 76 56 12
e5 2b 5d 99 e6 05 9d 6b 6e

TH_4 (CBOR Data Item) (34 bytes)
58 20 c9 02 b1 e3 a4 32 6c 93 c5 55 1f 5f 3a a6 c5 ec c0 24 68 06 76
56 12 e5 2b 5d 99 e6 05 9d 6b 6e

3.6. message_4

No external authorization data:

EAD_4 (CBOR Sequence) (0 bytes)

The Responder constructs PLAINTEXT_4:

```
PLAINTEXT_4 =
(
 ? EAD_4
)
```

PLAINTEXT_4 (CBOR Sequence) (0 bytes)

The Responder constructs the associated data for message_4:

```
A_4 =
[
 "Encrypt0",
 h'',
 h'c902b1e3a4326c93c5551f5f3aa6c5ecc0246806765612e5
   2b5d99e6059d6b6e'
]
```

```
A_4 (CBOR Data Item) (45 bytes)
83 68 45 6e 63 72 79 70 74 30 40 58 20 c9 02 b1 e3 a4 32 6c 93 c5 55
1f 5f 3a a6 c5 ec c0 24 68 06 76 56 12 e5 2b 5d 99 e6 05 9d 6b 6e
```

The Responder constructs the input needed to derive the EDHOC message_4 key, see Section 4.1.2 of [I-D.ietf-lake-edhoc], using the EDHOC hash algorithm:

```
K_4   = EDHOC_KDF( PRK_4e3m, 8, TH_4, key_length )
      = HKDF-Expand( PRK_4e3m, info, key_length )
```

where key_length is the key length in bytes for the EDHOC AEAD algorithm, and info for K_4 is:

```
info =
(
 8,
 h'c902b1e3a4326c93c5551f5f3aa6c5ecc0246806765612e5
   2b5d99e6059d6b6e',
 16
)
```

where the last value is the key length in bytes for the EDHOC AEAD algorithm.

```
info for K_4 (CBOR Sequence) (36 bytes)
08 58 20 c9 02 b1 e3 a4 32 6c 93 c5 55 1f 5f 3a a6 c5 ec c0 24 68 06
76 56 12 e5 2b 5d 99 e6 05 9d 6b 6e 10
```

```
K_4 (Raw Value) (16 bytes)
d3 c7 78 72 b6 ee b5 08 91 1b db d3 08 b2 e6 a0
```

The Responder constructs the input needed to derive the EDHOC message_4 nonce, see Section 4.1.2 of [I-D.ietf-lake-edhoc], using the EDHOC hash algorithm:

```
      IV_4 = EDHOC_KDF( PRK_4e3m, 9, TH_4, iv_length )
           = HKDF-Expand( PRK_4e3m, info, iv_length )
```

where iv_length is the nonce length in bytes for the EDHOC AEAD algorithm, and info for IV_4 is:

```
info =
(
 9,
 h'c902b1e3a4326c93c5551f5f3aa6c5ecc0246806765612e5
   2b5d99e6059d6b6e',
 13
)
```

where the last value is the nonce length in bytes for the EDHOC AEAD algorithm.

info for IV_4 (CBOR Sequence) (36 bytes)
09 58 20 c9 02 b1 e3 a4 32 6c 93 c5 55 1f 5f 3a a6 c5 ec c0 24 68 06
76 56 12 e5 2b 5d 99 e6 05 9d 6b 6e 0d

IV_4 (Raw Value) (13 bytes)
04 ff 0f 44 45 6e 96 e2 17 85 3c 36 01

The Responder calculates CIPHERTEXT_4 as 'ciphertext' of COSE_Encrypt0 applied using the EDHOC AEAD algorithm with plaintext PLAINTEXT_4, additional data A_4, key K_4 and nonce IV_4.

CIPHERTEXT_4 (8 bytes)
28 c9 66 b7 ca 30 4f 83

message_4 is the CBOR bstr encoding of CIPHERTEXT_4:

message_4 (CBOR Sequence) (9 bytes)
48 28 c9 66 b7 ca 30 4f 83

3.7. PRK_out and PRK_exporter

PRK_out is specified in Section 4.1.3 of [I-D.ietf-lake-edhoc].

```
PRK_out = EDHOC_KDF( PRK_4e3m, 7, TH_4, hash_length ) =
        = HKDF-Expand( PRK_4e3m, info,  hash_length )
```

where hash_length is the length in bytes of the output of the EDHOC hash algorithm, and info for PRK_out is:

```
info =
(
 7,
 h'c902b1e3a4326c93c5551f5f3aa6c5ecc0246806765612e5
   2b5d99e6059d6b6e',
 32
)
```

where the last value is the length in bytes of the output of the EDHOC hash algorithm.

info for PRK_out (CBOR Sequence) (37 bytes)
07 58 20 c9 02 b1 e3 a4 32 6c 93 c5 55 1f 5f 3a a6 c5 ec c0 24 68 06
76 56 12 e5 2b 5d 99 e6 05 9d 6b 6e 18 20

PRK_out (Raw Value) (32 bytes)
2c 71 af c1 a9 33 8a 94 0b b3 52 9c a7 34 b8 86 f3 0d 1a ba 0b 4d c5
1b ee ae ab df ea 9e cb f8

The OSCORE Master Secret and OSCORE Master Salt are derived with the EDHOC_Exporter as specified in 4.2.1 of [I-D.ietf-lake-edhoc].

```
EDHOC_Exporter( label, context, length )
= EDHOC_KDF( PRK_exporter, label, context, length )
```

where PRK_exporter is derived from PRK_out:

```
 PRK_exporter = EDHOC_KDF( PRK_out, 10, h'', hash_length ) =
              = HKDF-Expand( PRK_out, info,  hash_length )
```

where hash_length is the length in bytes of the output of the EDHOC hash algorithm, and info for the PRK_exporter is:

```
info =
(
 10,
 h'',
 32
)
```

where the last value is the length in bytes of the output of the EDHOC hash algorithm.

info for PRK_exporter (CBOR Sequence) (4 bytes)
0a 40 18 20

PRK_exporter (Raw Value) (32 bytes)
e1 4d 06 69 9c ee 24 8c 5a 04 bf 92 27 bb cd 4c e3 94 de 7d cb 56 db
43 55 54 74 17 1e 64 46 db

3.8. OSCORE Parameters

The derivation of OSCORE parameters is specified in Appendix A.1 of [I-D.ietf-lake-edhoc].

The AEAD and Hash algorithms to use in OSCORE are given by the selected cipher suite:

Application AEAD Algorithm (int)
10

Application Hash Algorithm (int)
-16

The mapping from EDHOC connection identifiers to OSCORE Sender/ Recipient IDs is defined in Section 3.3.3 of [I-D.ietf-lake-edhoc].

C_R is mapped to the Recipient ID of the server, i.e., the Sender ID of the client. The byte string 0x27, which as C_R is encoded as the CBOR integer 0x27, is converted to the server Recipient ID 0x27.

Client's OSCORE Sender ID (Raw Value) (1 byte)
27

C_I is mapped to the Recipient ID of the client, i.e., the Sender ID of the server. The byte string 0x37, which as C_I is encoded as the CBOR integer 0x0e is converted to the client Recipient ID 0x37.

Server's OSCORE Sender ID (Raw Value) (1 byte)
37

The OSCORE Master Secret is computed through EDHOC_Expand() using the Application hash algorithm, see Appendix A.1 of [I-D.ietf-lake-edhoc]:

OSCORE Master Secret = EDHOC_Exporter(0, h'', oscore_key_length)
= EDHOC_KDF(PRK_exporter, 0, h'', oscore_key_length)
= HKDF-Expand(PRK_exporter, info, oscore_key_length)

where oscore_key_length is by default the key length in bytes for the Application AEAD algorithm, and info for the OSCORE Master Secret is:

```
info =
(
 0,
 h'',
 16
)
```

where the last value is the key length in bytes for the Application AEAD algorithm.

info for OSCORE Master Secret (CBOR Sequence) (3 bytes)
00 40 10

OSCORE Master Secret (Raw Value) (16 bytes)
f9 86 8f 6a 3a ca 78 a0 5d 14 85 b3 50 30 b1 62

The OSCORE Master Salt is computed through EDHOC_Expand() using the Application hash algorithm, see Section 4.2 of [I-D.ietf-lake-edhoc]:

OSCORE Master Salt = EDHOC_Exporter(1, h'', oscore_salt_length)
= EDHOC_KDF(PRK_exporter, 1, h'', oscore_salt_length)
= HKDF-Expand(PRK_4x3m, info, oscore_salt_length)

where oscore_salt_length is the length in bytes of the OSCORE Master Salt, and info for the OSCORE Master Salt is:

```
info =
(
 1,
 h'',
 8
)
```

where the last value is the length in bytes of the OSCORE Master Salt.

info for OSCORE Master Salt (CBOR Sequence) (3 bytes)
01 40 08

OSCORE Master Salt (Raw Value) (8 bytes)
ad a2 4c 7d bf c8 5e eb

3.9. Key Update

Key update is defined in Appendix H of [I-D.ietf-lake-edhoc].

```
EDHOC_KeyUpdate( context ):
PRK_out = EDHOC_KDF( PRK_out, 11, context, hash_length )
        = HKDF-Expand( PRK_out, info, hash_length )
```

where hash_length is the length in bytes of the output of the EDHOC hash function, context for KeyUpdate is

context for KeyUpdate (Raw Value) (16 bytes)
a0 11 58 fd b8 20 89 0c d6 be 16 96 02 b8 bc ea

context for KeyUpdate (CBOR Data Item) (17 bytes)
50 a0 11 58 fd b8 20 89 0c d6 be 16 96 02 b8 bc ea

and where info for key update is:

```
info =
(
 11,
 h'a01158fdb820890cd6be169602b8bcea',
 32
)
```

PRK_out after KeyUpdate (Raw Value) (32 bytes)
f9 79 53 77 43 fe 0b d6 b9 b1 41 dd bd 79 65 6c 52 e6 dc 7c 50 ad 80 77 54 d7 4d 07 e8 7d 0d 16

After key update the PRK_exporter needs to be derived anew:

```
 PRK_exporter = EDHOC_KDF( PRK_out, 10, h'', hash_length ) =
              = HKDF-Expand( PRK_out, info,  hash_length )
```

where info and hash_length are unchanged as in Section 3.7.

PRK_exporter after KeyUpdate (Raw Value) (32 bytes)
00 fc f7 db 9b 2e ad 73 82 4e 7e 83 03 63 c8 05 c2 96 f9 02 83 0f ac 23 d8 6c 35 9c 75 2f 0f 17

The OSCORE Master Secret is derived with the updated PRK_exporter:

```
OSCORE Master Secret =
= HKDF-Expand(PRK_exporter, info, oscore_key_length)
```

where info and key_length are unchanged as in Section 2.6.

OSCORE Master Secret after KeyUpdate (Raw Value) (16 bytes)
49 f7 2f ac 02 b4 65 8b da 21 e2 da c6 6f c3 74

The OSCORE Master Salt is derived with the updated PRK_exporter:

OSCORE Master Salt = HKDF-Expand(PRK_exporter, info, salt_length)

where info and salt_length are unchanged as in Section 2.6.

OSCORE Master Salt after KeyUpdate (Raw Value) (8 bytes)
dd 8b 24 f2 aa 9b 01 1a

4. Invalid Traces

This section contains examples of invalid messages, which a compliant implementation will not compose and must or may reject according to [I-D.ietf-lake-edhoc], [RFC8949], [RFC9053], and [SP-800-56A]. This is just a small set of examples of different reasons a message might be invalid. The same types of invalidities applies to other fields and messages as well. Implementations should make sure to check for similar types of invalidities in all EHDOC fields and messages.

4.1. Encoding Errors

4.1.1. Surplus array encoding of message

Invalid encoding of message_1 as array. Correct encoding is a CBOR sequence according to Section 5.2.1 of [I-D.ietf-lake-edhoc].

Invalid message_1 (38 bytes)
84 03 02 58 20 74 1a 13 d7 ba 04 8f bb 61 5e 94 38 6a a3 b6 1b ea 5b 3d 8f 65 f3 26 20 b7 49 be e8 d2 78 ef a9 0e

4.1.2. Surplus bstr encoding of connection identifier

Invalid encoding 41 0e of C_I = 0x0e. Correct encoding is 0e according to Section 3.3.2 of [I-D.ietf-lake-edhoc].

Invalid message_1 (38 bytes)
03 02 58 20 74 1a 13 d7 ba 04 8f bb 61 5e 94 38 6a a3 b6 1b ea 5b 3d 8f 65 f3 26 20 b7 49 be e8 d2 78 ef a9 41 0e

4.1.3. Surplus array encoding of ciphersuite

Invalid array encoding 81 02 of SUITES_I = 2. Correct encoding is 02 according to Section 5.2.2 of [I-D.ietf-lake-edhoc].

Invalid message_1 (38 bytes)
03 81 02 58 20 74 1a 13 d7 ba 04 8f bb 61 5e 94 38 6a a3 b6 1b ea 5b 3d 8f 65 f3 26 20 b7 49 be e8 d2 78 ef a9 0e

4.1.4. Text string encoding of ephemeral key

Invalid type of the third element (G_X). Correct encoding is a byte string according to Section 5.2.1 of [I-D.ietf-lake-edhoc].

Invalid message_1 (37 bytes)
03 02 78 20 20 61 69 72 20 73 70 65 65 64 20 6F 66 20 61 20 75 6E 6C 61 64 65 6E 20 73 77 61 6C 6C 6F 77 20 0e

4.1.5. Wrong number of CBOR sequence elements

Invalid number of elements in the CBOR sequence. Correct number of elements is 1 according to Section 5.3.1 of [I-D.ietf-lake-edhoc].

Invalid message_2 (46 bytes)
58 20 41 97 01 d7 f0 0a 26 c2 dc 58 7a 36 dd 75 25 49 f3 37 63 c8 93 42 2c 8e a0 f9 55 a1 3a 4f f5 d5 4B 98 62 a1 1d e4 2a 95 d7 85 38 6a

4.1.6. Surplus map encoding of ID_CRED field

Invalid encoding a1 04 42 32 10 of ID_CRED_R in PLAINTEXT_2. Correct encoding is 42 32 10 according to Section 3.5.3.2 of [I-D.ietf-lake-edhoc].

Invalid PLAINTEXT_2 (15 bytes)
27 a1 04 42 32 10 48 fa 5e fa 2e bf 92 0b f3

4.1.7. Surplus bstr encoding of ID_CRED field

Invalid encoding 41 32 of ID_CRED_R in PLAINTEXT_2. Correct encoding is 32 according to Section 3.5.3.2 of [I-D.ietf-lake-edhoc].

Invalid PLAINTEXT_2 (12 bytes)
27 41 32 48 fa 5e fa 2e bf 92 0b f3

4.2. Crypto-related Errors

4.2.1. Error in length of ephemeral key

Invalid length of the third element (G_X). Selected cipher suite is cipher suite 24 with curve P-384 according to Sections 5.2.2, and 10.2 of [I-D.ietf-lake-edhoc]. Correct length of x-coordinate is 48 bytes according to Section 3.7 of [I-D.ietf-lake-edhoc] and Section 7.1.1 of [RFC9053].

Invalid message_1 (40 bytes)
03 82 02 18 18 58 20 74 1a 13 d7 ba 04 8f bb 61 5e 94 38 6a a3 b6 1b ea 5b 3d 8f 65 f3 26 20 b7 49 be e8 d2 78 ef a9 0e

4.2.2. Error in elliptic curve representation

Invalid x-coordinate in G_X as x p. Requirement that x < p according to Section 9.2 of [I-D.ietf-lake-edhoc] and Section 5.6.2.3 of [SP-800-56A].

Invalid message_1 (37 bytes)
03 02 58 20 ff ff ff ff 00 00 00 01 00 00 00 00 00 00 00 00 00 00
00 ff ff ff ff ff ff ff ff ff ff ff ff 0e

4.2.3. Error in elliptic curve point

Invalid x-coordinate in (G_X) not corresponding to a point on the P-256 curve. Requirement that y^2 x^3 + a x + b (mod p) according to Section 9.2 of [I-D.ietf-lake-edhoc] and Section 5.6.2.3 of [SP-800-56A].

Invalid message_1 (37 bytes)
03 02 58 20 a0 4e 73 60 1d f5 44 a7 0b a7 ea 1e 57 03 0f 7d 4b 4e b7
f6 73 92 4e 58 d5 4c a7 7a 5e 7d 4d 4a 0e

4.2.4. Curve point of low order

Curve25519 point of low order which fails the check for all-zero output according to Section 9.2 of [I-D.ietf-lake-edhoc].

Invalid message_1 (37 bytes)
03 00 58 20 ed ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff 7f 0e

4.2.5. Error in length of MAC

Invalid length of third element (Signature_or_MAC_2). The length of Signature_or_MAC_2 is given by the cipher suite and the MAC length is at least 8 bytes according to Section 9.3 of [I-D.ietf-lake-edhoc].

Invalid PLAINTEXT_2 (7 bytes)
27 32 44 fa 5e fa 2e

4.2.6. Error in elliptic curve encoding

Invalid encoding of third element (G_X). Correct encoding is with leading zeros according to Section 3.7 of [I-D.ietf-lake-edhoc] and Section 7.1.1 of [RFC9053].

Invalid message_1 (36 bytes)
03 02 58 1f d9 69 77 25 d2 3a 68 8b 12 d1 c7 e0 10 8a 08 c9 f7 1a 85
a0 9c 20 81 49 76 ab 21 12 22 48 fc 0e

4.3. Non-deterministic CBOR

4.3.1. Unnecessary long encoding

Invalid 16-bit encoding 19 00 03 of METHOD = 3. Correct is the deterministic encoding 03 according to Section 3.1 of [I-D.ietf-lake-edhoc] and Section 4.2.1 of [RFC8949], which states that the arguments for integers, lengths in major types 2 through 5, and tags are required to be as short as possible.

Invalid message_1 (39 bytes)
19 00 03 02 58 20 74 1a 13 d7 ba 04 8f bb 61 5e 94 38 6a a3 b6 1b ea 5b 3d 8f 65 f3 26 20 b7 49 be e8 d2 78 ef a9 0e

4.3.2. Indefinite-length array encoding

Invalid indefinite-length array encoding 9F 06 02 FF of SUITES_I = [6, 2]. Correct encoding is 82 06 02 according to Section 5.2.2 of [I-D.ietf-lake-edhoc].

Invalid message_1 (40 bytes)
03 9F 06 02 FF 58 20 74 1a 13 d7 ba 04 8f bb 61 5e 94 38 6a a3 b6 1b ea 5b 3d 8f 65 f3 26 20 b7 49 be e8 d2 78 ef a9 0e

5. Security Considerations

This document contains examples of EDHOC [I-D.ietf-lake-edhoc] whose security considerations apply. The keys printed in these examples cannot be considered secret and MUST NOT be used.

6. IANA Considerations

There are no IANA considerations.

7. References

7.1. Normative References

[I-D.ietf-lake-edhoc]
Selander, G., Mattsson, J. P., and F. Palombini, "Ephemeral Diffie-Hellman Over COSE (EDHOC)", Work in Progress, Internet-Draft, draft-ietf-lake-edhoc-23, 22 January 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-lake-edhoc-23>.

[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/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

7.2. Informative References

[CborMe] Bormann, C., "CBOR playground", August 2023, <https://cbor.me/>.

[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/rfc/rfc7252>.

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

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/rfc/rfc8032>.

[RFC8392] Jones, M., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "CBOR Web Token (CWT)", RFC 8392, DOI 10.17487/RFC8392, May 2018, <https://www.rfc-editor.org/rfc/rfc8392>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/rfc/rfc8949>.

[RFC9053] Schaad, J., "CBOR Object Signing and Encryption (COSE): Initial Algorithms", RFC 9053, DOI 10.17487/RFC9053, August 2022, <https://www.rfc-editor.org/rfc/rfc9053>.

[SP-800-186] Chen, L., Moody, D., Randall, K., Regenscheid, A., and A. Robinson, "Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters", NIST Special Publication 800-186, February 2023, <https://doi.org/10.6028/NIST.SP.800-186>.

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

Acknowledgments

The authors want to thank all people verifying EDHOC test vectors and/or contributing to the interoperability testing including: Christian Amsüss, Timothy Claeys, Stefan Hristozov, Rikard Höglund, Christos Koulamas, Francesca Palombini, Lidia Pocero, Peter van der Stok, and Michel Veillette.

Authors' Addresses

Göran Selander
Ericsson
Sweden
Email: goran.selander@ericsson.com

John Preuß Mattsson
Ericsson
Sweden
Email: john.mattsson@ericsson.com

Marek Serafin
ASSA ABLOY
Poland
Email: marek.serafin@assaabloy.com

Marco Tiloca
RISE
Sweden
Email: marco.tiloca@ri.se

Malia Vuini
Inria
France
Email: malisa.vucinic@inria.fr

LAKE Working Group
Internet-Draft
Intended status: Informational
Expires: 15 August 2024

M. Tiloca
RISE AB

12 February 2024

Implementation Considerations for Ephemeral Diffie-Hellman Over COSE (EDHOC)
draft-tiloca-lake-edhoc-implem-cons-01

Abstract

This document provides considerations for guiding the implementation of the authenticated key exchange protocol Ephemeral Diffie-Hellman Over COSE (EDHOC).

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the Lightweight Authenticated Key Exchange Working Group mailing list (lake@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/lake/.

Source for this draft and an issue tracker can be found at https://gitlab.com/crimson84/draft-tiloca-lake-edhoc-implem-cons.

Status of This Memo

Table of Contents

1. Introduction

Ephemeral Diffie-Hellman Over COSE (EDHOC) [I-D.ietf-lake-edhoc] is a lightweight authenticated key exchange protocol, especially intended for use in constrained scenarios.

During the development of EDHOC, a number of side topics were raised and discussed, as emerging from reviews of the protocol latest design and from implementation activities. These topics were identified as strongly pertaining to the implementation of EDHOC rather than to the

protocol in itself. Hence, they are not discussed in [I-D.ietf-lake-edhoc], which rightly focuses on specifying the actual protocol.

At the same time, implementors of an application using the EDHOC protocol or of an "EDHOC library" enabling its use cannot simply ignore such topics, and will have to take them into account throughout their implementation work.

In order to prevent multiple, independent re-discoveries and assessments of those topics, as well as to facilitate and guide implementation activities, this document collects such topics and discusses them through considerations about the implementation of EDHOC. At a high-level, the topics in question are summarized below.

* Handling of completed EDHOC sessions when they become invalid, and of application keys derived from an EDHOC session when those become invalid. This topic is discussed in Section 2.

* Enforcing of different trust models, with respect to learning new authentication credentials during an execution of EDHOC. This topic is discussed in Section 3.

* Branched-off, side processing of incoming EDHOC messages, with particular reference to: i) fetching and validation of authentication credentials; and ii) processing of External Authorization Data (EAD) items, which in turn might play a role in the fetching and validation of authentication credentials. This topic is discussed in Section 4.

1.1. Terminology

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

The reader is expected to be familiar with terms and concepts related to the EDHOC protocol and defined in [I-D.ietf-lake-edhoc].

2. Handling of Invalid EDHOC Sessions and Application Keys

This section considers the most common situation where, given a certain peer, only the application at that peer has visibility and control of both:

* The EDHOC sessions at that peer; and

* The application keys for that application at that peer, including the knowledge of whether they have been derived from an EDHOC session, i.e., by means of the EDHOC_Exporter interface after the successful completion of an execution of EDHOC (see Section 4.1 of [I-D.ietf-lake-edhoc]).

Building on the above, the following expands on three relevant cases concerning the handling of EDHOC sessions and application keys, in the event that any of those becomes invalid.

As a case in point to provide more concrete guidance, the following also considers the specific case where "applications keys" stands for the keying material and parameters that compose an OSCORE Security Context [RFC8613] and that are derived from an EDHOC session (see Appendix A.1 of [I-D.ietf-lake-edhoc]).

Nevertheless, the same considerations are applicable in case EDHOC is used to derive other application keys, e.g., to key different security protocols than OSCORE or to provide the application with secure values bound to an EDHOC session.

2.1. EDHOC Sessions Become Invalid

The application at a peer P may have learned that a completed EDHOC session S has to be invalidated. When S is marked as invalid, the application at P purges S and deletes each set of application keys (e.g., the OSCORE Security Context) that was generated from S.

Then, the applications runs a new execution of the EDHOC protocol with the other peer. Upon successfully completing the EDHOC execution, the two peers derive and install a new set of application keys from this latest EDHOC session.

The flowchart in Figure 1 shows the handling of an EDHOC session that has become invalid.

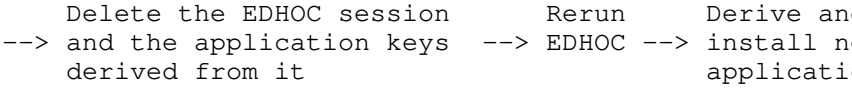


Figure 1: Handling of an EDHOC Session that has Become Invalid

An EDHOC session may have become invalid, for example, because an authentication credential CRED_X may have expired, or because P may have learned from a trusted source that CRED_X has been revoked. This effectively invalidates CRED_X, and therefore also invalidates any EDHOC session where CRED_X was used as authentication credential of either peer in the session (i.e., P itself or the other peer). In such a case, the application at P has to additionally delete CRED_X and any stored, corresponding credential identifier.

2.2. Application Keys Become Invalid

The application at a peer P may have learned that a set of application keys is not safe to use anymore. When such a set is specifically an OSCORE Security Context, the application may have learned that from the used OSCORE library or from an OSCORE layer that takes part to the communication stack.

A current set SET of application keys shared with another peer can become unsafe to use, for example, due to the following reasons.

* SET has reached its pre-determined expiration time; or

* SET has been established for a pre-defined, now elapsed amount of time, according to enforced application policies; or

* Some elements of SET have been used enough times to approach cryptographic limits that should not be passed, e.g., according to the properties of the specifically used security algorithms. With particular reference to an OSCORE Security Context, such limits are discussed in [I-D.ietf-core-oscore-key-limits].

When this happens, the application at the peer P proceeds as follows.

1. If the following conditions both hold, then the application moves to step 2. Otherwise, it moves to step 3.

    * Let us define S as the EDHOC session from which the peer P has derived SET or the eldest SET's ancestor set of application keys. Then, since the completion of S with the other peer, the application at P has received from the other peer at least one message protected with any set of application keys derived from S. That is, P has persisted S (see Section 5.4.2 of [I-D.ietf-lake-edhoc]).

   * The peer P supports a key update protocol, as an alternative to performing a new execution of EDHOC with the other peer. When SET is specifically an OSCORE Security Context, this means that the peer P supports the key update protocol KUDOS defined in [I-D.ietf-core-oscore-key-update].

2. The application at P runs the key update protocol mentioned at step 1 with the other peer, in order to update SET. When SET is specifically an OSCORE Security Context, this means that the application at P runs KUDOS with the other peer.

   If the key update protocol terminates successfully, the updated application keys are installed and no further actions are taken. Otherwise, the application at P moves to step 3.

3. The application at the peer P performs the following actions.

   * It deletes SET.

   * It deletes the EDHOC session from which SET was generated, or from which the eldest SET's ancestor set of application keys was generated before any key update occurred (e.g., by means of the EDHOC_KeyUpdate interface defined in Appendix H of [I-D.ietf-lake-edhoc] or other key update methods).

   * It runs a new execution of the EDHOC protocol with the other peer. Upon successfully completing the EDHOC execution, the two peers derive and install a new set of application keys from this latest EDHOC session.

The flowchart in Figure 2 shows the handling of a set of application keys that has become invalid.

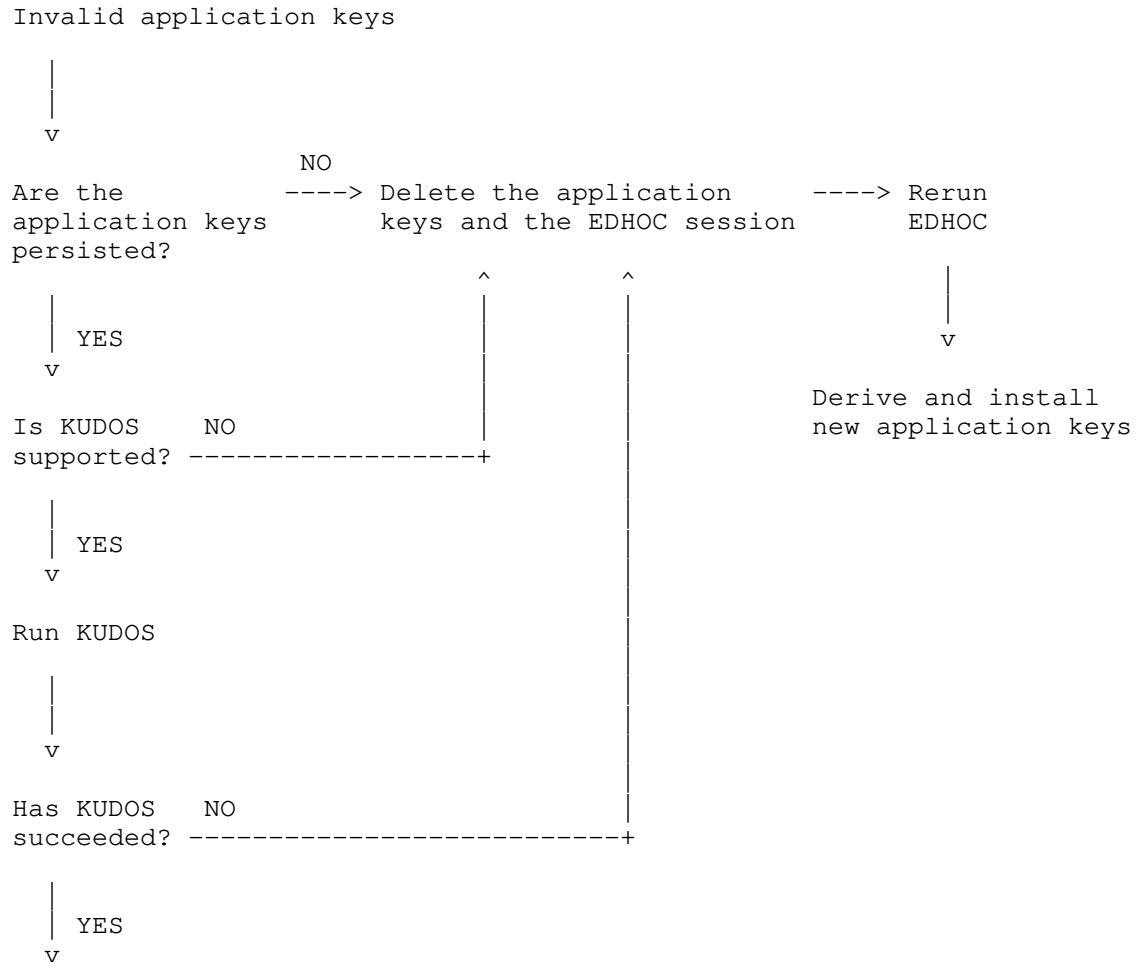


Figure 2: Handling of a set of Application Keys that has Become Invalid

2.3. Application Keys or Bound Access Rights Become Invalid

The following considers two peers that use the ACE framework for authentication and authorization in constrained environments (ACE) [RFC9200], and specifically the EDHOC and OSCORE profile of ACE defined in [I-D.ietf-ace-edhoc-oscore-profile].

When doing so, one of the two peers acts as ACE Resource Server (RS) hosting protected resources. The other peer acts as ACE Client, requests from an ACE Authorization Server (AS) an Access Token that specifies access rights for accessing protected resources at the RS, and uploads the Access Token to the RS as part of the ACE workflow.

Consistent with the used EDHOC and OSCORE profile of ACE, the two peers run EDHOC in order to specifically derive an OSCORE Security Context as their shared set of application keys (see Appendix A.1 of [I-D.ietf-lake-edhoc]). In particular, the peer acting as ACE Client acts as EDHOC Initiator, while the peer acting as ACE RS acts as EDHOC Responder (see Section 2 of [I-D.ietf-lake-edhoc]). The successfully completed EDHOC session is bound to the Access Token.

After that, the peer acting as ACE Client can access the protected resources hosted at the other peer, according to the access rights specified in the Access Token. The communications between the two peers are protected by means of the established OSCORE Security Context, which is also bound to the used Access Token.

Later on, the application at one of the two peers P may have learned that the established OSCORE Security Context CTX is not safe to use anymore, e.g., from the used OSCORE library or from an OSCORE layer that takes part to the communication stack. The reasons that make CTX not safe to use anymore are the same ones discussed in Section 2.2 when considering a set of application keys in general, plus the event where the Access Token bound to CTX becomes invalid (e.g., it has expired or it has been revoked).

When this happens, the application at the peer P proceeds as follows.

1. If the following conditions both hold, then the application moves to step 2. Otherwise, it moves to step 3.

   * The Access Token is still valid. That is, it has not expired yet and the peer P is not aware that it has been revoked.

   * Let us define S as the EDHOC session from which the peer P has derived CTX or the eldest CTX's ancestor OSCORE Security Context. Then, since the completion of S with the other peer, the application at P has received from the other peer at least one message protected with any set of application keys derived from S. That is, P has persisted S (see Section 5.4.2 of [I-D.ietf-lake-edhoc]).

2. If the peer P supports the key update protocol KUDOS [I-D.ietf-core-oscore-key-update], then P runs KUDOS with the other peer, in order to update CTX. If the execution of KUDOS terminates successfully, the updated OSCORE Security Context is installed and no further actions are taken.

   If the execution of KUDOS does not terminate successfully or if the peer P does not support KUDOS altogether, then the application at P moves to step 3.

3. The application at the peer P performs the following actions.

   * If the Access Token is not valid anymore, the peer P deletes all the EDHOC sessions associated with the Access Token, as well as the OSCORE Security Context derived from each of those sessions.

     If the peer P acted as ACE Client, then P obtains a new Access Token from the ACE AS, and uploads it to the other peer acting as ACE RS.

     Finally, the application at P moves to step 4.

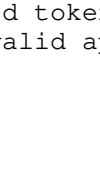
   * If the Access Token is valid while the OSCORE Security Context CTX is not, then the peer P deletes CTX.

     After that, the peer P deletes the EDHOC session from which CTX was generated, or from which the eldest CTX's ancestor OSCORE Security Context was generated before any key update occurred (e.g., by means of KUDOS or other key update methods).

     Finally, the application at P moves to step 4.

4. The peer P runs a new execution of the EDHOC protocol with the other peer. Upon successfully completing the EDHOC execution, the two peers derive and install a new OSCORE Security Context from this latest EDHOC session.

The flowchart in Figure 3 shows the handling of an Access Token or of a set of application keys that have become invalid.



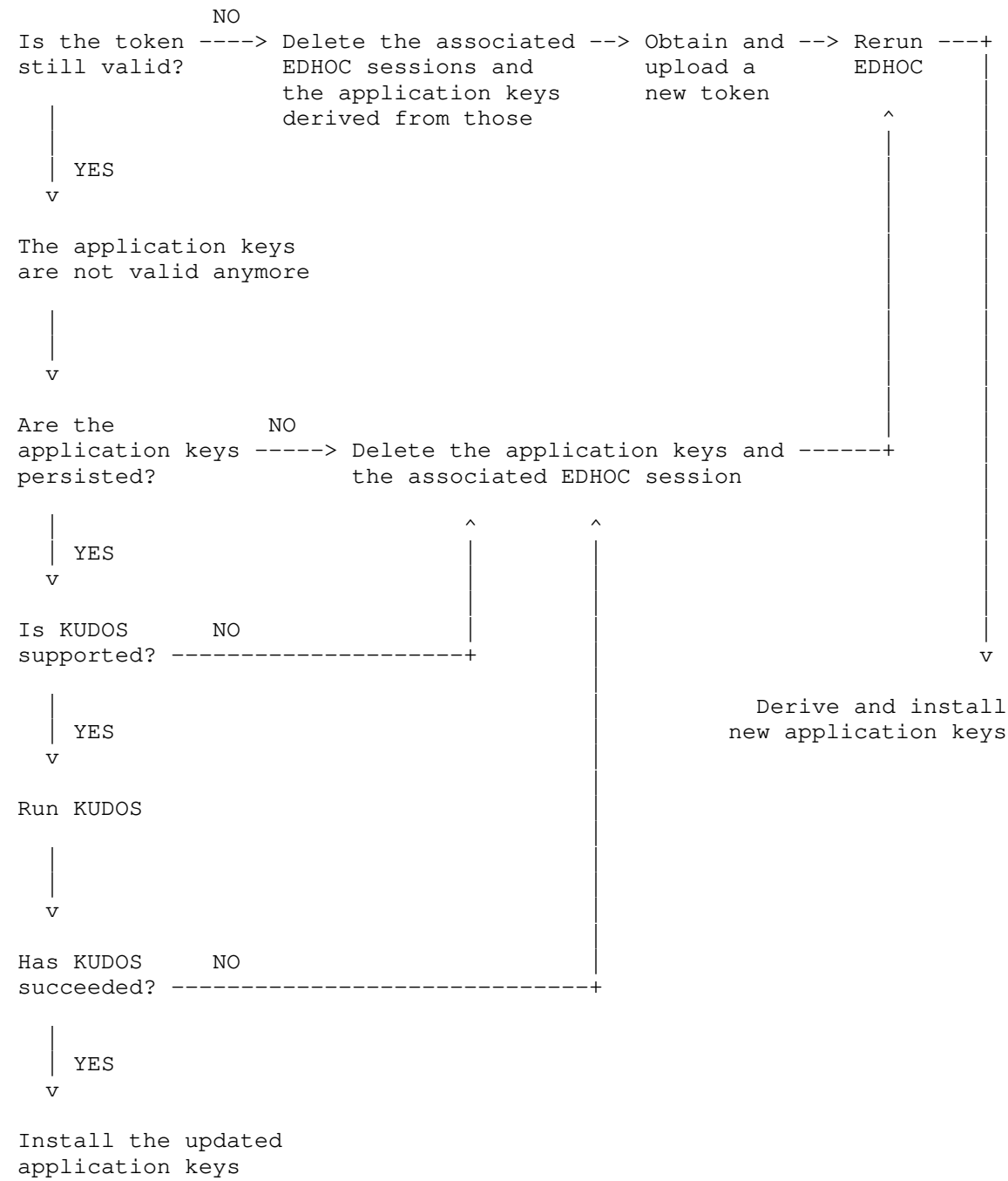

NO
Is the token ----> Delete the associated --> Obtain and --> Rerun ---+
still valid?   EDHOC sessions and   upload a   EDHOC |
                   the application keys   new token |
                   derived from those |
                                                                     |
YES |
v |
The application keys |
are not valid anymore |
                                                                     |
v |
Are the   NO |
application keys -----> Delete the application keys and ------+ |
persisted?   the associated EDHOC session | |
                                                              | |
YES | |
v | |
Is KUDOS   NO | |
supported? ---------------------+ | v
                                | |
YES | Derive and install
v | new application keys
Run KUDOS |
                                |
v |
Has KUDOS   NO |
succeeded? ---------------------+
YES
v
Install the updated
application keys

Figure 3: Handling of an Access Token or of a set of Application Keys that have Become Invalid

3. Trust Models for Learning New Authentication Credentials

A peer P relies on authentication credentials of other peers, in order to authenticate those peers when running EDHOC with them.

There are different ways for P to acquire an authentication credential CRED of another peer. For example, CRED can be supplied to P out-of-band by a trusted provider.

Alternatively, CRED can be specified by the other peer during the EDHOC execution with P. This relies on EDHOC message_2 or message_3, whose respective ID_CRED_R and ID_CRED_I can specify CRED by value, or instead a URI or other external reference where CRED can be retrieved from (see Section 3.5.3 of [I-D.ietf-lake-edhoc]).

Also during the EDHOC execution, an External Authorization Data (EAD) field might include an EAD item that specifies CRED by value or reference. This is the case, e.g., for the EAD item defined in [I-D.ietf-ace-edhoc-oscore-profile], which is used in the EAD_3 field of EDHOC message_3 and transports (a reference to) an Access Token that in turn specifies CRED_I by value or by reference.

When obtaining a new credential CRED, the peer P has to validate it before storing it. The validation steps to perform depend on the specific type of CRED (e.g., a public key certificate [RFC5280][I-D.ietf-cose-cbor-encoded-cert]), and can rely on (the authentication credential of) a trusted third party acting as a trust anchor.

Upon retrieving a new CRED through the processing of a received EDHOC message and following the successful validation of CRED, the peer P stores CRED only if it assesses CRED to be also trusted, and must not store CRED otherwise.

An exception applies for the two unauthenticated operations described in Appendix D.5 of [I-D.ietf-lake-edhoc], where a trust relationship with an unknown or not-yet-trusted endpoint is established later. That is, CRED is verified out-of-band at a later stage, or an EDHOC session key is bound to an identity out-of-band at a later stage.

If P stores CRED, then P will consider CRED as valid and trusted until it possibly becomes invalid, e.g., because it expires or because P gains knowledge that it has been revoked.

When storing CRED, the peer P should generate the authentication credential identifier(s) corresponding to CRED and store them as associated with CRED. For example, if CRED is a public key certificate, an identifier of CRED can be the hash of the certificate. In general, P should generate and associate with CRED one corresponding identifier for each type of authentication credential identifier that P supports and that is compatible with CRED.

In future executions of EDHOC with the other peer associated with CRED, this allows such other peer to specify CRED by reference, e.g., by indicating its credential identifier as ID_CRED_R/ID_CRED_I in the EDHOC message_2 or message_3 addressed to the peer P. In turn, this allows P to retrieve CRED from its local storage.

When processing a received EDHOC message M that specifies an authentication credential CRED, the peer P can enforce one of the following trust policies in order to determine whether to trust CRED.

* NO-LEARNING: according to this policy, the peer P trusts CRED if and only if P is already storing CRED at message reception time.

  That is, upon receiving M, the peer P can continue the execution of EDHOC only if both the following conditions hold.

  - P currently stores CRED, as specified by reference or by value in ID_CRED_I/ID_CRED_R or in the value of an EAD item; and

  - CRED is still valid, i.e., P believes CRED to not be expired or revoked.

* LEARNING: according to this policy, the peer P trusts CRED even if P is not already storing CRED at message reception time.

  That is, upon receiving M, the peer P performs the following steps.

  1. P retrieves CRED, as specified by reference or by value in ID_CRED_I/ID_CRED_R or in the value of an EAD item.

  2. P checks whether CRED is already being stored and if it is still valid. In such a case, P trusts CRED and can continue the EDHOC execution. Otherwise, P moves to step 3.

  3. P attempts to validate CRED. If the validation process is not successful, P aborts the EDHOC session with the other peer. Otherwise, P trusts and stores CRED, and can continue the EDHOC execution.

Irrespective of the adopted trust policy, P actually uses CRED only if it is determined to be fine to use in the context of the ongoing EDHOC session, also depending on the specific identity of the other peer (see Sections 3.5 and D.2 of [I-D.ietf-lake-edhoc]). If this is not the case, P aborts the EDHOC session with the other peer.

4. Side Processing of Incoming EDHOC Messages

This section describes an approach that EDHOC peers can use upon receiving EDHOC messages, in order to fetch/validate authentication credentials and to process External Authorization Data (EAD) items.

As per Section 9.1 of [I-D.ietf-lake-edhoc], the EDHOC protocol provides a transport mechanism for conveying EAD items, but specifications defining those items have to set the ground for "agreeing on the surrounding context and the meaning of the information passed to and from the application".

The approach described in this section aims to help implementors navigate the surrounding context mentioned above, irrespective of the specific EAD items conveyed in the EDHOC messages. In particular, the described approach takes into account the following points.

* The fetching and validation of the other peer's authentication credential relies on ID_CRED_I in EDHOC message_2, or on ID_CRED_R in EDHOC message_3, or on the value of an EAD item. When this occurs upon receiving EDHOC message_2 or message_3, the decryption of the EDHOC message has to be completed first.

  The validation of the other peer's authentication credential might depend on using the value of an EAD item, which in turn has to be validated first. For instance, an EAD item within the EAD_2 field may contain a voucher to be used for validating the other peer's authentication credential (see [I-D.ietf-lake-authz]).

* Some EAD items may be processed only after having successfully verified the EDHOC message, i.e., after a successful verification of the Signature_or_MAC field.

  For instance, an EAD item within the EAD_3 field may contain a Certificate Signing Request (CSR) [RFC2986]. Hence, such an EAD item can be processed only once the recipient peer has attained proof of the other peer possessing its private key.

In order to conveniently handle such processing, the application can prepare in advance one "side-processor object" (SPO), which takes care of the operations above during the EDHOC execution.

In particular, the application provides EDHOC with the SPO before starting an EDHOC execution, during which EDHOC will temporarily transfer control to the SPO at the right point in time, in order to perform the required side-processing of an incoming EDHOC message.

Furthermore, the application has to instruct the SPO about how to prepare any EAD item such that: it has to be included in an outgoing EDHOC message; and it is independent of the processing of other EAD items included in incoming EDHOC messages. This includes, for instance, the preparation of padding EAD items.

At the right point in time during the processing of an incoming EDHOC message M at the peer P, EDHOC invokes the SPO and provides it with the following input:

* When M is EDHOC message_2 or message_3, an indication of whether this invocation is happening before or after the message verification (i.e., before or after having verified the Signature_or_MAC field).

* The full set of information related to the current EDHOC session. This especially includes the selected cipher suite and the ephemeral Diffie-Hellman public keys G_X and G_Y that the two peers have exchanged in the EDHOC session.

* The other peers' authentication credentials that the peer P stores.

* All the decrypted information elements retrieved from M.

* The EAD items included in M.

  - Note that EDHOC might do some preliminary work on M before invoking the SPO, in order to provide the SPO only with actually relevant EAD items. This requires the application to additionally provide EDHOC with the ead_labels of the EAD items that the peer P recognizes (see Section 3.8 of [I-D.ietf-lake-edhoc]).

    With such information available, EDHOC can early abort the current session in case M includes any EAD item which is both critical and not recognized by the peer P.

If no such EAD items are found, EDHOC can remove any padding EAD item (see Section 3.8.1 of [I-D.ietf-lake-edhoc]), as well as any EAD item which is neither critical nor recognized (since the SPO is going to ignore it anyway). As a result, EDHOC is able to provide the SPO only with EAD items that will be recognized and that require actual processing.

- Note that, after having processed the EAD items, the SPO might actually need to store them throughout the whole EDHOC execution, e.g., in order to refer to them also when processing later EDHOC messages in the current EDHOC session.

The SPO performs the following tasks on an incoming EDHOC message M.

* The SPO fetches and/or validates the other peer's authentication credential CRED, based on a dedicated EAD item of M or on the ID_CRED field of M (for EDHOC message_2 or message_3).

* The SPO processes the EAD items conveyed in the EAD field of M.

* The SPO stores the results of the performed operations, and makes such results available to the application.

* When the SPO has completed its side processing and transfers control back to EDHOC, the SPO provides EDHOC with the produced EAD items to include in the EAD field of the next outgoing EDHOC message. The production of such EAD items can be triggered, e.g., by:

  - The consumption of EAD items included in M; and

  - The execution of instructions that the SPO has received from the application, concerning EAD items to produce irrespective of other EAD items included in M.

The following subsections describe more in detail the actions performed by the SPO on the different, incoming EDHOC messages.

4.1. EDHOC message_1

During the processing of an incoming EDHOC message_1, EDHOC invokes the SPO only once, after the Responder peer has successfully decoded the message and accepted the selected cipher suite.

If the EAD_1 field is present, the SPO processes the EAD items included therein.

Once all such EAD items have been processed the SPO transfers control back to EDHOC. When doing so, the SPO also provides EDHOC with any produced EAD items to include in the EAD field of the next outgoing EDHOC message.

Then, EDHOC resumes its execution and advances its protocol state.

4.2. EDHOC message_4

During the processing of an incoming EDHOC message_4, EDHOC invokes the SPO only once, after the Initiator peer has successfully decrypted the message.

If the EAD_4 field is present, the SPO processes the EAD items included therein.

Once all such EAD items have been processed, the SPO transfers control back to EDHOC, which resumes its execution and advances its protocol state.

4.3. EDHOC message_2 and message_3

The following refers to "message_X" as an incoming EDHOC message_2 or message_3, and to "message verification" as the verification of Signature_or_MAC_X in message_X.

During the processing of a message_X, EDHOC invokes the SPO two times:

* Right after message_X has been decrypted and before its verification starts. Following this invocation, the SPO performs the actions described in Section 4.3.1.

* Right after message_X has been successfully verified. Following this invocation, the SPO performs the actions described in Section 4.3.2.

The flowcharts in Section 4.3.3 show the high-level interaction between the core EDHOC processing and the SPO, as well as the different steps taken for processing an incoming message_X.

4.3.1. Pre-Verification Side Processing

The pre-verification side processing occurs in two sequential phases, namely PHASE_1 and PHASE_2.

PHASE_1 - During PHASE_1, the SPO at the recipient peer P determines CRED, i.e., the other peer's authentication credential to use in the ongoing EDHOC session. In particular, the SPO performs the following steps.

1. The SPO determines CRED based on ID_CRED_X or on an EAD item in message_X.

   Those may specify CRED by value or by reference, including a URI or other external reference where CRED can be retrieved from.

   If CRED is already installed, the SPO moves to step 2. Otherwise, the SPO moves to step 3.

2. The SPO determines if the stored CRED is currently valid, e.g., by asserting that CRED has not expired and has not been revoked.

   Performing such a validation may require the SPO to first process an EAD item included in message_X. For example, it can be an EAD item in EDHOC message_2, which confirms or revokes the validity of CRED_R specified by ID_CRED_R, as the result of an OCSP process [RFC6960].

   In case CRED is determined to be valid, the SPO moves to step 9. Otherwise, the SPO moves to step 11.

3. The SPO attempts to retrieve CRED, and then moves to step 4.

4. If the retrieval of CRED has succeeded, the SPO moves to step 5. Otherwise, the SPO moves to step 11.

5. If the enforced trust policy for new authentication credentials is "NO-LEARNING" (see Section 3), the SPO moves to step 11. Otherwise, the SPO moves to step 6.

6. If this step has been reached, the peer P enforces the trust policy "LEARNING" (see Section 3) and it is not already storing the retrieved CRED.

   Consistently, the SPO determines if CRED is currently valid, e.g., by asserting that CRED has not expired and has not been revoked. Then, the SPO moves to step 7.

Validating CRED may require the SPO to first process an EAD item included in message_X. For example, it can be an EAD item in EDHOC message_2 that: i) specifies a voucher for validating CRED_R as a CWT Claims Set (CCS) [RFC8392] transported by value in ID_CRED_R (see [I-D.ietf-lake-authz]); or instead ii) an OCSP response [RFC6960] for validating CRED_R as a certificate transported by value or reference in ID_CRED_R.

7. If CRED has been determined valid, the SPO moves to step 8. Otherwise, the SPO moves to step 11.

8. The SPO stores CRED as a valid and trusted authentication credential associated with the other peer, together with corresponding authentication credential identifiers (see Section 3). Then, the SPO moves to step 9.

9. The SPO checks if CRED is fine to use in the context of the ongoing EDHOC session, also depending on the specific identity of the other peer (see Sections 3.5 and D.2 of [I-D.ietf-lake-edhoc]).

   If this is the case, the SPO moves to step 10. Otherwise, the SPO moves to step 11.

10. P uses CRED as authentication credential of the other peer in the ongoing EDHOC session.

    Then, PHASE_1 ends, and the pre-verification side processing moves to the next PHASE_2 (see below).

11. The SPO has not found a valid authentication credential associated with the other peer that can be used in the ongoing EDHOC session. Therefore, the EDHOC session with the other peer is aborted.

PHASE_2 - During PHASE_2, the SPO processes any EAD item included in message_X such that both the following conditions hold.

* The EAD item has _not_ been already processed during PHASE_1.

* The EAD item can be processed before performing the verification of message_X.

Once all such EAD items have been processed, the SPO transfers control back to EDHOC, which either aborts the ongoing EDHOC session or continues the processing of message_X with its corresponding message verification.

4.3.2. Post-Verification Side Processing

During the post-verification side processing, the SPO processes any EAD item included in message_X such that the processing of that EAD item had to wait for completing the successful message verification.

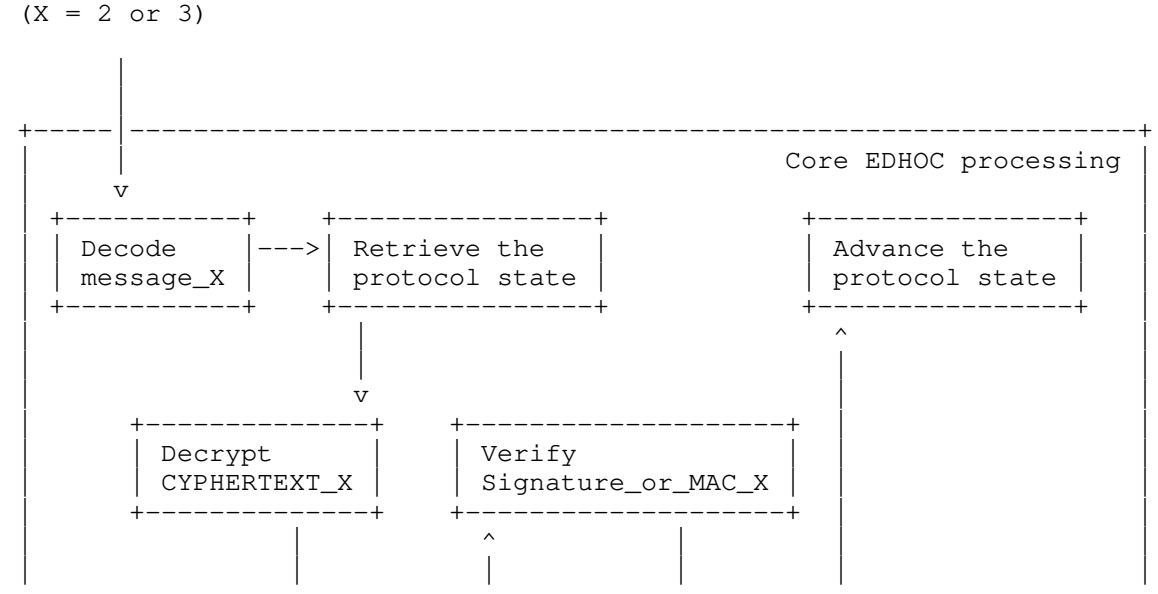
The late processing of such EAD items is typically due to the fact that a pre-requirement has to be fulfilled first. For example, the recipient peer P has to have first asserted that the other peer does possess the private key corresponding to the public key of the other peer's authentication credential CRED determined during the pre-verification side processing (see Section 4.3.1). This requirement is fulfilled after a successful message verification of message_X.

Once all such EAD items have been processed, the SPO transfers control back to EDHOC. When doing so, the SPO also provides EDHOC with any produced EAD items to include in the EAD field of the next outgoing EDHOC message.

Then, EDHOC resumes its execution and advances its protocol state.

4.3.3. Flowcharts

The flowchart in Figure 4 shows the high-level interaction between the core EDHOC processing and the SPO, with particular reference to an incoming EDHOC message_2 or message_3.



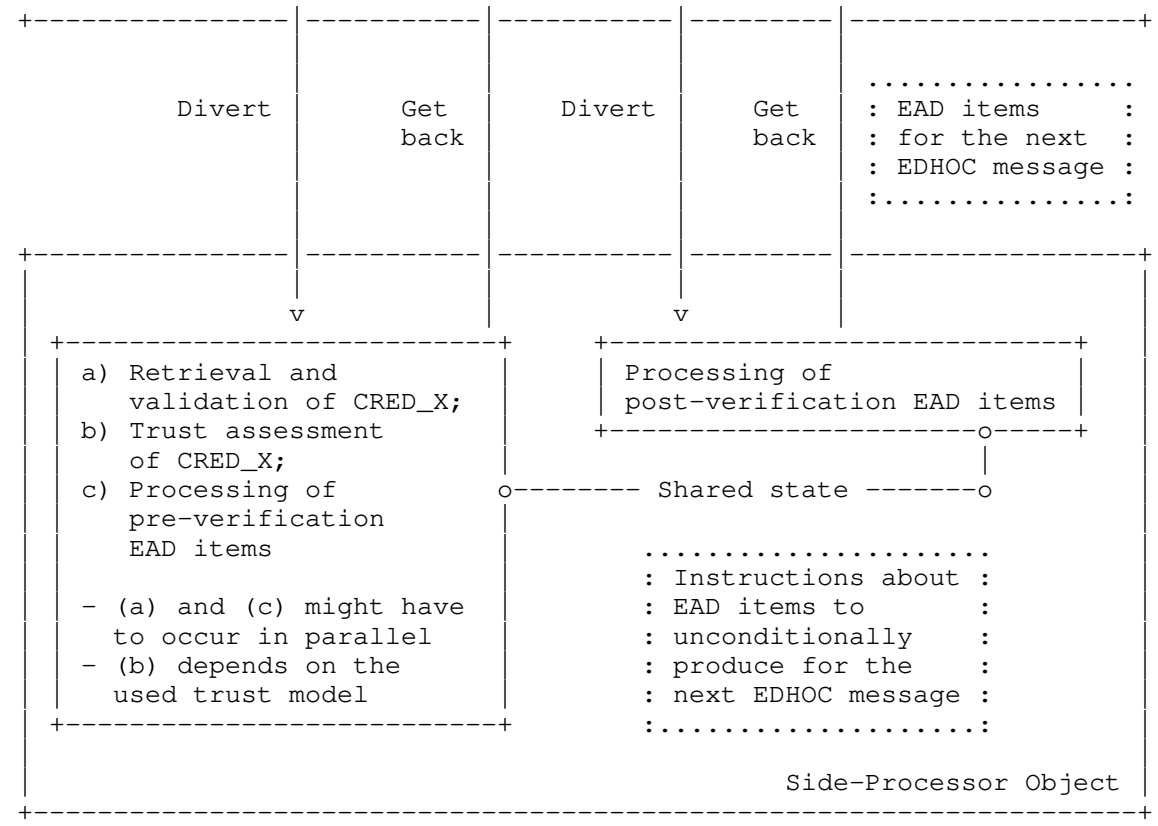
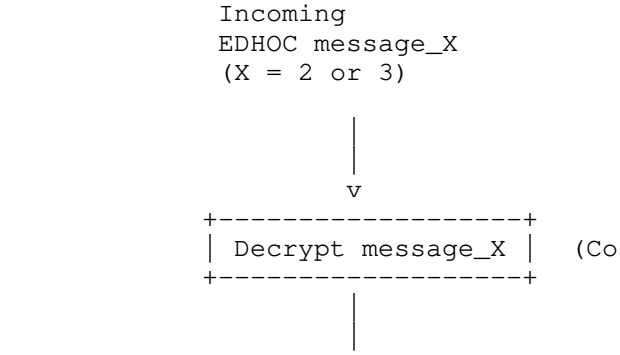
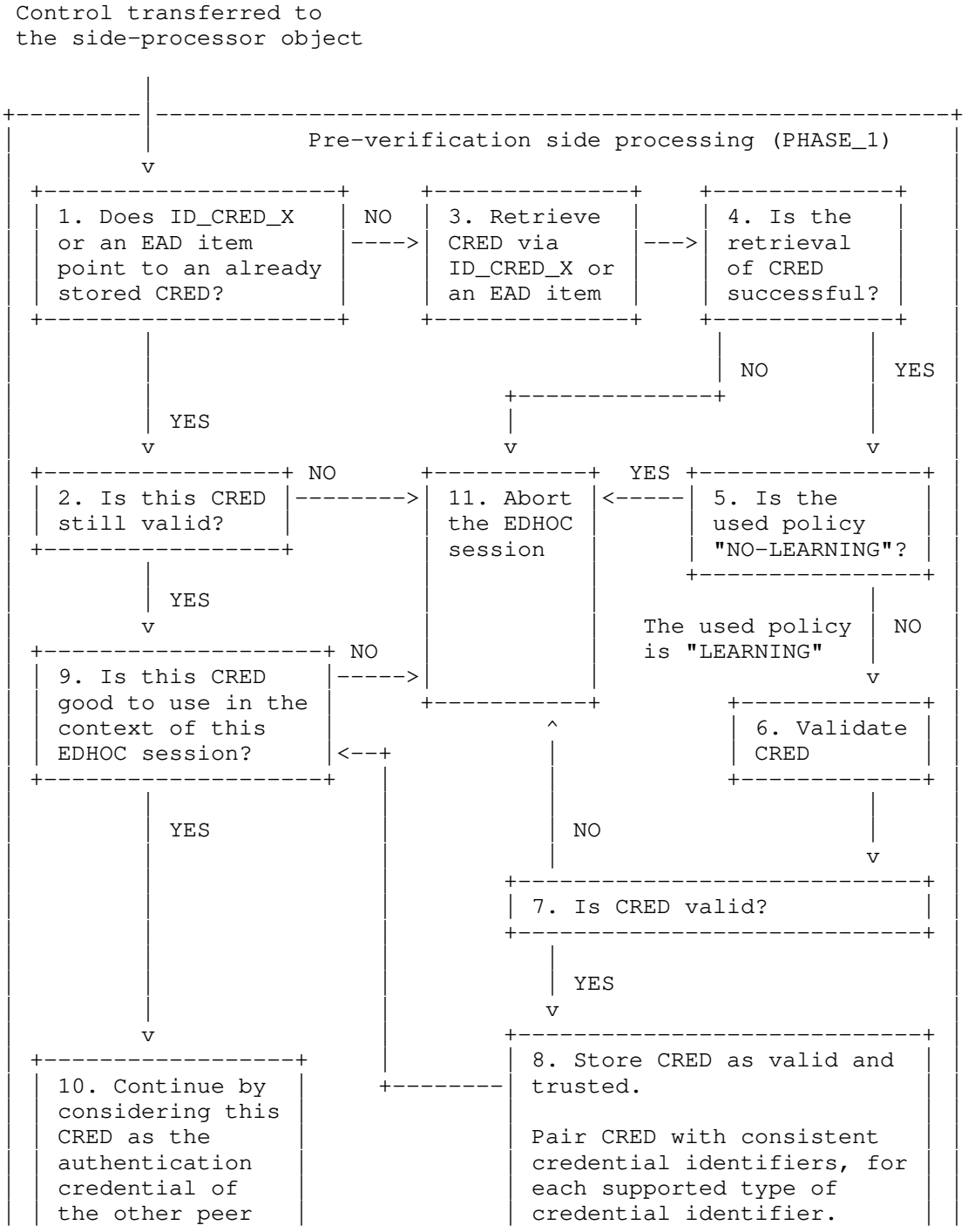
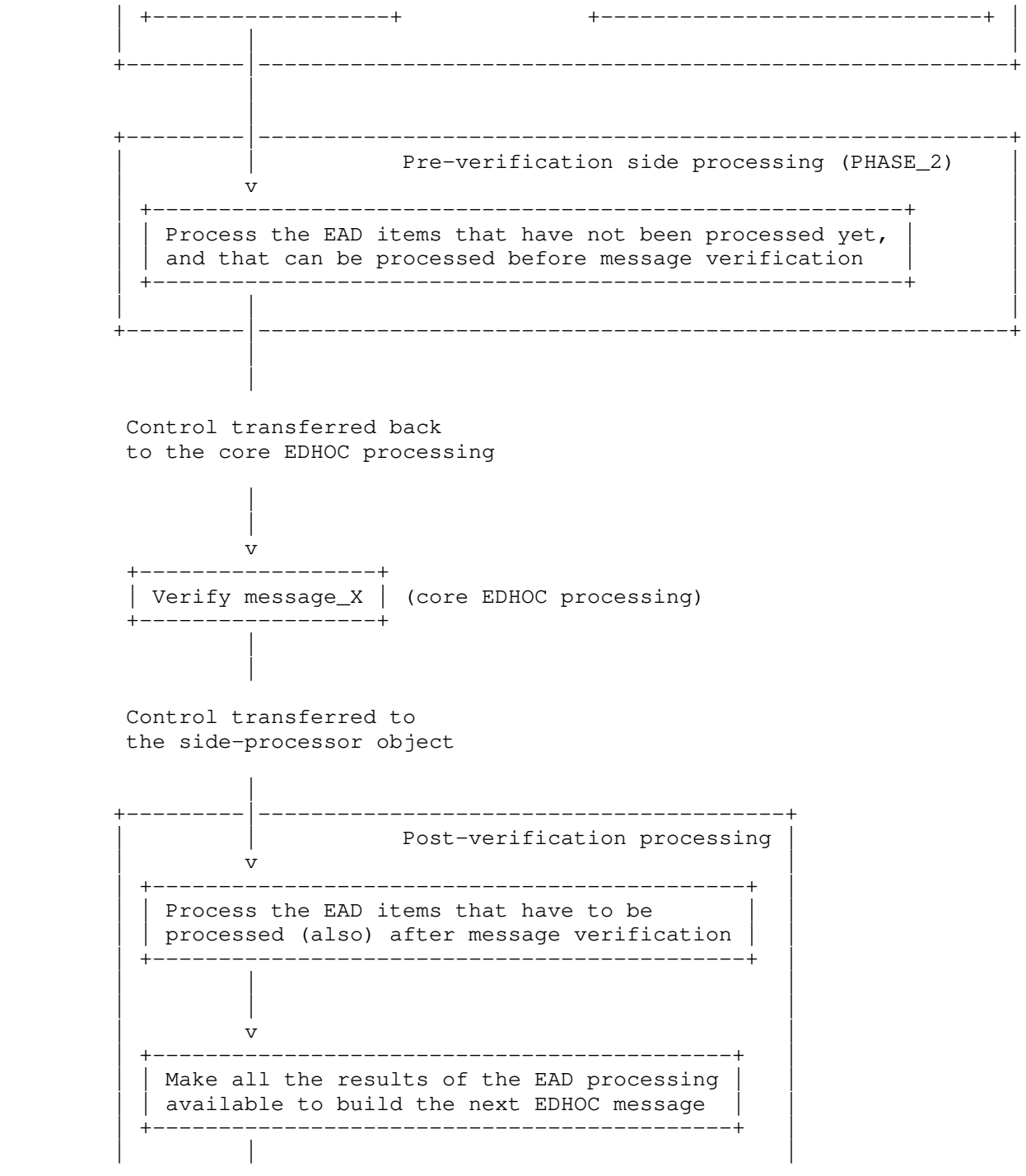


Figure 4: High-Level Interaction Between the Core EDHOC Processing and the Side-Processor Object (SPO)

The flowchart in Figure 5 shows the different steps taken for processing an incoming EDHOC message_2 and message_3.




Control transferred to
the side-processor object
Pre-verification side processing (PHASE_1)
1. Does ID_CRED_X or an EAD item point to an already stored CRED?
NO
3. Retrieve CRED via ID_CRED_X or an EAD item
4. Is the retrieval of CRED successful?
NO
YES
YES
2. Is this CRED still valid?
NO
11. Abort the EDHOC session
YES
5. Is the used policy "NO-LEARNING"?
YES
The used policy is "LEARNING"
NO
9. Is this CRED good to use in the context of this EDHOC session?
NO
6. Validate CRED
YES
NO
7. Is CRED valid?
YES
8. Store CRED as valid and trusted.
Pair CRED with consistent credential identifiers, for each supported type of credential identifier.
10. Continue by considering this CRED as the authentication credential of the other peer


Pre-verification side processing (PHASE_2)
Process the EAD items that have not been processed yet,
and that can be processed before message verification
Control transferred back
to the core EDHOC processing
Verify message_X
(core EDHOC processing)
Control transferred to
the side-processor object
Post-verification processing
Process the EAD items that have to be
processed (also) after message verification
Make all the results of the EAD processing
available to build the next EDHOC message

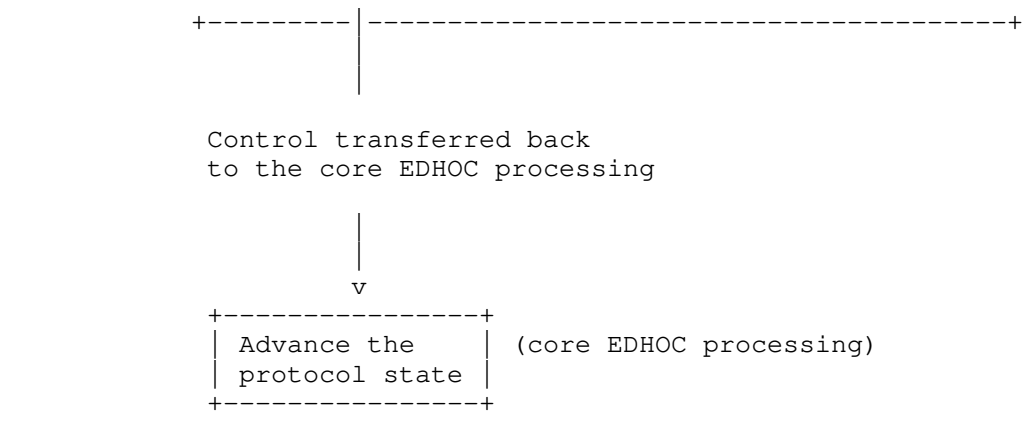


Figure 5: Processing steps for EDHOC message_2 and message_3

5. Security Considerations

TBD

6. IANA Considerations

This document has no actions for IANA.

7. References

7.1. Normative References

[I-D.ietf-lake-edhoc]
Selander, G., Mattsson, J. P., and F. Palombini, "Ephemeral Diffie-Hellman Over COSE (EDHOC)", Work in Progress, Internet-Draft, draft-ietf-lake-edhoc-23, 22 January 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-lake-edhoc-23>.

[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/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8613] Selander, G., Mattsson, J., Palombini, F., and L. Seitz, "Object Security for Constrained RESTful Environments (OSCORE)", RFC 8613, DOI 10.17487/RFC8613, July 2019, <https://www.rfc-editor.org/rfc/rfc8613>.

7.2. Informative References

[I-D.ietf-ace-edhoc-oscore-profile]
Selander, G., Mattsson, J. P., Tiloca, M., and R. Höglund, "Ephemeral Diffie-Hellman Over COSE (EDHOC) and Object Security for Constrained Environments (OSCORE) Profile for Authentication and Authorization for Constrained Environments (ACE)", Work in Progress, Internet-Draft, draft-ietf-ace-edhoc-oscore-profile-03, 23 October 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-ace-edhoc-oscore-profile-03>.

[I-D.ietf-core-oscore-key-limits]
Höglund, R. and M. Tiloca, "Key Usage Limits for OSCORE", Work in Progress, Internet-Draft, draft-ietf-core-oscore-key-limits-02, 10 January 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-core-oscore-key-limits-02>.

[I-D.ietf-core-oscore-key-update]
Höglund, R. and M. Tiloca, "Key Update for OSCORE (KUDOS)", Work in Progress, Internet-Draft, draft-ietf-core-oscore-key-update-06, 23 October 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-core-oscore-key-update-06>.

[I-D.ietf-cose-cbor-encoded-cert]
Mattsson, J. P., Selander, G., Raza, S., Höglund, J., and M. Furuhed, "CBOR Encoded X.509 Certificates (C509 Certificates)", Work in Progress, Internet-Draft, draft-ietf-cose-cbor-encoded-cert-07, 20 October 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-cose-cbor-encoded-cert-07>.

[I-D.ietf-lake-authz]
Selander, G., Mattsson, J. P., Vuini, M., and M. Richardson, "Lightweight Authorization using Ephemeral Diffie-Hellman Over COSE", Work in Progress, Internet-Draft, draft-ietf-lake-authz-00, 23 October 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-lake-authz-00>.

[RFC2986] Nystrom, M. and B. Kaliski, "PKCS #10: Certification Request Syntax Specification Version 1.7", RFC 2986, DOI 10.17487/RFC2986, November 2000, <https://www.rfc-editor.org/rfc/rfc2986>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/rfc/rfc5280>.

[RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <https://www.rfc-editor.org/rfc/rfc6960>.

[RFC8392] Jones, M., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "CBOR Web Token (CWT)", RFC 8392, DOI 10.17487/RFC8392, May 2018, <https://www.rfc-editor.org/rfc/rfc8392>.

[RFC9200] Seitz, L., Selander, G., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "Authentication and Authorization for Constrained Environments Using the OAuth 2.0 Framework (ACE-OAuth)", RFC 9200, DOI 10.17487/RFC9200, August 2022, <https://www.rfc-editor.org/rfc/rfc9200>.

Acknowledgments

The author sincerely thanks Christian Amsüss, Geovane Fedrecheski, Rikard Höglund, John Preuß Mattsson, Göran Selander, and Malia Vuini for their comments and feedback.

Author's Address

Marco Tiloca
RISE AB
Isafjordsgatan 22
SE-16440 Stockholm Kista
Sweden
Email: marco.tiloca@ri.se