Workgroup: CoRE Working Group
Internet-Draft:
draft-ietf-core-oscore-key-update-02
U 8613 (if approved)
p
d
a
t
e
s
:
Published: 11 July 2022
Intended Status: Standards Track
Expires: 12 January 2023
A R. Höglund M. Tiloca
uRISE AB RISE AB
t
h
o
r
s
:

Key Update for OSCORE (KUDOS)

Abstract

Object Security for Constrained RESTful Environments (OSCORE) uses AEAD algorithms to ensure confidentiality and integrity of exchanged messages. Due to known issues allowing forgery attacks against AEAD algorithms, limits should be followed on the number of times a specific key is used for encryption or decryption. Among other reasons, approaching key usage limits requires updating the OSCORE keying material before communications can securely continue.

This document defines how two OSCORE peers must follow these key usage limits and what steps they must take to preserve the security of their communications. Also, it specifies Key Update for OSCORE (KUDOS), a lightweight procedure that two peers can use to update their keying material and establish a new OSCORE Security Context. Finally, this document specifies a method that two peers can use to update their OSCORE identifiers, as a stand-alone procedure or embedded in a KUDOS execution. Thus, this document updates RFC 8613.

Discussion Venues

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

Discussion of this document takes place on the Constrained RESTful Environments Working Group mailing list (core@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/core/.

Source for this draft and an issue tracker can be found at https://github.com/core-wg/oscore-key-update.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction
  1.1. Terminology
2. AEAD Key Usage Limits in OSCORE
  2.1. Problem Overview
    2.1.1. Limits for 'q' and 'v'
  2.2. Additional Information in the Security Context
    2.2.1. Common Context
    2.2.2. Sender Context
    2.2.3. Recipient Context
  2.3. OSCORE Messages Processing
    2.3.1. Protecting a Request or a Response
    2.3.2. Verifying a Request or a Response
3. Current methods for Rekeying OSCORE
4. Key Update for OSCORE (KUDOS)
  4.1. Extensions to the OSCORE Option
  4.2. Function for Security Context Update
  4.3. Key Update with Forward Secrecy
    4.3.1. Client-Initiated Key Update
    4.3.2. Server-Initiated Key Update
  4.4. Key Update with or without Forward Secrecy
    4.4.1. Handling and Use of Keying Material
    4.4.2. Selection of KUDOS Mode
  4.5. Preserving Observations across Key Updates
    4.5.1. Management of Observations

4.6. Retention Policies
4.7. Discussion
5. Update of OSCORE Sender/Recipient IDs
5.1. The Recipient-ID Option
5.1.1. Client-Initiated OSCORE IDs Update
5.1.2. Server-Initiated OSCORE IDs Update
5.1.3. Additional Actions for Stand-Alone Execution
6. Security Considerations
7. IANA Considerations
7.1. CoAP Option Numbers Registry
7.2. OSCORE Flag Bits Registry
8. References
8.1. Normative References
8.2. Informative References
Appendix A. Detailed considerations for AEAD_AES_128_CCM_8
Appendix B. Estimation of 'count_q'
Appendix C. Document Updates
C.1. Version -01 to -02
C.2. Version -00 to -01
Acknowledgments
Authors' Addresses

1. Introduction

Object Security for Constrained RESTful Environments (OSCORE) [RFC8613] provides end-to-end protection of CoAP [RFC7252] messages at the application-layer, ensuring message confidentiality and integrity, replay protection, as well as binding of response to request between a sender and a recipient.

In particular, OSCORE uses AEAD algorithms to provide confidentiality and integrity of messages exchanged between two peers. Due to known issues allowing forgery attacks against AEAD algorithms, limits should be followed on the number of times a specific key is used to perform encryption or decryption [I-D.irtf-cfrg-aead-limits].

The original OSCORE specification [RFC8613] does not consider such key usage limits. However, should they be exceeded, an adversary may break the security properties of the AEAD algorithm, such as message confidentiality and integrity, e.g., by performing a message forgery attack. Among other reasons, approaching the key usage limits requires updating the OSCORE keying material before communications can securely continue.

This document updates [RFC8613] as follows.

* It defines what steps an OSCORE peer takes to preserve the security of its communications, by stopping using the OSCORE Security Context shared with another peer when approaching the key usage limits.

* It specifies KUDOS, a lightweight key update procedure that the two peers can use in order to update their current keying material and establish a new OSCORE Security Context. This deprecates and replaces the procedure specified in Appendix B.2 of [RFC8613].

* It specifies a method that two peers can use to update their OSCORE identifiers. This can be run as a stand-alone procedure, or instead embedded in a KUDOS execution.

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 be familiar with the terms and concepts related to the CoAP [RFC7252], Observe [RFC7641], CBOR [RFC8949], OSCORE [RFC8613] and EDHOC [I-D.ietf-lake-edhoc].

This document additionally defines the following terminology.

* Initiator: the peer starting the KUDOS execution, by sending the first KUDOS message.

* Responder: the peer that receives the first KUDOS message in a KUDOS execution.

* FS mode: the KUDOS execution mode that achieves forward secrecy (see Section 4.3).

* No-FS mode: the KUDOS execution mode that does not achieve forward secrecy (see Section 4.4).

2. AEAD Key Usage Limits in OSCORE

This section details how key usage limits for AEAD algorithms must be considered when using OSCORE. In particular, it discusses specific limits for common AEAD algorithms used with OSCORE; necessary additions to the OSCORE Security Context; and updates to the OSCORE message processing.

2.1. Problem Overview

The OSCORE security protocol [RFC8613] uses AEAD algorithms to provide integrity and confidentiality of messages, as exchanged between two peers sharing an OSCORE Security Context.

When processing messages with OSCORE, each peer should follow specific limits as to the number of times it uses a specific key. This applies separately to the Sender Key used to encrypt outgoing messages, and to the Recipient Key used to decrypt and verify incoming protected messages.

Exceeding these limits may allow an adversary to break the security properties of the AEAD algorithm, such as message confidentiality and integrity, e.g., by performing a message forgery attack.

The following refers to the two parameters 'q' and 'v' introduced in [I-D.irtf-cfrg-aead-limits], to use when deploying an AEAD algorithm.

- 'q': this parameter has as value the number of messages protected with a specific key, i.e., the number of times the AEAD algorithm has been invoked to encrypt data with that key.

- 'v': this parameter has as value the number of alleged forgery attempts that have been made against a specific key, i.e., the amount of failed decryptions that have occurred with the AEAD algorithm for that key.

When a peer uses OSCORE:

- The key used to protect outgoing messages is its Sender Key from its Sender Context.

- The key used to decrypt and verify incoming messages is its Recipient Key from its Recipient Context.

Both keys are derived as part of the establishment of the OSCORE Security Context, as defined in Section 3.2 of [RFC8613].

As mentioned above, exceeding specific limits for the 'q' or 'v' value can weaken the security properties of the AEAD algorithm used, thus compromising secure communication requirements.

Therefore, in order to preserve the security of the used AEAD algorithm, OSCORE has to observe limits for the 'q' and 'v' values, throughout the lifetime of the used AEAD keys.

2.1.1. Limits for 'q' and 'v'

Formulas for calculating the security levels, as Integrity Advantage (IA) and Confidentiality Advantage (CA) probabilities, are presented in [I-D.irtf-cfrg-aead-limits]. These formulas take as input specific values for 'q' and 'v' (see section Section 2.1) and for 'l', i.e., the maximum length of each message (in cipher blocks).

For the algorithms shown in Figure 1 that can be used as AEAD Algorithm for OSCORE, the key property to achieve is having IA and CA values which are no larger than p = 2^-64, which will ensure a safe security level for the AEAD Algorithm. This can be entailed by using the values q = 2^20, v = 2^20, and l = 2^10, that this document recommends to use for these algorithms.

Figure 1 also shows the resulting IA and CA probabilities enjoyed by the considered algorithms, when taking the value of 'q', 'v' and 'l' above as input to the formulas defined in [I-D.irtf-cfrg-aead-limits].

Algorithm name	IA probability	CA probability
AEAD_AES_128_CCM	2^-64	2^-66
AEAD_AES_128_GCM	2^-97	2^-89
AEAD_AES_256_GCM	2^-97	2^-89
AEAD_CHACHA20_POLY1305	2^-73	-

Figure 1: Probabilities for algorithms based on chosen q, v and l values.

When AEAD_AES_128_CCM_8 is used as AEAD Algorithm for OSCORE, the triplet (q, v, l) considered above yields larger values of IA and CA. Hence, specifically for AEAD_AES_128_CCM_8, this document recommends using the triplet (q, v, l) = (2^20, 2^14, 2^8). This is appropriate, since the resulting CA and IA values are not greater than the threshold value of 2^-50 defined in [I-D.irtf-cfrg-aead-limits], and thus yields an acceptable security level. Achieving smaller values of CA and IA would require to inconveniently reduce 'q', 'v' or 'l', with no corresponding increase in terms of security, as further elaborated in Appendix A.

Algorithm name	l=2^6 in bytes	l=2^8 in bytes	l=2^10 in bytes
AEAD_AES_128_CCM	1024	4096	16384
AEAD_AES_128_GCM	1024	4096	16384
AEAD_AES_256_GCM	1024	4096	16384
AEAD_AES_128_CCM_8	1024	4096	16384
AEAD_CHACHA20_POLY1305	4096	16384	65536

Figure 2: Maximum length of each message (in bytes)

2.2. Additional Information in the Security Context

In addition to what defined in Section 3.1 of [RFC8613], the OSCORE Security Context MUST also include the following information.

2.2.1. Common Context

The Common Context is extended to include the following parameter.

* 'exp': with value the expiration time of the OSCORE Security Context, as a non-negative integer. The parameter contains a numeric value representing the number of seconds from 1970-01-01T00:00:00Z UTC until the specified UTC date/time, ignoring leap seconds, analogous to what specified for NumericDate in Section 2 of [RFC7519].

  At the time indicated in this field, a peer MUST stop using this Security Context to process any incoming or outgoing message, and is required to establish a new Security Context to continue OSCORE-protected communications with the other peer.

2.2.2. Sender Context

The Sender Context is extended to include the following parameters.

* 'count_q': a non-negative integer counter, keeping track of the current 'q' value for the Sender Key. At any time, 'count_q' has as value the number of messages that have been encrypted using the Sender Key. The value of 'count_q' is set to 0 when establishing the Sender Context.

* 'limit_q': a non-negative integer, which specifies the highest value that 'count_q' is allowed to reach, before stopping using the Sender Key to process outgoing messages.

  The value of 'limit_q' depends on the AEAD algorithm specified in the Common Context, considering the properties of that algorithm. The value of 'limit_q' is determined according to Section 2.1.1.

Note for implementation: it is possible to avoid storing and maintaining the counter 'count_q'. Rather, an estimated value to be compared against 'limit_q' can be computed, by leveraging the Sender Sequence Number of the peer and (an estimate of) the other peer's. A possible method to achieve this is described in Appendix B. While this relieves peers from storing and maintaining the precise 'count_q' value, it results in overestimating the number of encryptions performed with a Sender Key. This in turn results in approaching 'limit_q' sooner and thus in performing a key update procedure more frequently.

2.2.3. Recipient Context

The Recipient Context is extended to include the following parameters.

* 'count_v': a non-negative integer counter, keeping track of the current 'v' value for the Recipient Key. At any time, 'count_v' has as value the number of failed decryptions occurred on incoming messages using the Recipient Key. The value of 'count_v' is set to 0 when establishing the Recipient Context.

* 'limit_v': a non-negative integer, which specifies the highest value that 'count_v' is allowed to reach, before stopping using the Recipient Key to process incoming messages.

  The value of 'limit_v' depends on the AEAD algorithm specified in the Common Context, considering the properties of that algorithm. The value of 'limit_v' is determined according to Section 2.1.1.

2.3. OSCORE Messages Processing

In order to keep track of the 'q' and 'v' values and ensure that AEAD keys are not used beyond reaching their limits, the processing of OSCORE messages is extended as defined in this section. A limitation that is introduced is that, in order to not exceed the selected value for 'l', the total size of the COSE plaintext, authentication Tag, and possible cipher padding for a message may not exceed the block size for the selected algorithm multiplied with 'l'.

In particular, the processing of OSCORE messages follows the steps outlined in Section 8 of [RFC8613], with the additions defined below.

2.3.1. Protecting a Request or a Response

Before encrypting the COSE object using the Sender Key, the 'count_q' counter MUST be incremented.

If 'count_q' exceeds the 'limit_q' limit, the message processing MUST be aborted. From then on, the Sender Key MUST NOT be used to encrypt further messages.

2.3.2. Verifying a Request or a Response

If an incoming message is detected to be a replay (see Section 7.4 of [RFC8613]), the 'count_v' counter MUST NOT be incremented.

If the decryption and verification of the COSE object using the Recipient Key fails, the 'count_v' counter MUST be incremented.

After 'count_v' has exceeded the 'limit_v' limit, incoming messages MUST NOT be decrypted and verified using the Recipient Key, and their processing MUST be aborted.

3. Current methods for Rekeying OSCORE

Before the limit of 'q' or 'v' defined in Section 2.1.1 has been reached for an OSCORE Security Context, the two peers have to establish a new OSCORE Security Context, in order to continue using OSCORE for secure communication.

In practice, the two peers have to establish new Sender and Recipient Keys, as the keys actually used by the AEAD algorithm. When this happens, both peers reset their 'count_q' and 'count_v' values to 0 (see Section 2.2).

Other specifications define a number of ways to accomplish this, as summarized below.

* The two peers can run the procedure defined in Appendix B.2 of [RFC8613]. That is, the two peers exchange three or four messages, protected with temporary Security Contexts adding randomness to the ID Context.

  As a result, the two peers establish a new OSCORE Security Context with new ID Context, Sender Key and Recipient Key, while keeping the same OSCORE Master Secret and OSCORE Master Salt from the old OSCORE Security Context.

  This procedure does not require any additional components to what OSCORE already provides, and it does not provide forward secrecy.

  The procedure defined in Appendix B.2 of [RFC8613] is used in 6TiSCH networks [RFC7554][RFC8180] when handling failure events. That is, a node acting as Join Registrar/Coordinator (JRC) assists new devices, namely "pledges", to securely join the network as per the Constrained Join Protocol [RFC9031]. In

particular, a pledge exchanges OSCORE-protected messages with the JRC, from which it obtains a short identifier, link-layer keying material and other configuration parameters. As per Section 8.3.3 of [RFC9031], a JRC that experiences a failure event may likely lose information about joined nodes, including their assigned identifiers. Then, the reinitialized JRC can establish a new OSCORE Security Context with each pledge, through the procedure defined in Appendix B.2 of [RFC8613].

* The two peers can run the OSCORE profile [I-D.ietf-ace-oscore-profile] of the Authentication and Authorization for Constrained Environments (ACE) Framework [I-D.ietf-ace-oauth-authz].

  When a CoAP client uploads an Access Token to a CoAP server as an access credential, the two peers also exchange two nonces. Then, the two peers use the two nonces together with information provided by the ACE Authorization Server that issued the Access Token, in order to derive an OSCORE Security Context.

  This procedure does not provide forward secrecy.

* The two peers can run the EDHOC key exchange protocol based on Diffie-Hellman and defined in [I-D.ietf-lake-edhoc], in order to establish a pseudo-random key in a mutually authenticated way.

  Then, the two peers can use the established pseudo-random key to derive external application keys. This allows the two peers to securely derive an OSCORE Master Secret and an OSCORE Master Salt, from which an OSCORE Security Context can be established.

  This procedure additionally provides forward secrecy.

* If one peer is acting as LwM2M Client and the other peer as LwM2M Server, according to the OMA Lightweight Machine to Machine Core specification [LwM2M], then the LwM2M Client peer may take the initiative to bootstrap again with the LwM2M Bootstrap Server, and receive again an OSCORE Security Context. Alternatively, the LwM2M Server can instruct the LwM2M Client to initiate this procedure.

  If the OSCORE Security Context information on the LwM2M Bootstrap Server has been updated, the LwM2M Client will thus receive a fresh OSCORE Security Context to use with the LwM2M Server.

  In addition to that, the LwM2M Client, the LwM2M Server as well as the LwM2M Bootstrap server are required to use the procedure defined in Appendix B.2 of [RFC8613] and overviewed above, when they use a certain OSCORE Security Context for the first time [LwM2M-Transport].

Manually updating the OSCORE Security Context at the two peers should be a last resort option, and it might often be not practical or feasible.

Even when any of the alternatives mentioned above is available, it is RECOMMENDED that two OSCORE peers update their Security Context by using the KUDOS procedure as defined in Section 4 of this document.

It is RECOMMENDED that the peer initiating the key update procedure starts it before reaching the 'q' or 'v' limits. Otherwise, the AEAD keys to be possibly used during the key update procedure itself may already be or become invalid before the rekeying is completed, which may prevent a successful establishment of the new OSCORE Security Context altogether.

In addition to approaching the 'q' or 'v' limits, there may be other reasons for a peer to initiate a key update procedure. These include: the OSCORE Security Context approaching its expiration, as per the 'exp' parameter defined in Section 2.2.1; application policies prescribing a regular key rollover; approaching the exhaustion of the Sender Sequence Number space in the OSCORE Sender Context.

4. Key Update for OSCORE (KUDOS)

This section defines KUDOS, a lightweight procedure that two OSCORE peers can use to update their keying material and establish a new OSCORE Security Context.

KUDOS relies on the OSCORE Option defined in [RFC8613] and extended as defined in Section 4.1, as well as on the support function updateCtx() defined in Section 4.2.

The message exchange between the two peers is defined in Section 4.3, with particular reference to the stateful FS mode providing forward secrecy. Building on the same message exchange, the possible use of the stateless no-FS mode is defined in Section 4.4, as intended to peers that are not able to write in non-volatile memory. Two peers MUST run KUDOS in FS mode if they are both capable to.

The key update procedure fulfills the following properties.

* KUDOS can be initiated by either peer. In particular, the client or the server may start KUDOS by sending the first rekeying message.

* The new OSCORE Security Context enjoys forward secrecy, unless KUDOS is run in no-FS mode (see Section 4.4).

* The same ID Context value used in the old OSCORE Security Context is preserved in the new Security Context. Furthermore, the ID Context value never changes throughout the KUDOS execution.

* KUDOS is robust against a peer rebooting, and it especially avoids the reuse of AEAD (nonce, key) pairs.

* KUDOS completes in one round trip. The two peers achieve mutual proof-of-possession in the following exchange, which is protected with the newly established OSCORE Security Context.

4.1. Extensions to the OSCORE Option

In order to support the message exchange for establishing a new OSCORE Security Context, this document extends the use of the OSCORE Option originally defined in [RFC8613] as follows.

* This document defines the usage of the seventh least significant bit, called "Extension-1 Flag", in the first byte of the OSCORE Option containing the OSCORE flag bits. This flag bit is specified in Section 7.2.

  When the Extension-1 Flag is set to 1, the second byte of the OSCORE Option MUST include the set of OSCORE flag bits 8-15.

* This document defines the usage of the least significant bit "Nonce Flag", 'd', in the second byte of the OSCORE Option containing the OSCORE flag bits 8-15. This flag bit is specified in Section 7.2.

  When it is set to 1, the compressed COSE object contains a 'nonce', to be used for the steps defined in Section 4.3. The 1 byte 'x' following 'kid context' (if any) encodes the length of 'nonce', together with signaling bits that indicate the specific behavior to adopt during the KUDOS execution. Specifically, the encoding of 'x' is as follows:

  - The four least significant bits encode the 'nonce' length in bytes minus 1, namely 'm'.

  - The fifth least significant bit is the "No Forward Secrecy" 'p' bit. The sender peer indicates its wish to run KUDOS in FS mode or in no-FS mode, by setting the 'p' bit to 0 or 1, respectively. This makes KUDOS possible to run also for peers that cannot support the FS mode. At the same time, two peers MUST run KUDOS in FS mode if they are both capable to, as per Section 4.3. The execution of KUDOS in no-FS mode is defined in Section 4.4.

  - The sixth least significant bit is the "Preserve Observations" 'b' bit. The sender peer indicates its wish to preserve ongoing observations beyond the KUDOS execution or not, by setting the 'b' bit to 1 or 0, respectively. The related processing is defined in Section 4.5.

  - The seventh and eight least significant bits are reserved for future use. These bits SHALL be set to zero when not in use. According to this specification, if any of these bits are set to 1, the message is considered to be malformed and decompression fails as specified in item 2 of Section 8.2 of [RFC8613].

  Hereafter, this document refers to a message where the 'd' flag is set to 0 as "non KUDOS (request/response) message", and to a message where the 'd' flag is set to 1 as "KUDOS (request/response) message".

* The second-to-eighth least significant bits in the second byte of the OSCORE Option containing the OSCORE flag bits are reserved

for future use. These bits SHALL be set to zero when not in use. According to this specification, if any of these bits are set to 1, the message is considered to be malformed and decompression fails as specified in item 2 of Section 8.2 of [RFC8613].

Figure 3 shows the OSCORE Option value including also 'nonce'.

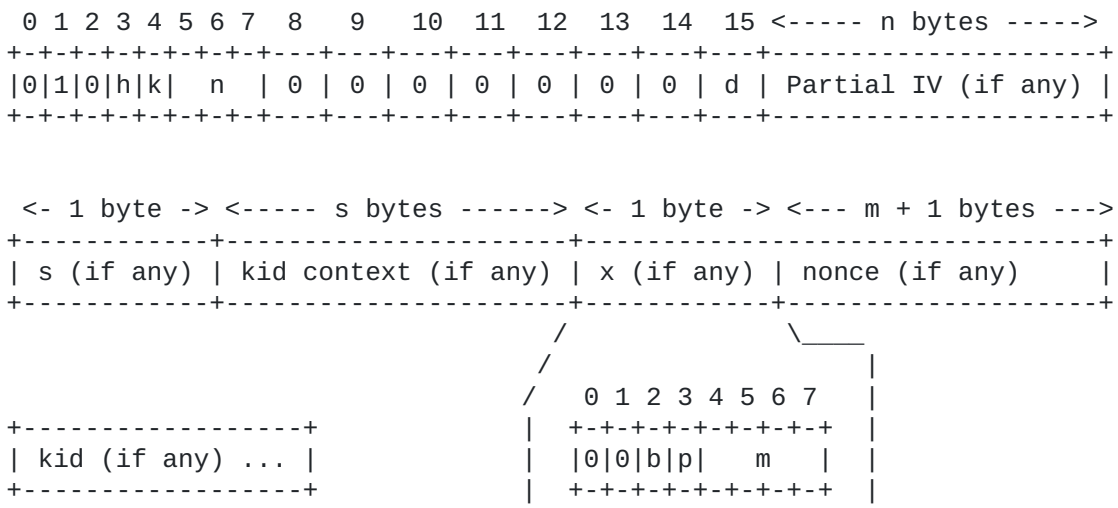


Figure 3: The OSCORE Option value, including 'nonce'

4.2. Function for Security Context Update

The updateCtx() function shown in Figure 4 takes as input three parameters X, N and CTX_IN. In particular, X and N are built from the 'x' and 'nonce' fields transported in the OSCORE Option value of the exchanged KUDOS messages (see Section 4.3.1), while CTX_IN is the OSCORE Security Context to update. The function returns a new OSCORE Security Context CTX_OUT.

As a first step, the updateCtx() function builds the two CBOR byte strings X_cbor and N_cbor, with value the input parameter X and N, respectively. Then, it builds the CBOR byte string X_N, with value the byte concatenation of X_cbor and N_cbor.

After that, the updateCtx() function derives the new values of the Master Secret and Master Salt for CTX_OUT. Specifically, the updateCtx() function uses one of the two following methods, depending on how the two peers established their original Security Context, i.e., the Security Context that they shared before performing KUDOS with one another for the first time.

*METHOD 1 - If the original Security Context was established by running the EDHOC protocol [I-D.ietf-lake-edhoc], the following applies.

First, the EDHOC-KeyUpdate() function defined in Section 4.2.2 of [I-D.ietf-lake-edhoc] is invoked. This results in deriving a new EDHOC key PRK_out shared by the two peers, and in using it to derive a new EDHOC key PRK_exporter, as per Section 4.2.1 of [I-D.ietf-lake-edhoc].

After that, the EDHOC-Exporter() function defined in Section 4.2.1 of [I-D.ietf-lake-edhoc] is used to derive the new values for the OSCORE Master Secret and Master Salt, consistently with what is defined in Appendix A.1 of [I-D.ietf-lake-edhoc]. In particular, the EDHOC-Exporter() function takes the new EDHOC key PRK_exporter derived above as first argument, while the context parameter provided as second argument is the empty CBOR byte string (0x40) [RFC8949], which is denoted as h''.

Note that, compared to the compliance requirements in Section 7 of [I-D.ietf-lake-edhoc], a peer MUST support the EDHOC-KeyUpdate() function, in case it establishes an original Security Context through the EDHOC protocol and intends to perform KUDOS.

The points in time when the two peers can delete the old EDHOC keys PRK_out and PRK_exporter are defined in Section 4.3.1 and Section 4.3.2, when specifying the client-initiated and server-initiated key update procedure, respectively. Until that point in time, a peer MUST retain the old EDHOC keys and MUST NOT replace them with the newly derived ones.

*METHOD 2 - If the original Security Context was established through other means than the EDHOC protocol, the new Master Secret is derived through an HKDF-Expand() step, which takes as input the Master Secret value from the Security Context CTX_IN, the literal string "key update", X_N and the length of the Master Secret. Instead, the new Master Salt takes N as value.

In either case, the derivation of new values follows the same approach used in TLS 1.3, which is also based on HKDF-Expand (see Section 7.1 of [RFC8446]) and used for computing new keying material in case of key update (see Section 4.6.3 of [RFC8446]).

After that, the new Master Secret and Master Salt parameters are used to derive a new Security Context CTX_OUT as per Section 3.2 of [RFC8613]. Any other parameter required for the derivation takes the same value as in the Security Context CTX_IN. Finally, the function returns the newly derived Security Context CTX_OUT.

Since the updateCtx() function also takes X as input, the derivation of CTX_OUT also considers as input the information from the 'x' field transported in the OSCORE Option value of the exchanged KUDOS messages. In turn, this ensures that, if successfully completed, a KUDOS execution occurs as intended by the two peers.

[

NOTE: In the case its EDHOC session has become invalid, a peer is unable to utilize the first method based on the EDHOC-KeyUpdate() function. In such a case, rather than running the EDHOC protocol again, it is preferable to fall back for the second method based on HKDF-Expand(). In order for the peer to signal the need for such a fall back, the seventh least significant bit in the 'x' byte of the OSCORE Option can be used. This may further justify a possible restructuring of the pseudocode, as two distinct updateCtx() functions implementing one method each.

]

```
updateCtx(X, N, CTX_IN) {

  CTX_OUT       // The new Security Context
  MSECRET_NEW   // The new Master Secret
  MSALT_NEW     // The new Master Salt

  X_cbor = bstr .cbor X // CBOR bstr wrapping of X
  N_cbor = bstr .cbor N // CBOR bstr wrapping of N
  X_N = bstr .cbor (X_cbor | N_cbor) // CBOR bstr wrapping of above

  oscore_key_length = < Size of CTX_IN.MasterSecret in bytes >

  if <the original Security Context was established through EDHOC> {
    // METHOD 1

    // Update the EDHOC key PRK_out, and use the
    // new one to update the EDHOC key PRK_exporter
    (new PRK_out, new PRK_exporter) = EDHOC-KeyUpdate(X_N)

    MSECRET_NEW = EDHOC-Exporter(0, h'', oscore_key_length)
      = EDHOC-KDF(new PRK_exporter, 0, h'', oscore_key_length)

    oscore_salt_length = < Size of CTX_IN.MasterSalt in bytes >

    MSALT_NEW = EDHOC-Exporter(1, h'', oscore_salt_length)
      = EDHOC-KDF(new PRK_exporter, 1, h'', oscore_salt_length)

  }
  else {
    // METHOD 2

    Label = "key update"

    MSECRET_NEW = HKDF-Expand-Label(CTX_IN.MasterSecret, Label,
                                    X_N, oscore_key_length)
                = HKDF-Expand(CTX_IN.MasterSecret, HkdfLabel,
                              oscore_key_length)

    MSALT_NEW = N;
  }

  < Derive CTX_OUT using MSECRET_NEW and MSALT_NEW,
    together with other parameters from CTX_IN >

  Return CTX_OUT;

}

Where HkdfLabel is defined as

struct {
    uint16 length = oscore_key_length;
    opaque label<7..255> = "oscore " + Label;
    opaque context<0..255> = X_N;
} HkdfLabel;
```

Figure 4: Function for deriving a new OSCORE Security Context

4.3. Key Update with Forward Secrecy

This section defines the actual KUDOS procedure performed by two peers to update their OSCORE keying material. Before starting KUDOS, the two peers share the OSCORE Security Context CTX_OLD. Once successfully completed the KUDOS execution, the two peers agree on a newly established OSCORE Security Context CTX_NEW.

The following specifically defines how KUDOS is run in its stateful FS mode achieving forward secrecy. That is, in the OSCORE Option value of all the exchanged KUDOS messages, the "No Forward Secrecy" bit is set to 0.

In order to run KUDOS in FS mode, both peers have to be able to write in non-volatile memory the OSCORE Master Secret and OSCORE Master Salt from the newly derived Security Context CTX_NEW. If this is not the case, the two peers have to run KUDOS in its stateless no-FS mode (see Section 4.4).

When running KUDOS, each peer contributes by generating a fresh value N1 or N2, and providing it to the other peer. Furthermore, X1 and X2 are the value of the 'x' byte specified in the OSCORE Option of the first and second KUDOS message, respectively. As defined in Section 4.3.1, these values are used by the peers to build the input N and X to the updateCtx() function, in order to derive a new OSCORE Security Context. As for any new OSCORE Security Context, the Sender Sequence Number and the replay window are re-initialized accordingly (see Section 3.2.2 of [RFC8613]).

Once a peer has successfully derived the new OSCORE Security Context CTX_NEW, that peer MUST use CTX_NEW to protect outgoing non KUDOS messages.

Also, that peer MUST terminate all the ongoing observations [RFC7641] that it has with the other peer as protected with the old Security Context CTX_OLD, unless the two peers have explicitly agreed otherwise as defined in Section 4.5. More specifically, if either or both peers indicate the wish to cancel their observations, those will be all cancelled following a successful KUDOS execution.

Note that, even though that peer had no real reason to update its OSCORE keying material, running KUDOS can be intentionally exploited as a more efficient way to terminate all the ongoing observations with the other peer, compared to sending one cancellation request per observation (see Section 3.6 of [RFC7641]).

Once a peer has successfully decrypted and verified an incoming message protected with CTX_NEW, that peer MUST discard the old Security Context CTX_OLD.

KUDOS can be started by the client or the server, as defined in Section 4.3.1 and Section 4.3.2, respectively. The following properties hold for both the client- and server-initiated version of KUDOS.

*The initiator always offers the fresh value N1.

*The responder always offers the fresh value N2

* The responder is always the first one deriving the new OSCORE Security Context CTX_NEW.

* The initiator is always the first one achieving key confirmation, hence the first one able to safely discard the old OSCORE Security Context CTX_OLD.

* Both the initiator and the responder use the same respective OSCORE Sender ID and Recipient ID. Also, they both preserve and use the same OSCORE ID Context from CTX_OLD.

The length of the nonces N1 and N2 is application specific. The application needs to set the length of each nonce such that the probability of its value being repeated is negligible. To this end, each nonce is typically at least 8 bytes long.

Once a peer acting as initiator (responder) has sent (received) the first KUDOS message, that peer MUST NOT send a non KUDOS message to the other peer, until having completed the key update process on its side. The initiator completes the key update process when receiving the second KUDOS message and successfully verifying it with the new OSCORE Security Context CTX_NEW. The responder completes the key update process when sending the second KUDOS message, as protected with the new OSCORE Security Context CTX_NEW.

In the following sections, 'Comb(a,b)' denotes the byte concatenation of two CBOR byte strings, where the first one has value 'a' and the second one has value 'b'. That is, Comb(a,b) = bstr .cbor a | bstr .cbor b, where | denotes byte concatenation.

4.3.1. Client-Initiated Key Update

Figure 5 shows the KUDOS workflow with the client acting as initiator.

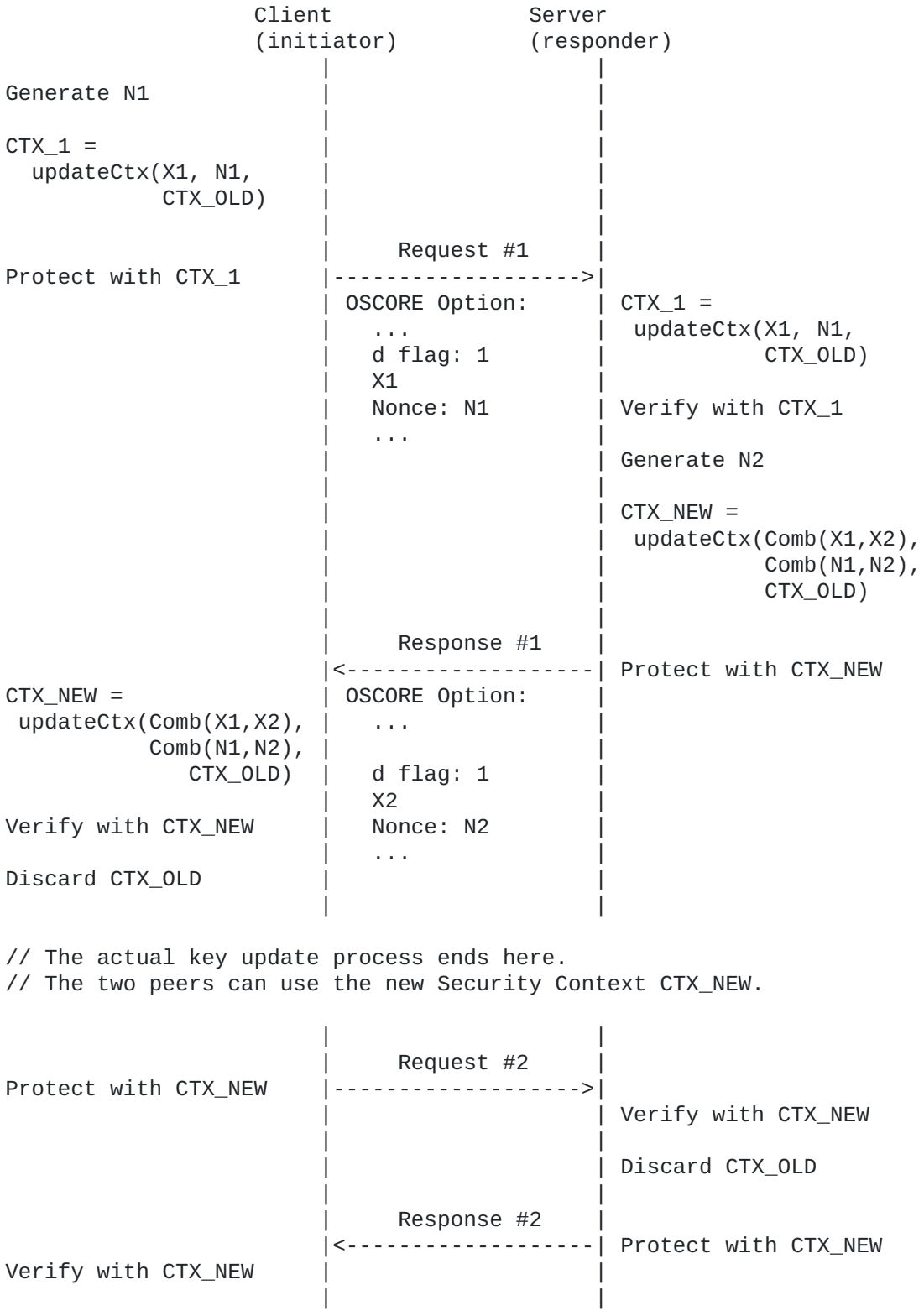


Figure 5: Client-Initiated KUDOS Workflow

First, the client generates a random value N1, and uses the nonce N = N1 and X = X1 together with the old Security Context CTX_OLD, in order to derive a temporary Security Context CTX_1.

If the peers' original Security Context was derived through the EDHOC protocol and the updateCtx() function in Section 4.2 used METHOD 1 to derive CTX_1, then the client deletes the newly derived EDHOC keys PRK_out and PRK_exporter, which do not replace the old ones.

Then, the client sends an OSCORE request to the server, protected with the Security Context CTX_1. In particular, the request has the 'd' flag bit set to 1, and specifies X1 as 'x' and N1 as 'nonce' (see Section 4.1). After that, the client deletes CTX_1.

Upon receiving the OSCORE request, the server retrieves the value N1 from the 'nonce' field of the request, the value X1 from the 'x' byte of the OSCORE Option, and provides the updateCtx() function with the input N = N1, X = X1 and the old Security Context CTX_OLD, in order to derive the temporary Security Context CTX_1.

Figure 6 shows an example of how the two peers compute X and N provided as input to the updateCtx() function, and how they compute X_N within the updateCtx() function, when deriving CTX_1 (see Section 4.2).

```
X1 and N1 expressed as raw values
X1 = 0x80
N1 = 0x018a278f7faab55a

updateCtx() is called with
X = 0x80
N = 0x018a278f7faab55a

In updateCtx(), X_cbor and N_cbor are built as CBOR byte strings
X_cbor = 0x4180               (h'80')
N_cbor = 0x48018a278f7faab55a (h'018a278f7faab55a')

In updateCtx(), X_N is built from N_cbor and X_cbor
X_N = 0x4b418048018a278f7faab55a (h'418048018a278f7faab55a')
```

Figure 6: Example of X, N and X_N computing for the first KUDOS message

If the peers' original Security Context was derived through the EDHOC protocol and the updateCtx() function in Section 4.2 used METHOD 1 to derive CTX_1, then the server deletes the newly derived EDHOC keys PRK_out and PRK_exporter, which do not replace the old ones.

Then, the server verifies the request by using the Security Context CTX_1.

After that, the server generates a random value N2, and uses N = Comb(N1, N2) and X = Comb(X1, X2) together with the old Security Context CTX_OLD, in order to derive the new Security Context CTX_NEW.

An example of this nonce processing on the server with values for N1, X1, N2 and X2 is presented in Figure 7.

```
X1, X2, N1 and N2 expressed as raw values
X1 = 0x80
X2 = 0x80
N1 = 0x018a278f7faab55a
N2 = 0x25a8991cd700ac01

X1, X2, N1 and N2 as CBOR byte strings
X1 = 0x4180 (h'80')
X2 = 0x4180 (h'80')
N1 = 0x48018a278f7faab55a (h'018a278f7faab55a')
N2 = 0x4825a8991cd700ac01 (h'25a8991cd700ac01')

updateCtx() is called with
X = 0x41804180
N = 0x48018a278f7faab55a4825a8991cd700ac01

In updateCtx(), X_cbor and N_cbor are built as CBOR byte strings
X_cbor = 0x4441804180 (h'41804180')
N_cbor = 0x5248018a278f7faab55a4825a8991cd700ac01
         (h'48018a278f7faab55a4825a8991cd700ac01')

In updateCtx(), X_N is built from N_cbor and X_cbor
X_N = 0x581844418041805248018a278f7faab55a4825a8991cd700ac01
      (h'444180418052480 18a278f7faab55a4825a8991cd700ac01')
```

Figure 7: Example of X, N and X_N computing for the second KUDOS message

Then, the server sends an OSCORE response to the client, protected with the new Security Context CTX_NEW. In particular, the response has the 'd' flag bit set to 1 and specifies N2 as 'nonce'. After that, the server deletes CTX_1.

Upon receiving the OSCORE response, the client retrieves the value N2 from the 'nonce' field of the response, and the value X2 from the 'x' byte of the OSCORE Option. Since the client has received a response to an OSCORE request it made with the 'd' flag bit set to 1, the client provides the updateCtx() function with the input N = Comb(N1, N2), X = Comb(X1, X2) and the old Security Context CTX_OLD, in order to derive the new Security Context CTX_NEW. Finally, the client verifies the response by using the Security Context CTX_NEW and deletes the old Security Context CTX_OLD.

If the peers' original Security Context was derived through the EDHOC protocol and the updateCtx() function in Section 4.2 used METHOD 1 to derive CTX_NEW, then the client replaces the old EDHOC keys PRK_out and PRK_exporter with the newly derived ones.

Then, the client can send a new OSCORE request protected with the new Security Context CTX_NEW.

When successfully verifying the request using the Security Context CTX_NEW, the server deletes the old Security Context CTX_OLD and can reply with an OSCORE response protected with the new Security Context CTX_NEW.

If the peers' original Security Context was derived through the EDHOC protocol and the updateCtx() function in Section 4.2 used

METHOD 1 to derive CTX_NEW, then the server replaces the old EDHOC keys PRK_out and PRK_exporter with the newly derived ones.

From then on, the two peers can protect their message exchanges by using the new Security Context CTX_NEW.

Note that the server achieves key confirmation only when receiving a message from the client as protected with the new Security Context CTX_NEW. If the server sends a non KUDOS request to the client protected with CTX_NEW before then, and the server receives a 4.01 (Unauthorized) error response as reply, the server SHOULD delete the new Security Context CTX_NEW and start a new client-initiated key update process, by taking the role of initiator as per Figure 5.

Also note that, if both peers reboot simultaneously, they will run the client-initiated version of KUDOS defined in this section. That is, one of the two peers implementing a CoAP client will send KUDOS Request #1 in Figure 5.

4.3.1.1. Avoiding In-Transit Requests During a Key Update

Before sending the KUDOS message Request #1 in Figure 5, the client MUST ensure that it has no outstanding interactions with the server (see Section 4.7 of [RFC7252]), with the exception of ongoing observations [RFC7641] with that server.

If there are any, the client MUST NOT initiate the KUDOS execution, before either: i) having all those outstanding interactions cleared; or ii) freeing up the Token values used with those outstanding interactions, with the exception of ongoing observations with the server.

Later on, this prevents a non KUDOS response protected with the new Security Context CTX_NEW to cryptographically match with both the corresponding request also protected with CTX_NEW and with an older request protected with CTX_OLD, in case the two requests were protected using the same OSCORE Partial IV.

4.3.2. Server-Initiated Key Update

Figure 8 shows the KUDOS workflow with the server acting as initiator.

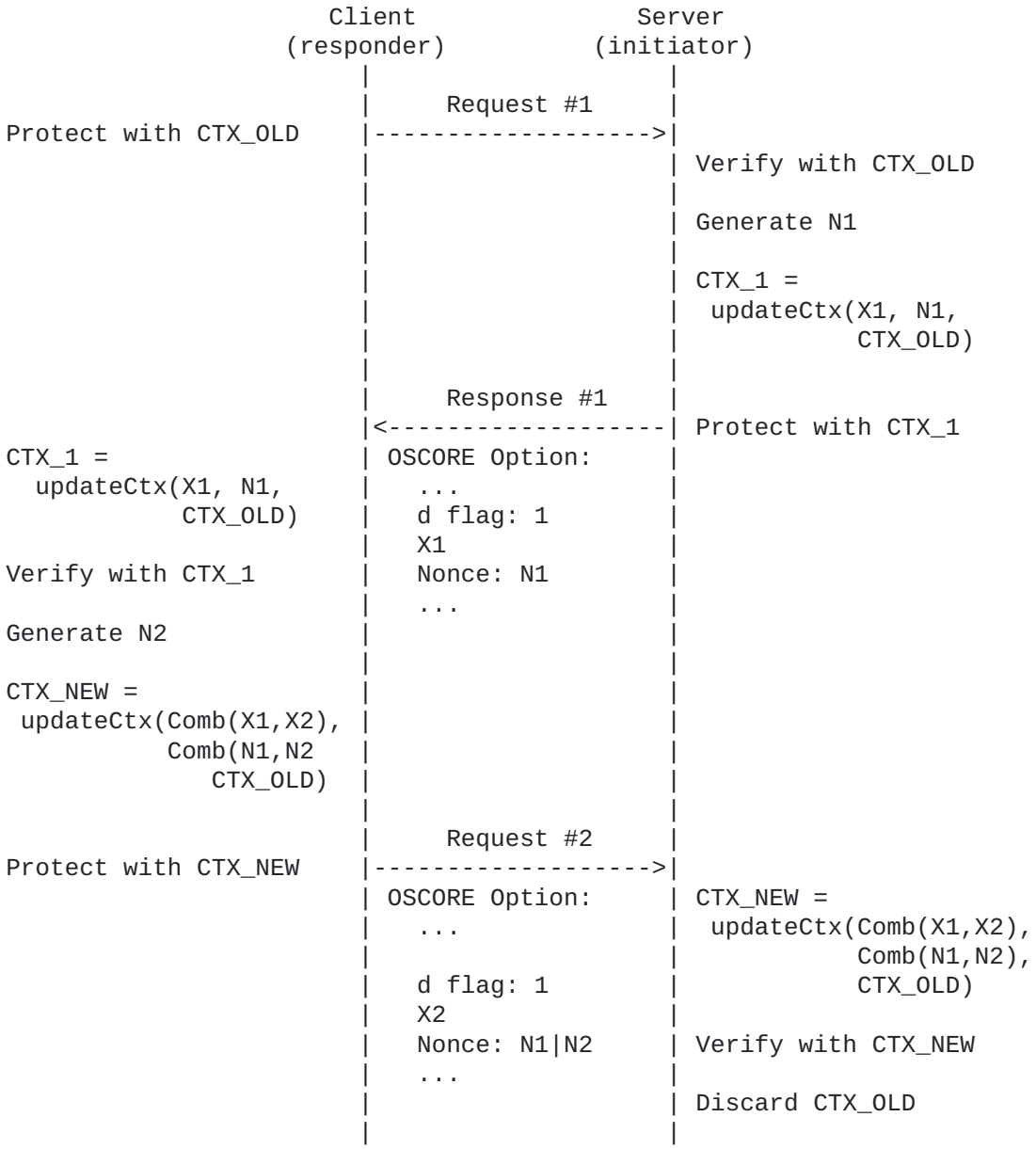


Figure 8: Server-Initiated KUDOS Workflow

First, the client sends a normal OSCORE request to the server, protected with the old Security Context CTX_OLD and with the 'd' flag bit set to 0.

Upon receiving the OSCORE request and after having verified it with the old Security Context CTX_OLD as usual, the server generates a random value N1 and provides the updateCtx() function with the input N = N1, X = X1 and the old Security Context CTX_OLD, in order to derive the temporary Security Context CTX_1.

If the peers' original Security Context was derived through the EDHOC protocol and the updateCtx() function in Section 4.2 used METHOD 1 to derive CTX_1, then the server deletes the newly derived EDHOC keys PRK_out and PRK_exporter, which do not replace the old ones.

Then, the server sends an OSCORE response to the client, protected with the Security Context CTX_1. In particular, the response has the 'd' flag bit set to 1 and specifies N1 as 'nonce' (see Section 4.1). After that, the server deletes CTX_1.

Upon receiving the OSCORE response, the client retrieves the value N1 from the 'nonce' field of the response, the value X1 from the 'x' byte of the OSCORE Option, and provides the updateCtx() function with the input N = N1, X = X1 and the old Security Context CTX_OLD, in order to derive the temporary Security Context CTX_1.

If the peers' original Security Context was derived through the EDHOC protocol and the updateCtx() function in Section 4.2 used METHOD 1 to derive CTX_1, then the client deletes the newly derived EDHOC keys PRK_out and PRK_exporter, which do not replace the old ones.

Then, the client verifies the response by using the Security Context CTX_1.

After that, the client generates a random value N2, and provides the updateCtx() function with the input N = Comb(N1, N2), X = Comb(X1, X2) and the old Security Context CTX_OLD, in order to derive the new Security Context CTX_NEW. Then, the client sends an OSCORE request to the server, protected with the new Security Context CTX_NEW. In particular, the request has the 'd' flag bit set to 1 and specifies N1 | N2 as 'nonce'. After that, the client deletes CTX_1.

Upon receiving the OSCORE request, the server retrieves the value N1 | N2 from the request and the value X2 from the 'x' byte of the OSCORE Option. Then, the server verifies that: i) the value N1 is identical to the value N1 specified in a previous OSCORE response with the 'd' flag bit set to 1; and ii) the value N1 | N2 has not been received before in an OSCORE request with the 'd' flag bit set to 1.

If the verification succeeds, the server provides the updateCtx() function with the input N = Comb(N1, N2), X = Comb(X1, X2) and the old Security Context CTX_OLD, in order to derive the new Security Context CTX_NEW. Finally, the server verifies the request by using the Security Context CTX_NEW and deletes the old Security Context CTX_OLD.

If the peers' original Security Context was derived through the EDHOC protocol and the updateCtx() function in Section 4.2 used

METHOD 1 to derive CTX_NEW, then the server replaces the old EDHOC keys PRK_out and PRK_exporter with the newly derived ones.

After that, the server can send an OSCORE response protected with the new Security Context CTX_NEW.

When successfully verifying the response using the Security Context CTX_NEW, the client deletes the old Security Context CTX_OLD.

If the peers' original Security Context was derived through the EDHOC protocol and the updateCtx() function in Section 4.2 used METHOD 1 to derive CTX_NEW, then the client replaces the old EDHOC keys PRK_out and PRK_exporter with the newly derived ones.

From then on, the two peers can protect their message exchanges by using the new Security Context CTX_NEW.

Note that the client achieves key confirmation only when receiving a message from the server as protected with the new Security Context CTX_NEW. If the client sends a non KUDOS request to the server protected with CTX_NEW before then, and the client receives a 4.01 (Unauthorized) error response as reply, the client SHOULD delete the new Security Context CTX_NEW and start a new client-initiated key update process, by taking the role of initiator as per Figure 5 in Section 4.3.1.

4.3.2.1. Avoiding In-Transit Requests During a Key Update

Before sending the KUDOS message Request #2 in Figure 8, the client MUST ensure that it has no outstanding interactions with the server (see Section 4.7 of [RFC7252]), with the exception of ongoing observations [RFC7641] with that server.

If there are any, the client MUST NOT initiate the KUDOS execution, before either: i) having all those outstanding interactions cleared; or ii) freeing up the Token values used with those outstanding interactions, with the exception of ongoing observations with the server.

Later on, this prevents a non KUDOS response protected with the new Security Context CTX_NEW to cryptographically match with both the corresponding request also protected with CTX_NEW and with an older request protected with CTX_OLD, in case the two requests were protected using the same OSCORE Partial IV.

4.3.2.2. Preventing Deadlock Situations

When the server-initiated version of KUDOS is used, the two peers risk to run into a deadlock, if all the following conditions hold.

* The client is a client-only device, i.e., it is not capable to act as CoAP server and thus does not listen for incoming requests.

* The server needs to execute KUDOS, which, due to the previous point, can only be performed in its server-initiated version as per Figure 8. That is, the server has to wait for an incoming non

KUDOS request, in order to initiate KUDOS by replying with the first KUDOS message as a response.

* The client sends only Non-confirmable CoAP requests to the server and does not expect responses sent back as reply, hence freeing up a request's Token value once the request is sent.

In such a case, in order to avoid experiencing a deadlock situation where the server needs to execute KUDOS but cannot practically initiate it, a client-only device that supports KUDOS SHOULD intersperse Non-confirmable requests it sends to that server with confirmable requests.

4.4. Key Update with or without Forward Secrecy

The FS mode of the KUDOS procedure defined in Section 4.3 ensures forward secrecy of the OSCORE keying material. However, it requires peers executing KUDOS to preserve their state (e.g., across a device reboot), by writing information such as data from the newly derived OSCORE Security Context CTX_NEW in non-volatile memory.

This can be problematic for devices that cannot dynamically write information to non-volatile memory. For example, some devices may support only a single writing in persistent memory when initial keying material is provided (e.g., at manufacturing or commissioning time), but no further writing after that. Therefore, these devices cannot perform a stateful key update procedure, and thus are not capable to run KUDOS in FS mode to achieve forward secrecy.

In order to address these limitations, KUDOS can be run in its stateless no-FS mode, as defined in the following. This allows two peers to achieve the same results as when running KUDOS in FS mode (see Section 4.3), with the difference that no forward secrecy is achieved and no state information is required to be dynamically written in non-volatile memory.

From a practical point of view, the two modes differ as to what exact OSCORE Master Secret and Master Salt are used as part of the OSCORE Security Context CTX_OLD provided as input to the updateCtx() function (see Section 4.2).

If either or both peers are not able to write in non-volatile memory the OSCORE Master Secret and OSCORE Master Salt from the newly derived Security Context CTX_NEW, then the two peers have to run KUDOS in no-FS mode.

4.4.1. Handling and Use of Keying Material

In the following, a device is denoted as "CAPABLE" if it is able to store information in non-volatile memory (e.g., on disk), beyond a one-time-only writing occurring at manufacturing or (re-)commissioning time.

The following terms are used to refer to OSCORE keying material.

* Bootstrap Master Secret and Bootstrap Master Salt. If pre-provisioned during manufacturing or (re-)commissioning, these

OSCORE Master Secret and Master Salt are initially stored on disk and are never going to be overwritten by the device.

* Latest Master Secret and Latest Master Salt. These OSCORE Master Secret and Master Salt can be dynamically updated by the device. In case of reboot, they are lost unless they have been stored on disk.

Note that:

* A peer running KUDOS can have none of the pairs above associated with another peer, only one or both.

* A peer that has neither of the pairs above associated with another peer, cannot run KUDOS in any mode with that other peer.

* A peer that has only one of the pairs above associated with another peer can attempt to run KUDOS with that other peer, but the procedure might fail depending on the other peer's capabilities. In particular:

  - In order to run KUDOS in FS mode, a peer must be a CAPABLE device. It follows that two peers have to both be CAPABLE devices in order to be able to run KUDOS in FS mode with one another.

  - In order to run KUDOS in no-FS mode, a peer must have Bootstrap Master Secret and Bootstrap Master Salt available as stored on disk.

As a general rule, once successfully generated a new OSCORE Security Context CTX (e.g., CTX is the CTX_NEW resulting from a KUDOS execution, or it has been established through the EDHOC protocol [I-D.ietf-lake-edhoc]), a peer considers the Master Secret and Master Salt of CTX as Latest Master Secret and Latest Master Salt. After that:

* If the peer is a CAPABLE device, it SHOULD store Latest Master Secret and Latest Master Salt on disk.

  As an exception, this does not apply to possible temporary OSCORE Security Contexts used during a key update procedure, such as CTX_1 used during the KUDOS execution. That is, the OSCORE Master Secret and Master Salt from such temporary Security Contexts MUST NOT be stored on disk.

* The peer MUST store Latest Master Secret and Latest Master Salt in volatile memory, thus making them available to OSCORE message processing and possible key update procedures.

4.4.1.1. Actions after Device Reboot

Building on the above, after having experienced a reboot, a peer A checks whether it has stored on disk a pair P1 = (Latest Master Secret, Latest Master Salt) associated with any another peer B.

* If a pair P1 is found, the peer A performs the following actions.

  - The peer A loads the Latest Master Secret and Latest Master Salt to volatile memory, and uses them to derive an OSCORE Security Context CTX_OLD.

  - The peer A runs KUDOS with the other peer B, acting as initiator. If the peer A is a CAPABLE device, it stores on disk the Master Secret and Master Salt from the newly established OSCORE Security Context CTX_NEW, as Latest Master Secret and Latest Master Salt, respectively.

* If a pair P1 is not found, the peer A checks whether it has stored on disk a pair P2 = (Bootstrap Master Secret, Bootstrap Master Salt) associated with the other peer B.

  - If a pair P2 is found, the peer A performs the following actions.

    o The peer A loads the Bootstrap Master Secret and Bootstrap Master Salt to volatile memory, and uses them to derive an OSCORE Security Context CTX_OLD.

    o If the peer A is a CAPABLE device, it stores on disk Bootstrap Master Secret and Bootstrap Master Salt as Latest Master Secret and Latest Master Salt, respectively. This supports the situation where A is a CAPABLE device and has never run KUDOS with the other peer B before.

    o The peer A runs KUDOS with the other peer B, acting as initiator. If the peer A is a CAPABLE device, it stores on disk the Master Secret and Master Salt from the newly established OSCORE Security Context CTX_NEW, as Latest Master Secret and Latest Master Salt, respectively.

  - If a pair P2 is not found, the peer A has to use alternative ways to establish a first OSCORE Security Context CTX_NEW with the other peer B, e.g., by running the EDHOC protocol. After that, if A is a CAPABLE device, it stores on disk the OSCORE Master Secret and Master Salt from the newly established OSCORE Security Context CTX_NEW, as Latest Master Secret and Latest Master Salt, respectively.

4.4.2. Selection of KUDOS Mode

During a KUDOS execution, the two peers agree on whether to perform the key update procedure in FS mode or no-FS mode, by leveraging the "No Forward Secrecy" bit, 'p', in the 'x' byte of the OSCORE Option value of the KUDOS messages (see Section 4.1). The 'p' bit practically determines what OSCORE Security Context to use as

CTX_OLD during the KUDOS execution, consistently with the indicated mode.

* If the 'p' bit is set to 0 (FS mode), the updateCtx() function used to derive CTX_1 or CTX_NEW considers as input CTX_OLD the current OSCORE Security Context shared with the other peer as is. In particular, CTX_OLD includes Latest Master Secret as OSCORE Master Secret and Latest Master Salt as OSCORE Master Salt.

* If the 'p' bit is set to 1 (no-FS mode), the updateCtx() function used to derive CTX_1 or CTX_NEW considers as input CTX_OLD the current OSCORE Security Context shared with the other peer, with the following difference: Bootstrap Master Secret is used as OSCORE Master Secret and Bootstrap Master Salt is used as OSCORE Master Salt. That is, every execution of KUDOS in no-FS mode between these two peers considers the same pair (Master Secret, Master Salt) in the OSCORE Security Context CTX_OLD provided as input to the updateCtx() function, hence the impossibility to achieve forward secrecy.

A peer determines to run KUDOS either in FS or no-FS mode with another peer as follows.

* If a peer A is not a CAPABLE device, it MUST run KUDOS only in no-FS mode. That is, when sending a KUDOS message, it MUST set to 1 the 'p' bit of the 'x' byte in the OSCORE Option value.

* If a peer A is a CAPABLE device, it SHOULD run KUDOS only in FS mode and SHOULD NOT run KUDOS as initiator in no-FS mode. That is, when sending a KUDOS message, it SHOULD set to 0 the 'p' bit of the 'x' byte in the OSCORE Option value. An exception applies in the following cases.

  - The peer A is running KUDOS with another peer B, which A has learned to not be a CAPABLE device (and hence not able to run KUDOS in FS mode).

    Note that, if the peer A is a CAPABLE device, it is able to store such information about the other peer B on disk and it MUST do so. From then on, the peer A will perform every execution of KUDOS with the peer B in no-FS mode, including after a possible reboot.

  - The peer A is acting as responder and running KUDOS with another peer B without knowing its capabilities, and A receives a KUDOS message where the 'p' bit of the 'x' byte in the OSCORE Option value is set to 1.

* If the peer A is a CAPABLE device and has learned that another peer B is also a CAPABLE device (and hence able to run KUDOS in FS mode), then the peer A MUST NOT run KUDOS with the peer B in no-FS mode. This also means that, if the peer A acts as responder when running KUDOS with the peer B, the peer A MUST terminate the KUDOS execution if it receives a KUDOS message from the peer B where the 'p' bit of the 'x' byte in the OSCORE Option value is set to 1.

Note that, if the peer A is a CAPABLE device, it is able to store such information about the other peer B on disk and it MUST do so. This ensures that the peer A will perform every execution of KUDOS with the peer B in FS mode. In turn, this prevents a possible downgrading attack, aimed at making A believe that B is not a CAPABLE device, and thus to run KUDOS in no-FS mode although the FS mode can actually be used by both peers.

Within the limitations above, two peers running KUDOS generate the new OSCORE Security Context CTX_NEW according to the mode indicated per the bit 'p' set by the responder in the second KUDOS message.

If, after having received the first KUDOS message, the responder can continue performing KUDOS, the bit 'p' in the reply message has the same value as in the bit 'p' set by the initiator, unless the value is 0 and the responder is not a CAPABLE device. More specifically:

* If both peers are CAPABLE devices, they will run KUDOS in FS mode. That is, both initiator and responder sets the 'p' bit to 0 in the respective sent KUDOS message.

* If both peers are not CAPABLE devices or only the peer acting as initiator is not a CAPABLE device, they will run KUDOS in no-FS mode. That is, both initiator and responder sets the 'p' bit to 1 in the respective sent KUDOS message.

* If only the peer acting as initiator is a CAPABLE device and it has knowledge of the other peer being a not CAPABLE device, they will run KUDOS in no-FS mode. That is, both initiator and responder sets the 'p' bit to 1 in the respective sent KUDOS message.

* If only the peer acting as initiator is a CAPABLE device and it has no knowledge of the other peer being a not CAPABLE device, they will not run KUDOS in FS mode and will rather set to ground for possibly retrying in no-FS mode. In particular, the initiator sets the 'p' bit of its sent KUDOS message to 0. Then:

  - If the responder is a server, it MUST reply with a 5.03 (Service Unavailable) error response. The response MUST be protected with the newly derived OSCORE Security Context CTX_NEW. The diagnostic payload MAY provide additional information. In the error response, the 'p' bit MUST be set to 1.

    When receiving the error response, the initiator learns that the responder is not a CAPABLE device (and hence not able to run KUDOS in FS mode). The initiator MAY try running KUDOS again. If it does so, the initiator MUST set the 'p' bit to 1, when sending a new request as first KUDOS message.

  - If the responder is a client, it sends to the initiator the second KUDOS message as a new request, which MUST be protected with the newly derived OSCORE Security Context CTX_NEW. In the newly sent request, the 'p' bit MUST be set to 1.

    When receiving the new request above (i.e., with the 'p' bit set to 1 as a follow-up to the previous KUDOS response having

the 'p' bit set to 0), the initiator learns that the responder is not a CAPABLE device (and hence not able to run KUDOS in FS mode).

In either case, both KUDOS peers delete the OSCORE Security Contexts CTX_1 and CTX_NEW.

4.5. Preserving Observations across Key Updates

As defined in Section 4.3, once a peer has completed the KUDOS execution and successfully derived the new OSCORE Security Context CTX_NEW, that peer normally terminates all the ongoing observations it has with the other peer [RFC7641], as protected with the old OSCORE Security Context CTX_OLD.

This section describes a method that the two peers can use to safely preserve the ongoing observations that they have with one another, beyond the completion of a KUDOS execution. In particular, this method ensures that an Observe notification can never successfully cryptographically match against the Observe requests of two different observations, i.e., against an Observe request protected with CTX_OLD and an Observe request protected with CTX_NEW.

The actual preservation of ongoing observations has to be agreed by the two peers at each execution of KUDOS that they run with one another, as defined in Section 4.5.1. If, at the end of a KUDOS execution, the two peers have not agreed on that, they MUST terminate the ongoing observations that they have with one another, just as defined in Section 4.3.

[

NOTE: While a dedicated signaling would have to be introduced, this rationale may be of more general applicability, i.e., in case an update of the OSCORE keying material is performed through a different means than KUDOS.

]

4.5.1. Management of Observations

As per Section 3.1 of [RFC7641], a client can register its interest in observing a resource at a server, by sending a registration request including the Observe Option with value 0.

If the server registers the observation as ongoing, the server sends back a successful response also including the Observe Option, hence confirming that an entry has been successfully added for that client.

If the client receives back the successful response above from the server, then the client also registers the observation as ongoing.

In case the client can ever consider to preserve ongoing observations beyond a key update as defined below, then the client MUST NOT simply forget about an ongoing observation if not interested in it anymore. Instead, the client MUST send an explicit cancellation request to the server, i.e., a request including the

Observe Option with value 1 (see Section 3.6 of [RFC7641]). After sending this cancellation request, if the client does not receive back a response confirming that the observation has been terminated, the client MUST NOT consider the observation terminated. The client MAY try again to terminate the observation by sending a new cancellation request.

In case a peer A performs a KUDOS execution with another peer B, and A has ongoing observations with B that it is interested to preserve beyond the key update, then A can explicitly indicate its interest to do so. To this end, the peer A sets to 1 the bit "Preserve Observations", 'b', in the 'x' byte of the OSCORE Option value (see Section 4.1), in the KUDOS message it sends to the other peer B.

If a peer acting as responder receives the first KUDOS message with the bit 'b' set to 0, then the peer MUST set to 0 the bit 'b' in the KUDOS message it sends as follow-up, regardless of its wish to preserve ongoing observations with the other peer.

If a peer acting as initiator has sent the first KUDOS message with the bit 'b' set to 0, the peer MUST ignore the bit 'b' in the follow-up KUDOS message that it receives from the other peer.

After successfully completing the KUDOS execution (i.e., after having successfully derived the new OSCORE Security Context CTX_NEW), both peers have expressed their interest in preserving their common ongoing observations if and only if the bit 'b' was set to 1 in both the exchanged KUDOS messages. In such a case, each peer X performs the following actions.

1. The peer X considers all the still ongoing observations that it has with the other peer, such that X acts as client in those observations. If there are no such observations, the peer X takes no further actions. Otherwise, it moves to step 2.

2. The peer X considers all the OSCORE Partial IV values used in the Observe registration request associated with any of the still ongoing observations determined at step 1.

3. The peer X determines the value PIV* as the highest OSCORE Partial IV value among those considered at step 2.

4. In the Sender Context of the OSCORE Security Context shared with the other peer, the peer X sets its own Sender Sequence Number to (PIV* + 1), rather than to 0.

As a result, each peer X will "jump" beyond the OSCORE Partial IV (PIV) values that are occupied and in use for ongoing observations with the other peer where X acts as client.

Note that, each time it runs KUDOS, a peer must determine if it wishes to preserve ongoing observations with the other peer or not, before sending its KUDOS message.

To this end, the peer should also assess the new value that PIV* would take after a successful completion of KUDOS, in case ongoing observations with the other peer are going to be preserved. If the peer considers such a new value of PIV* to be too close to the

maximum possible value admitted for the OSCORE Partial IV, then the peer may choose to run KUDOS with no intention to preserve its ongoing observations with the other peer, in order to "start over" from a fresh, entirely unused PIV space.

Application policies can further influence whether attempting to preserve observations beyond a key update is appropriate or not.

4.6. Retention Policies

Applications MAY define policies that allow a peer to also temporarily keep the old Security Context CTX_OLD, rather than simply overwriting it to become CTX_NEW. This allows the peer to decrypt late, still on-the-fly incoming messages protected with CTX_OLD.

When enforcing such policies, the following applies.

* Outgoing non KUDOS messages MUST be protected by using only CTX_NEW.

* Incoming non KUDOS messages MUST first be attempted to decrypt by using CTX_NEW. If decryption fails, a second attempt can use CTX_OLD.

* When an amount of time defined by the policy has elapsed since the establishment of CTX_NEW, the peer deletes CTX_OLD.

4.7. Discussion

KUDOS is intended to deprecate and replace the procedure defined in Appendix B.2 of [RFC8613], as fundamentally achieving the same goal, while displaying a number of improvements and advantages.

In particular, it is especially convenient for the handling of failure events concerning the JRC node in 6TiSCH networks (see Section 3). That is, among its intrinsic advantages compared to the procedure defined in Appendix B.2 of [RFC8613], KUDOS preserves the same ID Context value, when establishing a new OSCORE Security Context.

Since the JRC uses ID Context values as identifiers of network nodes, namely "pledge identifiers", the above implies that the JRC does not have to perform anymore a mapping between a new, different ID Context value and a certain pledge identifier (see Section 8.3.3 of [RFC9031]). It follows that pledge identifiers can remain constant once assigned, and thus ID Context values used as pledge identifiers can be employed in the long-term as originally intended.

5. Update of OSCORE Sender/Recipient IDs

This section defines a procedure that two peers can perform, in order to update the OSCORE Sender/Recipient IDs that they use in their shared OSCORE Security Context.

This procedure can be initiated by either peer. In particular, the client or the server may start it by sending the first OSCORE IDs

update message. When sending an OSCORE IDs update message, a peer provides its new intended OSCORE Recipient ID to the other peer.

Furthermore, this procedure can be executed stand-alone, or instead seamlessly integrated in an execution of KUDOS (see Section 4) using its FS mode or no-FS mode (see Section 4.4).

- In the former stand-alone case, updating the OSCORE Sender/ Recipient IDs effectively results in updating part of the current OSCORE Security Context.

  That is, both peers derive a new Sender Key, Recipient Key and Common IV, as defined in Section 3.2 of [RFC8613]. Also, both peer re-initialize the Sender Sequence Number and the replay window accordingly, as defined in Section 3.2.2 of [RFC8613]. Since the same Master Secret is preserved, forward secrecy is not achieved.

  As defined in Section 5.1.3, the two peers must take additional actions to ensure a safe execution of the OSCORE IDs update procedure.

- In the latter integrated case, the KUDOS initiator (responder) also acts as initiator (responder) for the OSCORE IDs update procedure.

[TODO: think about the possibility of safely preserving ongoing observations following an update of OSCORE IDs alone.]

5.1. The Recipient-ID Option

The Recipient ID Option defined in this section has the properties summarized in Figure 9, which extends Table 4 of [RFC7252]. That is, the option is elective, safe to forward, part of the cache key and non repeatable.

No.	C	U	N	R	Name	Format	Length	Default
TBD1					Recipient-ID	opaque	0-7	(none)

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

Figure 9: The Recipient-ID Option.

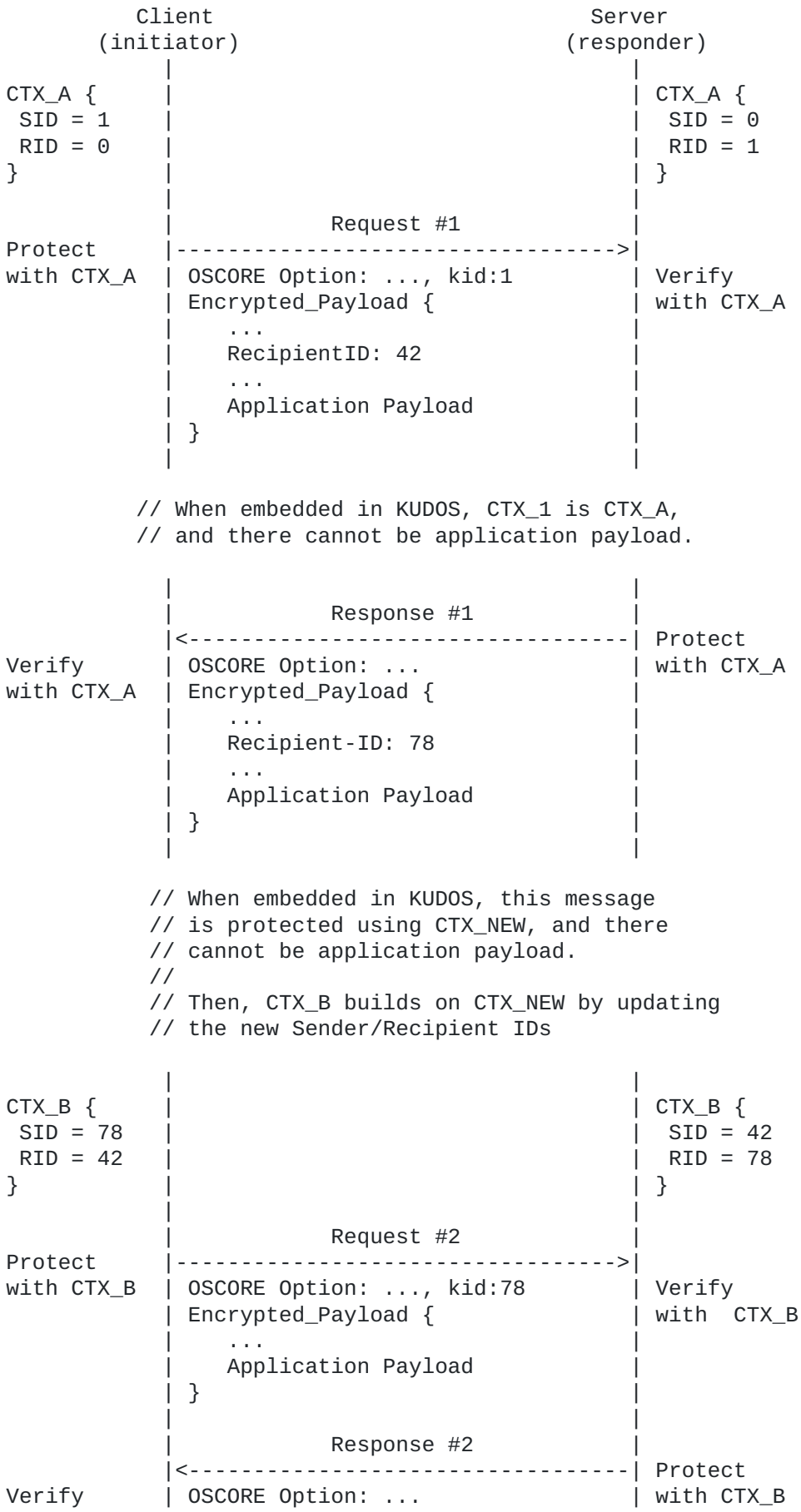
This document particularly defines how this option is used in messages protected with OSCORE. That is, when the option is included in an outgoing message, the option value specifies the new OSCORE Recipient ID that the sender endpoint intends to use with the other endpoint sharing the OSCORE Security Context.

The Recipient-ID Option is of class E in terms of OSCORE processing (see Section 4.1 of [RFC8613]).

5.1.1. Client-Initiated OSCORE IDs Update

Figure 10 shows the stand-alone OSCORE IDs update workflow, with the client acting as initiator.

On each peer, SID and RID denote the OSCORE Sender ID and Recipient ID of that peer, respectively.


Client
(initiator)
Server
(responder)
CTX_A {
SID = 1
RID = 0
}
CTX_A {
SID = 0
RID = 1
}
Request #1
Protect
with CTX_A
OSCORE Option: ..., kid:1
Encrypted_Payload {
...
RecipientID: 42
...
Application Payload
}
Verify
with CTX_A
// When embedded in KUDOS, CTX_1 is CTX_A,
// and there cannot be application payload.
Response #1
Verify
with CTX_A
OSCORE Option: ...
Encrypted_Payload {
...
Recipient-ID: 78
...
Application Payload
}
Protect
with CTX_A
// When embedded in KUDOS, this message
// is protected using CTX_NEW, and there
// cannot be application payload.
//
// Then, CTX_B builds on CTX_NEW by updating
// the new Sender/Recipient IDs
CTX_B {
SID = 78
RID = 42
}
CTX_B {
SID = 42
RID = 78
}
Request #2
Protect
with CTX_B
OSCORE Option: ..., kid:78
Encrypted_Payload {
...
Application Payload
}
Verify
with CTX_B
Response #2
Verify
OSCORE Option: ...
Protect
with CTX_B

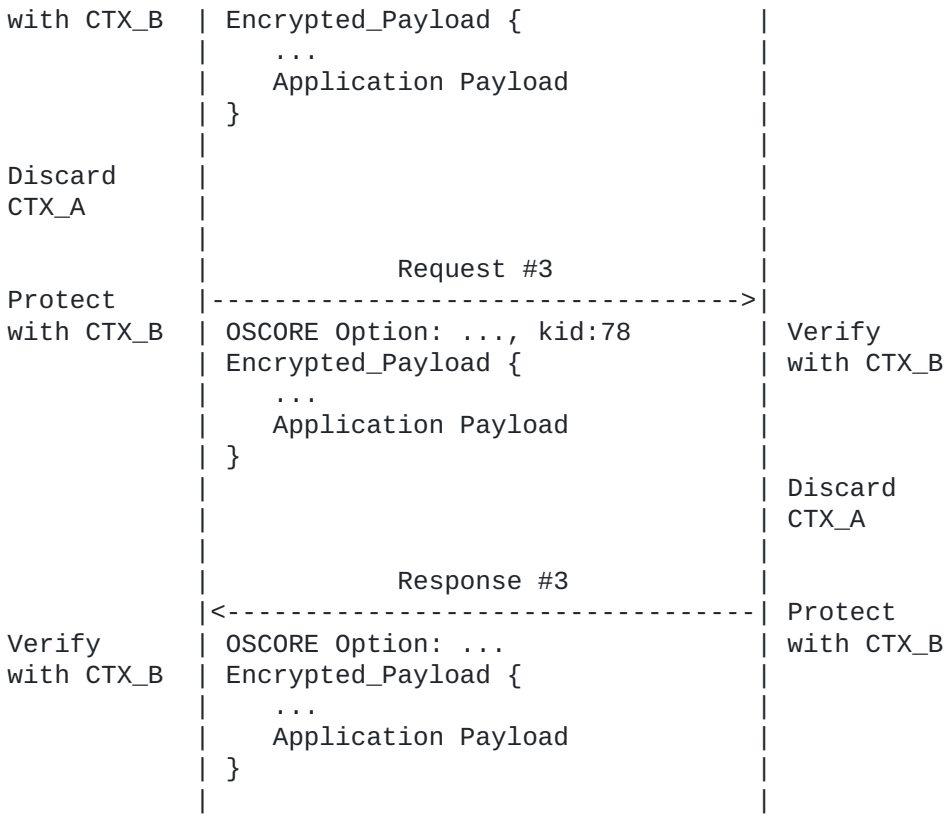

with CTX_B
Encrypted_Payload {
...
Application Payload
}
Discard
CTX_A
Request #3
Protect
with CTX_B
OSCORE Option: ..., kid:78
Encrypted_Payload {
...
Application Payload
}
Verify
with CTX_B
Discard
CTX_A
Response #3
Protect
with CTX_B
Verify
with CTX_B
OSCORE Option: ...
Encrypted_Payload {
...
Application Payload
}

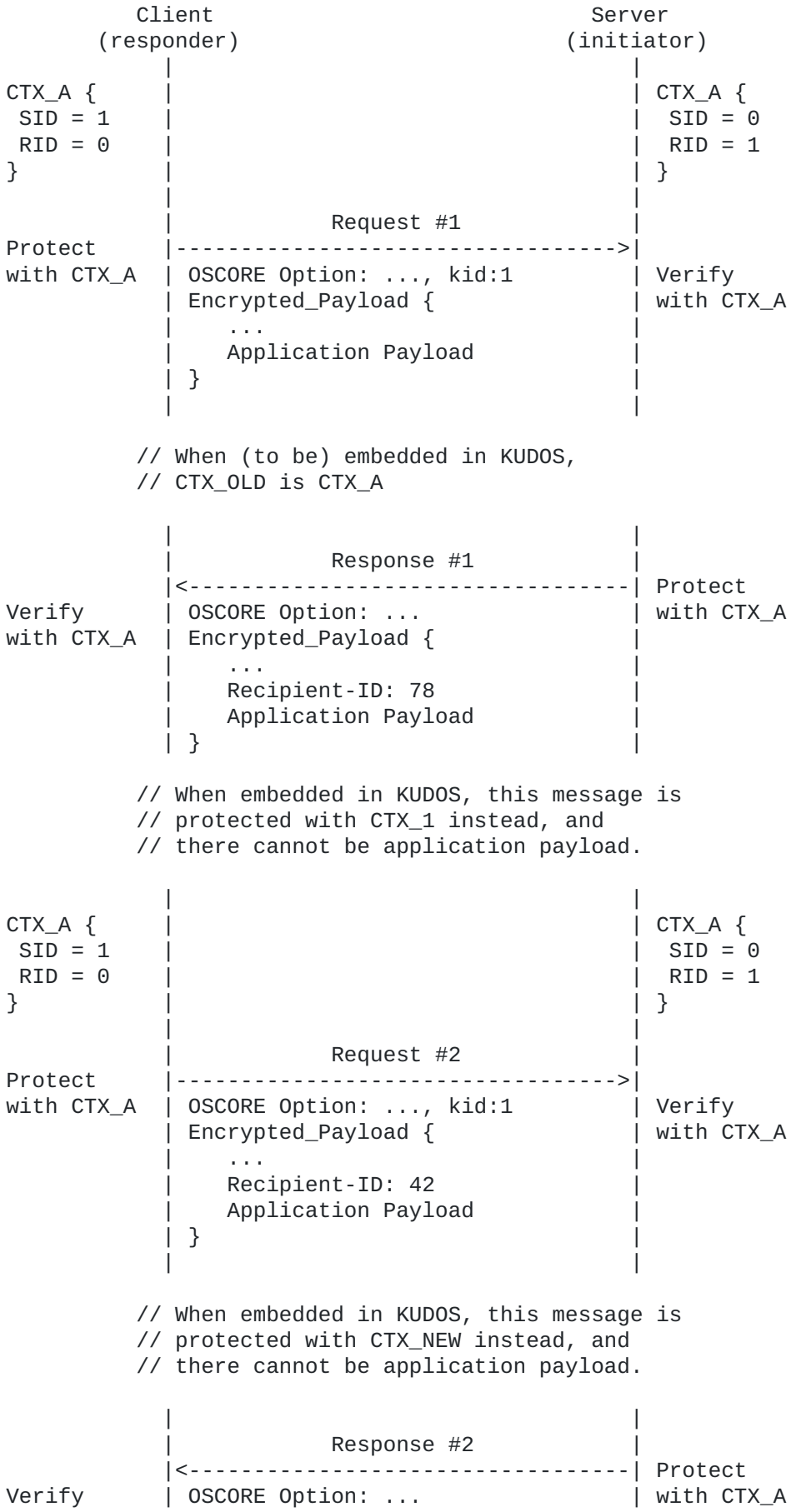
Figure 10: Client-Initiated OSCORE IDs Update Workflow

[TODO: discuss the example]

5.1.2. Server-Initiated OSCORE IDs Update

Figure 11 shows the stand-alone OSCORE IDs update workflow, with the server acting as initiator.

On each peer, SID and RID denote the OSCORE Sender ID and Recipient ID of that peer, respectively.


Client
(responder)
Server
(initiator)
CTX_A {
SID = 1
RID = 0
}
CTX_A {
SID = 0
RID = 1
}
Request #1
Protect
with CTX_A
OSCORE Option: ..., kid:1
Encrypted_Payload {
...
Application Payload
}
Verify
with CTX_A
// When (to be) embedded in KUDOS,
// CTX_OLD is CTX_A
Response #1
Protect
with CTX_A
Verify
with CTX_A
OSCORE Option: ...
Encrypted_Payload {
...
Recipient-ID: 78
Application Payload
}
// When embedded in KUDOS, this message is
// protected with CTX_1 instead, and
// there cannot be application payload.
CTX_A {
SID = 1
RID = 0
}
CTX_A {
SID = 0
RID = 1
}
Request #2
Protect
with CTX_A
OSCORE Option: ..., kid:1
Encrypted_Payload {
...
Recipient-ID: 42
Application Payload
}
Verify
with CTX_A
// When embedded in KUDOS, this message is
// protected with CTX_NEW instead, and
// there cannot be application payload.
Response #2
Protect
with CTX_A
Verify
OSCORE Option: ...

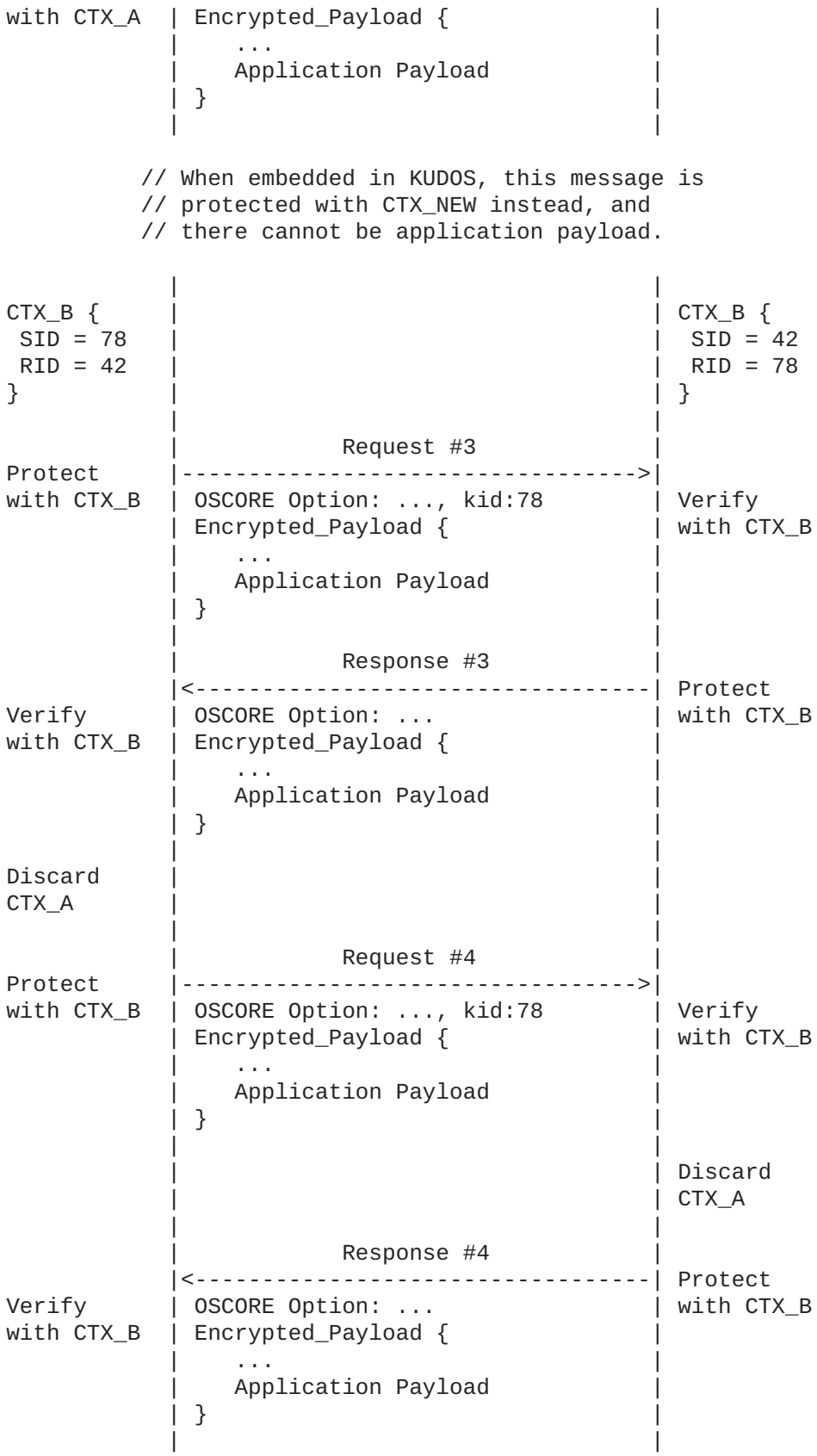

with CTX_A
Encrypted_Payload {
...
Application Payload
}
// When embedded in KUDOS, this message is
// protected with CTX_NEW instead, and
// there cannot be application payload.
CTX_B {
SID = 78
RID = 42
}
CTX_B {
SID = 42
RID = 78
}
Request #3
Protect
with CTX_B
OSCORE Option: ..., kid:78
Encrypted_Payload {
...
Application Payload
}
Verify
with CTX_B
Response #3
Protect
with CTX_B
OSCORE Option: ...
Encrypted_Payload {
...
Application Payload
}
Verify
with CTX_B
Discard
CTX_A
Request #4
Protect
with CTX_B
OSCORE Option: ..., kid:78
Encrypted_Payload {
...
Application Payload
}
Verify
with CTX_B
Discard
CTX_A
Response #4
Protect
with CTX_B
OSCORE Option: ...
Encrypted_Payload {
...
Application Payload
}
Verify
with CTX_B

Figure 11: Server-Initiated OSCORE IDs Update Workflow

[TODO: discuss the example]

5.1.3. Additional Actions for Stand-Alone Execution

After having experienced a loss of state, a peer MUST NOT participate in a stand-alone OSCORE IDs update procedure with another peer, until having performed a full-fledged establishment/renewal of an OSCORE Security Context with the other peer (e.g., by running KUDOS or the EDHOC protocol [I-D.ietf-lake-edhoc]).

More precisely, a peer has experienced a loss of state if it cannot access the latest snapshot of the latest OSCORE Security Context CTX_OLD or the whole set of OSCORE Sender/Recipient IDs that have been used with the triplet (Master Secret, Master Salt, ID Context) of CTX_OLD. This can happen, for instance, after a device reboot.

Furthermore, when participating in a stand-alone OSCORE IDs update procedure, a peer performs the following additional steps.

* When sending an OSCORE IDs update message, the peer MUST specify its new intended OSCORE Recipient ID as value of the Recipient-ID Option only if such a Recipient ID is not only available (see Section 3.3 of [RFC8613], but it has also never been used as Recipient ID with the current triplet (Master Secret, Master Salt, ID Context).

* When receiving an OSCORE IDs update message, the peer MUST abort the procedure if it has already used the identifier specified in the Recipient-ID Option as its own Sender ID with current triplet (Master Secret, Master Salt, ID Context).

In order to fulfill the conditions above, a peer has to keep track of the OSCORE Sender/Recipient IDs that it has used with the current triplet (Master Secret, Master Salt, ID Context) since the latest update of OSCORE Master Secret (e.g, performed by running KUDOS).

6. Security Considerations

This document mainly covers security considerations about using AEAD keys in OSCORE and their usage limits, in addition to the security considerations of [RFC8613].

Depending on the specific key update procedure used to establish a new OSCORE Security Context, the related security considerations also apply.

[TODO: Add more considerations.]

7. IANA Considerations

This document has the following actions for IANA.

Note to RFC Editor: Please replace all occurrences of "[RFC-XXXX]" with the RFC number of this specification and delete this paragraph.

7.1. CoAP Option Numbers Registry

IANA is asked to enter the following option number to the "CoAP Option Numbers" registry within the "CoRE Parameters" registry group.

Number	Name	Reference
TBD	Recipient-ID	[RFC-XXXX]

The number suggested to IANA for the Recipient-ID Option is 24.

7.2. OSCORE Flag Bits Registry

IANA is asked to add the following entries to the "OSCORE Flag Bits" registry within the "Constrained RESTful Environments (CoRE) Parameters" registry group.

Bit Position	Name	Description	Reference
1	Extension-1 Flag	Set to 1 if the OSCORE Option specifies a second byte of OSCORE flag bits	[RFC-XXXX]
15	Nonce Flag	Set to 1 if the compressed COSE object contains 'nonce'	[RFC-XXXX]

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-15, 10 July 2022, <https://www.ietf.org/archive/id/draft-ietf-lake-edhoc-15.txt>.

[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>.

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

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

[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>.

[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>.

8.2. Informative References

[I-D.ietf-ace-oauth-authz] Seitz, L., Selander, G., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework (ACE-OAuth)", Work in Progress, Internet-Draft, draft-ietf-ace-oauth-authz-46, 8 November 2021, <https://www.ietf.org/archive/id/draft-ietf-ace-oauth-authz-46.txt>.

[I-D.ietf-ace-oscore-profile] Palombini, F., Seitz, L., Selander, G., and M. Gunnarsson, "OSCORE Profile of the Authentication and Authorization for Constrained Environments Framework", Work in Progress, Internet-Draft, draft-ietf-ace-oscore-profile-19, 6 May 2021, <https://www.ietf.org/archive/id/draft-ietf-ace-oscore-profile-19.txt>.

[I-D.irtf-cfrg-aead-limits] Günther, F., Thomson, M., and C. A. Wood, "Usage Limits on AEAD Algorithms", Work in Progress, Internet-Draft, draft-irtf-cfrg-aead-limits-04, 7 March 2022, <https://www.ietf.org/archive/id/draft-irtf-cfrg-aead-limits-04.txt>.

[LwM2M] Open Mobile Alliance, "Lightweight Machine to Machine Technical Specification - Core, Approved Version 1.2, OMA-TS-LightweightM2M_Core-V1_2-20201110-A", November 2020, <http://www.openmobilealliance.org/release/LightweightM2M/V1_2-20201110-A/OMA-TS-LightweightM2M_Core-V1_2-20201110-A.pdf>.

[LwM2M-Transport] Open Mobile Alliance, "Lightweight Machine to Machine Technical Specification - Transport Bindings, Approved Version 1.2, OMA-TS-LightweightM2M_Transport-V1_2-20201110-A", November 2020, <http://www.openmobilealliance.org/release/LightweightM2M/V1_2-20201110-A/OMA-TS-LightweightM2M_Transport-V1_2-20201110-A.pdf>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/info/rfc7519>.

[RFC7554] Watteyne, T., Ed., Palattella, M., and L. Grieco, "Using IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH) in the Internet of Things (IoT): Problem Statement", RFC 7554, DOI 10.17487/RFC7554, May 2015, <https://www.rfc-editor.org/info/rfc7554>.

[RFC8180] Vilajosana, X., Ed., Pister, K., and T. Watteyne, "Minimal IPv6 over the TSCH Mode of IEEE 802.15.4e (6TiSCH) Configuration", BCP 210, RFC 8180, DOI 10.17487/RFC8180, May 2017, <https://www.rfc-editor.org/info/rfc8180>.

[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>.

[RFC9031] Vučinić, 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. Detailed considerations for AEAD_AES_128_CCM_8

For the AEAD_AES_128_CCM_8 algorithm when used as AEAD Algorithm for OSCORE, larger IA and CA values are achieved, depending on the value of 'q', 'v' and 'l'. Figure 12 shows the resulting IA and CA probabilities enjoyed by AEAD_AES_128_CCM_8, when taking different values of 'q', 'v' and 'l' as input to the formulas defined in [I-D.irtf-cfrg-aead-limits].

As shown in Figure 12, it is especially possible to achieve the lowest IA = 2^-50 and a good CA = 2^-70 by considering the largest possible value of the (q, v, l) triplet equal to (2^20, 2^10, 2^8), while still keeping a good security level. Note that the value of 'l' does not impact on IA, while CA displays good values for every considered value of 'l'.

'q', 'v' and 'l'	IA probability	CA probability
q=2^20, v=2^20, l=2^8	2^-44	2^-70
q=2^15, v=2^20, l=2^8	2^-44	2^-80
q=2^10, v=2^20, l=2^8	2^-44	2^-90
q=2^20, v=2^15, l=2^8	2^-49	2^-70
q=2^15, v=2^15, l=2^8	2^-49	2^-80
q=2^10, v=2^15, l=2^8	2^-49	2^-90
q=2^20, v=2^14, l=2^8	2^-50	2^-70
q=2^15, v=2^14, l=2^8	2^-50	2^-80
q=2^10, v=2^14, l=2^8	2^-50	2^-90
q=2^20, v=2^10, l=2^8	2^-54	2^-70
q=2^15, v=2^10, l=2^8	2^-54	2^-80
q=2^10, v=2^10, l=2^8	2^-54	2^-90
q=2^20, v=2^20, l=2^6	2^-44	2^-74
q=2^15, v=2^20, l=2^6	2^-44	2^-84
q=2^10, v=2^20, l=2^6	2^-44	2^-94
q=2^20, v=2^15, l=2^6	2^-49	2^-74
q=2^15, v=2^15, l=2^6	2^-49	2^-84
q=2^10, v=2^15, l=2^6	2^-49	2^-94
q=2^20, v=2^14, l=2^6	2^-50	2^-74
q=2^15, v=2^14, l=2^6	2^-50	2^-84
q=2^10, v=2^14, l=2^6	2^-50	2^-94
q=2^20, v=2^10, l=2^6	2^-54	2^-74
q=2^15, v=2^10, l=2^6	2^-54	2^-84
q=2^10, v=2^10, l=2^6	2^-54	2^-94

Figure 12: Probabilities for AEAD_AES_128_CCM_8 based on chosen q, v and l values.

Appendix B. Estimation of 'count_q'

This section defines a method to compute an estimate of the counter 'count_q' (see Section 2.2.2), hence not requiring a peer to store it in its own Sender Context.

This method relies on the fact that, at any point in time, a peer has performed *at most* ENC = (SSN + SSN*) encryptions using its own Sender Key, where:

- *SSN is the current value of this peer's Sender Sequence Number.
- *SSN* is the current value of other peer's Sender Sequence Number. That is, SSN* is an overestimation of the responses without Partial IV that this peer has sent.

Thus, when protecting an outgoing message (see Section 2.3.1), the peer aborts the message processing if the estimated est_q > limit_q, where est_q = (SSN + X) and X is determined as follows.

- *If the outgoing message is a response, X is the Partial IV specified in the corresponding request that this peer is responding to. Note that X < SSN* always holds.

If the outgoing message is a request, X is the highest Partial IV value marked as received in this peer's Replay Window plus 1, or 0 if it has not accepted any protected message from the other peer yet. That is, X is the highest Partial IV specified in message received from the other peer, i.e., the highest seen Sender Sequence Number of the other peer. Note that, also in this case, X < SSN always holds.

Appendix C. Document Updates

RFC EDITOR: PLEASE REMOVE THIS SECTION.

C.1. Version -01 to -02

*Extended terminology.

*Moved procedure for preserving observations across key updates to main body.

*Moved procedure to update OSCORE Sender/Recipient IDs to main body.

*Moved key update without forward secrecy section to main body.

*Define signaling bits present in the 'x' byte.

*Modifications and alignment of updateCtx() with EDHOC.

*Rules for deletion of old EDHOC keys PRK_out and PRK_exporter.

*Describe CBOR wrapping of involved nonces with examples.

*Renamed 'id detail' to 'nonce'.

*Editorial improvements.

C.2. Version -00 to -01

*Recommendation on limits for CCM_8. Details in Appendix.

*Improved message processing, also covering corner cases.

*Example of method to estimate and not store 'count_q'.

*Added procedure to update OSCORE Sender/Recipient IDs.

*Added method for preserving observations across key updates.

*Added key update without forward secrecy.

Acknowledgments

The authors sincerely thank Christian Amsüss, John Preuß Mattsson and Göran Selander for their feedback and comments.

The work on this document has been partly supported by VINNOVA and the Celtic-Next project CRITISEC; and by the H2020 project SIFIS-Home (Grant agreement 952652).

Authors' Addresses

Rikard Höglund
RISE AB
Isafjordsgatan 22
SE-16440 Stockholm Kista
Sweden

Email: rikard.hoglund@ri.se

Marco Tiloca
RISE AB
Isafjordsgatan 22
SE-16440 Stockholm Kista
Sweden

Email: marco.tiloca@ri.se