Workgroup: CoRE Working Group
Internet-Draft:
draft-ietf-core-oscore-key-update-07
Updates: 8613 (if approved)
Published: 4 March 2024
Intended Status: Standards Track
Expires: 5 September 2024
Authors: R. Höglund M. Tiloca
RISE AB RISE AB

Key Update for OSCORE (KUDOS)

Abstract

This document defines Key Update for OSCORE (KUDOS), a lightweight procedure that two CoAP endpoints can use to update their keying material by establishing a new OSCORE Security Context. Accordingly, it updates the use of the OSCORE flag bits in the CoAP OSCORE Option as well as the protection of CoAP response messages with OSCORE, and it deprecates the key update procedure specified in Appendix B.2 of RFC 8613. Thus, this document updates RFC 8613. Also, this document defines a procedure that two endpoints can use to update their OSCORE identifiers, run either stand-alone or during a KUDOS execution.

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

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 5 September 2024.

Copyright Notice

Copyright (c) 2024 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction
  1.1. Terminology
2. Current Methods for Rekeying OSCORE
3. Updated Protection of Responses with 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. Forward Message Flow
    4.3.2. Reverse Message Flow
  4.4. Avoiding Deadlocks
    4.4.1. Scenario 1
    4.4.2. Scenario 2
    4.4.3. Scenario 3
  4.5. Key Update with or without Forward Secrecy
    4.5.1. Handling and Use of Keying Material
    4.5.2. Selection of KUDOS Mode
  4.6. Preserving Observations Across Key Updates
    4.6.1. Management of Observations
  4.7. Retention Policies
  4.8. Discussion
    4.8.1. KUDOS Interleaved with Other Message Exchanges
    4.8.2. Communication Overhead
    4.8.3. Well-Known KUDOS Resource
    4.8.4. Rekeying when Using SCHC with OSCORE
  4.9. Signaling KUDOS support in EDHOC
5. Security Considerations
6. IANA Considerations
  6.1. OSCORE Flag Bits Registry
  6.2. EDHOC External Authorization Data Registry

6.3. The Well-Known URI Registry
6.4. Resource Type (rt=) Link Target Attribute Values Registry
7. References
7.1. Normative References
7.2. Informative References
Appendix A. Forward Message Flow using two CoAP Requests
Appendix B. Forward Message Flow with Response #1 unrelated to Request #1
Appendix C. Forward Message Flow Targeting a non-KUDOS Resource at Server
Appendix D. Document Updates
D.1. Version -06 to -07
D.2. Version -05 to -06
D.3. Version -04 to -05
D.4. Version -03 to -04
D.5. Version -02 to -03
D.6. Version -01 to -02
D.7. 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.

To ensure secure communication when using OSCORE, peers may need to update their shared keying material. Among other reasons, approaching key usage limits [I-D.irtf-cfrg-aead-limits] [I-D.ietf-core-oscore-key-limits] requires updating the OSCORE keying material before communications can securely continue.

This document updates [RFC8613] as follows.

* 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].

* With reference to the "OSCORE Flag Bits" registry defined in Section 13.7 of [RFC8613] as part of the "Constrained RESTful Environments (CoRE) Parameters" registry group, it updates the entries with Bit Position 0 and 1 (see Section 6), both originally marked as "Reserved". That is, it defines and registers the usage of the OSCORE flag bit with Bit Position 0,

as the one intended to expand the space for the OSCORE flag bits in the OSCORE Option (see Section 4.1). Also, it marks the bit with Bit Position of 1 as "Unassigned".

* It updates the protection of CoAP responses with OSCORE originally specified in Section 8.3 of [RFC8613], as defined in Section 3 of this document.

Furthermore, this document specifies a method that two peers can use to update their OSCORE identifiers. This can be run as a stand-alone procedure, or instead integrated 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 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.

* Forward message flow: the KUDOS execution workflow where the initiator acts as CoAP client (see Section 4.3.1).

* Reverse message flow: the KUDOS execution workflow where the initiator acts as CoAP server (see Section 4.3.2).

* 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 4.5).

2. Current Methods for Rekeying OSCORE

Two peers communicating using OSCORE may choose to renew their shared keying information by establishing a new OSCORE Security Context for a variety of reasons. A particular reason is approaching limits set for safe key usage [I-D.ietf-core-oscore-key-limits].

Practically, when the relevant limits have 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. That is, the two peers have to establish new Sender and Recipient Keys, as the keys actually used by the AEAD algorithm.

In addition to approaching the key usage limits, there may be other reasons for a peer to initiate a key update procedure. These include: the OSCORE Security Context approaching its expiration time; application policies prescribing a regular key rollover; approaching the exhaustion of the Sender Sequence Number space in the OSCORE Sender Context.

It is RECOMMENDED that the peer initiating the key update procedure starts it with some margin, i.e., well before actually experiencing the trigger event forcing to perform a key update, e.g., the OSCORE Security Context expiration or the exhaustion of the Sender Sequence Number space. If the rekeying is not initiated ahead of these events, it may become practically impossible to perform a key update with certain methods.

Other specifications define a number of ways for rekeying OSCORE, 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 [RFC9203] of the Authentication and Authorization for Constrained Environments (ACE) Framework [RFC9200].

  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.

  EDHOC also specifies an optional function, EDHOC_KeyUpdate, to perform a key update in a more efficient way than re-running EDHOC. The two communicating peers call EDHOC_KeyUpdate with equivalent input, which results in derivation of a new shared pseudo-random key. Usage of EDHOC_KeyUpdate preserves forward secrecy.

  Note that EDHOC may be run standalone or as part of other workflows, such as when using the EDHOC and OSCORE profile of ACE [I-D.ietf-ace-edhoc-oscore-profile].

* 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.

3. Updated Protection of Responses with OSCORE

The protection of CoAP responses with OSCORE is updated, by adding the following text at the end of step 3 of Section 8.3 of [RFC8613].

> If the server is using a different Security Context for the response compared to what was used to verify the request (e.g., due to an occurred key update), then the server MUST take the second alternative. That is, the server MUST include its Sender Sequence Number as Partial IV in the response and use it to build the AEAD nonce to protect the response.
>
> This prevents the server from using the same AEAD (key, nonce) pair for two responses, protected with different OSCORE Security Contexts. An exception is the procedure in Appendix B.2 of [RFC8613], which is secure although not complying with the above.

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.

In order to run KUDOS, two peers perform a message exchange of OSCORE-protected CoAP messages. This 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 4.5, 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 has the following properties.

* KUDOS can be initiated by either peer. In particular, the CoAP client or the CoAP server may start KUDOS by sending the first rekeying message, by running KUDOS in the forward message flow Section 4.3 or reverse message flow Section 4.3.2, respectively. A peer that supports KUDOS MUST support both the forward message flow and the reverse message flow.

* The new OSCORE Security Context enjoys forward secrecy, unless KUDOS is run in no-FS mode (see Section 4.5).

* 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 by exchanging two CoAP messages. The two peers achieve mutual key confirmation 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 eight least significant bit, called "Extension-1 Flag", in the first byte of the OSCORE Option containing the OSCORE flag bits. The registration of this flag bit in the "OSCORE Flag Bits" registry is specified in Section 6.1.

  When the Extension-1 Flag is set to 1, the second byte of the OSCORE Option MUST include the 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 6.1.

  When it is set to 1, the compressed COSE object contains a field 'x' and a field 'nonce', to be used for the steps defined in Section 4.3. In particular, the 1 byte 'x' following 'kid context' (if any) encodes the size of the following field 'nonce', together with signaling bits that indicate the specific behavior to adopt during the KUDOS execution.

Hereafter, a message is referred to as a "KUDOS (request/response) message", if and only if the second byte of flags is present and the 'd' bit is set to 1. If that is not the case, the message is referred to as a "non KUDOS (request/response) message".

The encoding of 'x' is as follows:

- The four least significant bits encode the 'nonce' size 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 4.5.

- 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 4.6.

- The seventh least significant bit is the 'z' bit. When it is set to 1, the compressed COSE object contains a field 'y' and a field 'old_nonce', to be used for the steps defined in Section 4.3. In particular, the 1 byte 'y' following 'nonce' encodes the size of the following field 'old_nonce'. This bit SHALL only be set in the second KUDOS message and only if it is a CoAP request. For an example see the execution of KUDOS in the reverse message flow shown in Figure 6.

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

The encoding of 'y' is as follows:

- The four least significant bits of the 'y' byte encode the 'old_nonce' size in bytes minus 1, namely 'w'.

- The fifth to seventh least significant bits SHALL be set to zero when not in use. According to this specification, if 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]

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

*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 1 shows extended OSCORE Option value, with the possible presence of 'nonce' and 'old_nonce'.

```
 0 1 2 3 4 5 6 7   8   9   10  11  12  13  14  15 <----- n bytes ----->
+-+-+-+-+-+-+-+-+---+---+---+---+---+---+---+---+---------------------+
|1|0|0|h|k|  n  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | d | Partial IV (if any) |
+-+-+-+-+-+-+-+-+---+---+---+---+---+---+---+---+---------------------+


 <- 1 byte -> <----- s bytes ------> <- 1 byte -> <--- m + 1 bytes --->
+------------+----------------------+-----------+--------------------+
| s (if any) | kid context (if any) | x (if any) | nonce (if any)     |
+------------+----------------------+-----------+--------------------+
                                   /             \____
                                  /                   |
                                 /   0 1 2 3 4 5 6 7  |
                                 |  +-+-+-+-+-+-+-+-+ |
                                 |  |0|z|b|p|   m   | |
                                 |  +-+-+-+-+-+-+-+-+ |

    <- 1 byte -> <--- w + 1 bytes --->
   +------------+--------------------+------------------+
   | y (if any) | old_nonce (if any)  | kid (if any) ... |
   +------------+--------------------+------------------+
  /             \____
 /                   |
/   0 1 2 3 4 5 6 7  |
|  +-+-+-+-+-+-+-+-+ |
|  |0|0|0|0|   w   | |
|  +-+-+-+-+-+-+-+-+ |
```

Figure 1: The extended OSCORE Option value, with the possible presence of 'nonce' and 'old_nonce'

4.2. Function for Security Context Update

The updateCtx() function shown in Figure 2 takes as input the 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 4.1), 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 X_N, as 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. In particular, the new Master Secret is derived through a KUDOS-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.

The definition of KUDOS-Expand depends on the key derivation function used for OSCORE by the two peers, as specified in CTX_IN. either peer If the key derivation function is an HKDF Algorithm (see Section 3.1 of [RFC8613]), then KUDOS-Expand is mapped to HKDF-Expand [RFC5869], as shown below. Also, the hash algorithm is the same one used by the HKDF Algorithm specified in CTX_IN.

```
KUDOS-Expand(CTX_IN.MasterSecret, ExpandLabel, oscore_key_length) =
   HKDF-Expand(CTX_IN.MasterSecret, ExpandLabel, oscore_key_length)
```

If a future specification updates [RFC8613] by admitting different key derivation functions than HKDF Algorithms (e.g., KMAC as based on the SHAKE128 or SHAKE256 hash functions), that specification has to update also the present document in order to define the mapping between such key derivation functions and KUDOS-Expand.

When an HKDF Algorithm is used, 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.

```
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 = X_cbor | N_cbor

  oscore_key_length = < Size of CTX_IN.MasterSecret in bytes >

  Label = "key update"

  MSECRET_NEW = KUDOS-Expand-Label(CTX_IN.MasterSecret, Label,
                                   X_N, oscore_key_length)
              = KUDOS-Expand(CTX_IN.MasterSecret, ExpandLabel,
                             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 ExpandLabel is defined as

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

Figure 2: 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. A peer may want to run

KUDOS for a variety of reasons, including expiration of the OSCORE Security Context, approaching limits for key usage [I-D.ietf-core-oscore-key-limits], application policies, and imminent exhaustion of the OSCORE Sender Sequence Number space. The expiration time of an OSCORE Security Context and the key usage limits are hard limits, at which point a peer MUST stop using the keying material in the OSCORE Security Context and has to perform a rekeying before resuming secure communication with the other peer. However, KUDOS can also be used for active rekeying, and a peer may run the KUDOS procedure at any point in time and for any reason.

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. From the newly derived Security Context CTX_NEW, the peers MUST store to non-volatile memory the immutable parts of the OSCORE Security Context as specified in Section 3.1 of [RFC8613], with the possible exception of the Common IV, Sender Key, and Recipient Key that can be derived again when needed, as specified in Section 3.2.1 of [RFC8613]. If the peer is unable to write in non-volatile memory, the two peers have to run KUDOS in its stateless no-FS mode (see Section 4.5).

When running KUDOS, each peer contributes by generating a nonce value N1 or N2, and providing it to the other peer. The size of the nonces N1 and N2 is application specific, and the use of 8 byte nonce values is RECOMMENDED. The nonces N1 and N2 SHOULD be random values. An exception is described later in Section Section 4.5.1.

Furthermore, X1 and X2 are the value of the 'x' byte specified in the OSCORE Option of the first and second KUDOS message, respectively. The X1 and X2 values are calculated by the sender peer based on: the length of nonce N1 and N2, specified in the 'nonce' field of the OSCORE Option of the first and second KUDOS message, respectively; as well as on the specific settings the peer wishes to run KUDOS with. 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]).

After a peer has generated or received the value N1, and after a peer has calculated or received the value X1, it shall retain these in memory until it has received and processed the second KUDOS message.

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, and MUST NOT use the originally shared OSCORE Security Context CTX_OLD for protecting outgoing messages. Once CTX_NEW has been derived, a peer deletes any OSCORE Security Context CTX_DEL older than CTX_OLD, such that both CTX_DEL and CTX_OLD have the same ID_CONTEXT or no ID Context. This can for instance occur in the forward message flow when the initiator has just received KUDOS Response #1 and immediately starts KUDOS again as initiator, before sending any non KUDOS messages which would give the responder key confirmation and allow it to safely discard CTX_OLD.

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 4.6. 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.

The peer starting the KUDOS execution is denoted as initiator, while the other peer is denoted as responder.

KUDOS may run with the initiator acting either as CoAP client or CoAP server. The former case is denoted as the "forward message flow" (see Section 4.3.1) and the latter as the "reverse message flow" (see Section 4.3.2). The following properties hold for both the forward and reverse message flow.

* The initiator always offers the fresh value N1.

* The responder always offers the fresh value N2

* The responder is always the first one deriving CTX_NEW.

* The initiator is always the first one achieving key confirmation, hence the first one able to safely discard CTX_OLD.
* Both the initiator and the responder use and preserve the same respective OSCORE Sender ID and Recipient ID. Also, if CTX_OLD specifies an OSCORE ID Context, both peers use and preserve the same OSCORE ID Context.

If a KUDOS message is a CoAP request, then it can target two different types of resources at the recipient CoAP server:

* The well-known KUDOS resource at /.well-known/kudos, or an alternative KUDOS resource with resource type "core.kudos" (see Sections Section 4.8.3 and Section 6.4). In such a case, no application processing is expected at the CoAP server, and the plain CoAP request composed before OSCORE protection should not include an application payload.
* A non-KUDOS resource, i.e., an actual application resource that a CoAP request can target in order to trigger application processing at the CoAP server. In such a case, the plain CoAP request composed before OSCORE protection may include an application payload, if admitted by the request method.

Similarly, any CoAP response can also be a KUDOS message. If the corresponding CoAP request has targeted a KUDOS resource, then the plain CoAP response composed before OSCORE encryption should not include an application payload. Otherwise, an application payload may be included.

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 CTX_NEW. The responder completes the key update process when sending the second KUDOS message, as protected with CTX_NEW.

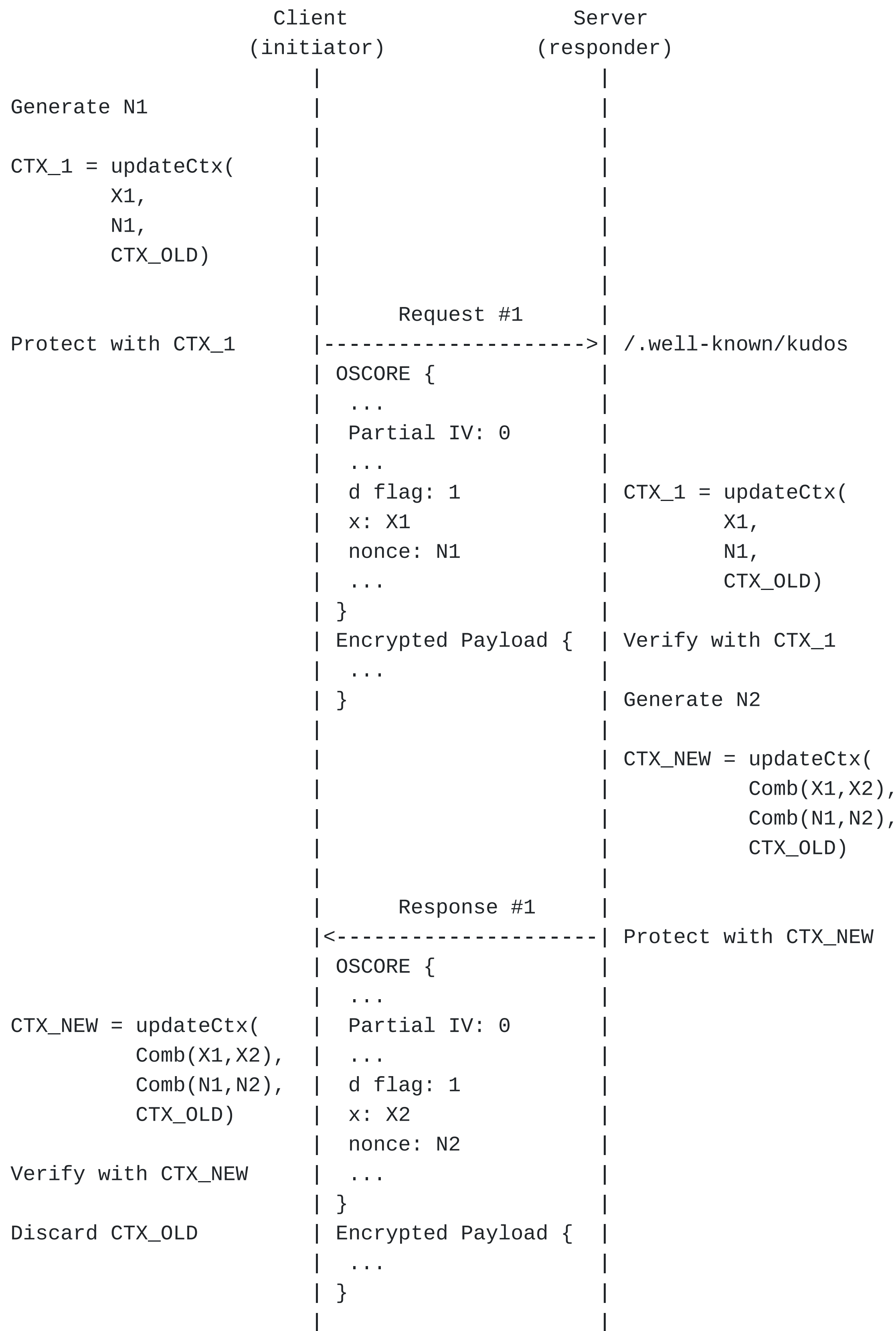
In particular, CTX_OLD is the most recent OSCORE Security Context that a peer has with a given ID Context or without ID Context, before initiating the KUDOS procedure or upon having received and successfully verified the first KUDOS message. In turn, CTX_NEW is the most recent OSCORE Security Context that a peer has with a given ID Context or without ID Context, before sending the second KUDOS message or upon having received and successfully verified the second KUDOS message.

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. Forward Message Flow

Figure 3 shows an example of KUDOS run in the forward message flow, with the client acting as KUDOS initiator. Even though in this example the first KUDOS message is a request and the second is a response, KUDOS is not constrained to this request/response model and a KUDOS execution can be performed with any combination of CoAP requests and responses. Appendix A shows an example where both KUDOS messages are CoAP requests. Furthermore, Appendix B presents an example where KUDOS Response #2 is a response to a different request than KUDOS Request #1.




```
// The actual key update process ends here.
// The two peers can use the new Security Context CTX_NEW.
```

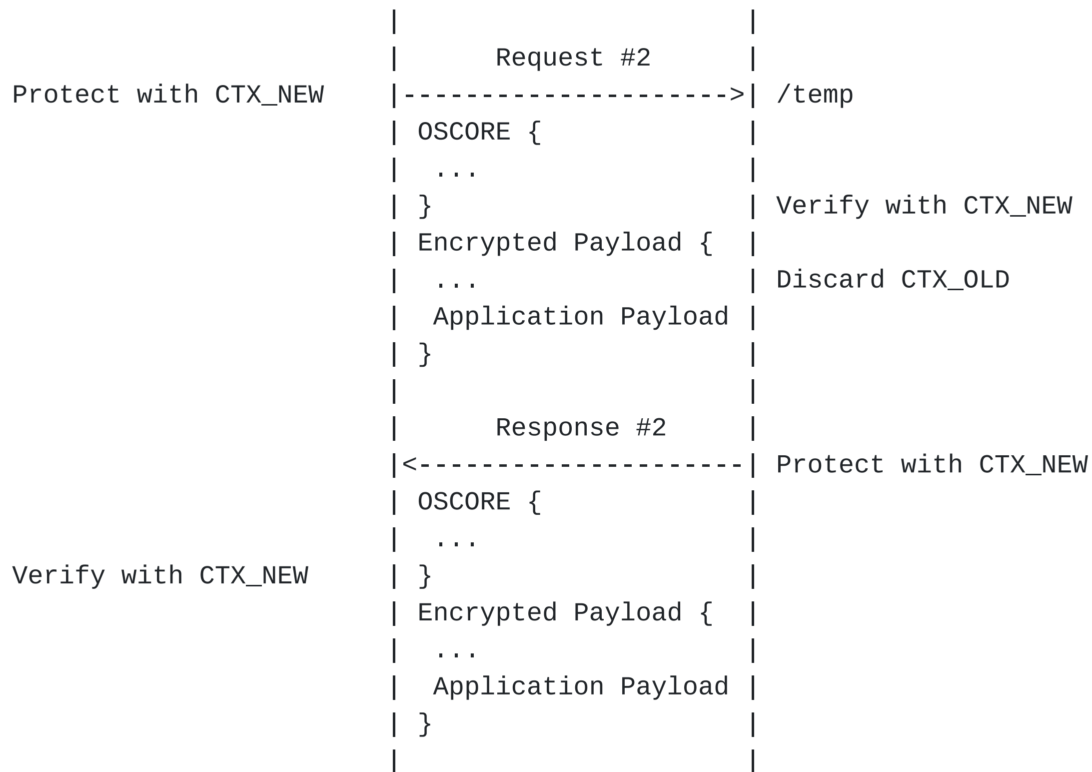


Figure 3: Example of the KUDOS forward message flow.

First, the client generates a 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.

Then, the client prepares a CoAP request targeting the well-known KUDOS resource (see Section 4.8.3) at "/.well-known/kudos". The client protects this CoAP request using CTX_1 and sends it to the server. When the client protects this request using OSCORE, it MUST use 0 as the value of Partial IV. 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 OSCORE Option, the value X1 from the 'x' byte of the OSCORE Option, and provides the updateCtx() function with the input N = N1, X = X1, and CTX_OLD, in order to derive the temporary Security Context CTX_1.

Figure 4 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 = 0x07
N1 = 0x018a278f7faab55a

updateCtx() is called with
X = 0x07
N = 0x018a278f7faab55a

In updateCtx(), X_cbor and N_cbor are built as CBOR byte strings
X_cbor = 0x4107               (h'07')
N_cbor = 0x48018a278f7faab55a (h'018a278f7faab55a')

In updateCtx(), X_N is the byte concatenation of X_cbor and N_cbor
X_N = 0x410748018a278f7faab55a
```

Figure 4: Example of X, N, and X_N when processing the first KUDOS message

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

After that, the server generates a value N2, and uses N = Comb(N1, N2) and X = Comb(X1, X2) together with 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 5.

```
X1, X2, N1, and N2 expressed as raw values
X1 = 0x07
X2 = 0x07
N1 = 0x018a278f7faab55a
N2 = 0x25a8991cd700ac01

X1, X2, N1, and N2 as CBOR byte strings
X1 = 0x4107 (h'07')
X2 = 0x4107 (h'07')
N1 = 0x48018a278f7faab55a (h'018a278f7faab55a')
N2 = 0x4825a8991cd700ac01 (h'25a8991cd700ac01')

updateCtx() is called with
X = 0x41074107
N = 0x48018a278f7faab55a4825a8991cd700ac01

In updateCtx(), X_cbor and N_cbor are built as CBOR byte strings
X_cbor = 0x4441074107 (h'41074107')
N_cbor = 0x5248018a278f7faab55a4825a8991cd700ac01
         (h'48018a278f7faab55a4825a8991cd700ac01')

In updateCtx(), X_N is the byte concatenation of X_cbor and N_cbor
X_N = 0x44410741075248018a278f7faab55a4825a8991cd700ac01
```

Figure 5: Example of X, N, and X_N when processing the second KUDOS message

Then, the server sends an OSCORE response to the client, protected with CTX_NEW. In particular, the response has the 'd' flag bit set to 1 and specifies N2 as 'nonce'. Consistently with Section 3, the server includes its Sender Sequence Number as Partial IV in the response. After that, the server deletes CTX_1.

Upon receiving the OSCORE response, the client retrieves the value N2 from the 'nonce' field of the OSCORE Option, and the value X2 from the 'x' byte of the OSCORE Option. Since the client has received a response to an OSCORE request that 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 CTX_OLD, in order to derive CTX_NEW. Finally, the client verifies the response by using CTX_NEW and deletes CTX_OLD.

From then on, the two peers can protect their message exchanges by using CTX_NEW. As soon as the server successfully verifies an incoming message protected with CTX_NEW, the server deletes CTX_OLD.

In the example in Figure 3, the client takes the initiative and sends a new OSCORE request protected with CTX_NEW.

In case the server does not successfully verify the request, the same error handling specified in Section 8.2 of [RFC8613] applies. This does not result in deleting CTX_NEW. If the server successfully verifies the request using CTX_NEW, the server deletes CTX_OLD and can reply with an OSCORE response protected with CTX_NEW.

Note that the server achieves key confirmation only when receiving a message from the client as protected with 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 CTX_NEW and start a new KUDOS execution acting as CoAP client, i.e., as initiator in the forward message flow.

Also note that, if both peers reboot simultaneously, they will run the KUDOS forward message flow as defined in this section. That is, one of the two peers implementing a CoAP client will send KUDOS Request #1 in Figure 3.

In case the KUDOS message Request #1 in Figure 3 targets a non-KUDOS resource and the application at the server requires freshness for the received requests, then the server does not deliver the request to the application even if the request has been succesfully verified, and the following KUDOS message (i.e., Response #1 in Figure 3) MUST be a 4.01 (Unauthorized) error response.

Upon receiving the 4.01 (Unauthorized) error response as the second KUDOS message Response #1, the client processes it like described above. After successfully completing the KUDOS execution, the client can send to the server a non-KUDOS request protected with CTX_NEW (i.e., Request #2 in Figure 3). Presumably, this request targets the same resource targeted by the previous Request #1, as the same application request or a different one, if the application permits it. Upon receiving, decrypting, and successfully verifying this request protected with CTX_NEW, the server asserts the request as fresh, leveraging the recent establishment of CTX_NEW.

An example of a KUDOS execution where Request #1 targets a non-KUDOS resource is shown in Appendix C.

4.3.1.1. Avoiding In-Transit Requests During a Key Update

Before sending the KUDOS message Request #1 in Figure 3, 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 CTX_NEW from cryptographically matching 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.

During an ongoing KUDOS execution the client MUST NOT send any non-KUDOS requests to the server, even when NSTART is greater than 1 (see Section 4.7 of [RFC7252]).

4.3.2. Reverse Message Flow

Figure 6 shows an example of KUDOS run in the reverse message flow, with the server acting as initiator.

```
                   Client                  Server
                (responder)              (initiator)
                        |                      |
                        |      Request #1      |
Protect with CTX_OLD    |--------------------->| /temp
                        | OSCORE {             |
                        |  ...                 |
                        | }                    | Verify with CTX_OLD
                        | Encrypted Payload {  |
                        |  ...                 | Generate N1
                        |  Application Payload |
                        | }                    | CTX_1 = updateCtx(
                        |                      |         X1,
                        |                      |         N1,
                        |                      |         CTX_OLD)
                        |                      |
                        |      Response #1     |
                        |<---------------------| Protect with CTX_1
                        | OSCORE {             |
                        |  ...                 |
CTX_1 = updateCtx(      |  Partial IV: 0       |
        X1,             |  ...                 |
        N1,             |  d flag: 1           |
        CTX_OLD)        |  x: X1               |
                        |  nonce: N1           |
Verify with CTX_1       |  ...                 |
                        | }                    |
Generate N2             | Encrypted Payload {  |
                        |  ...                 |
CTX_NEW = updateCtx(    | }                    |
          Comb(X1,X2),  |                      |
          Comb(N1,N2),  |                      |
          CTX_OLD)      |                      |
                        |                      |
                        |      Request #2      |
Protect with CTX_NEW    |--------------------->| /.well-known/kudos
                        | OSCORE {             |
                        |  ...                 |
                        |  d flag: 1           | CTX_NEW = updateCtx(
                        |  x: X2               |           Comb(X1,X2),
                        |  nonce: N2           |           Comb(N1,N2),
                        |  y: w                |           CTX_OLD)
                        |  old_nonce: N1       |
                        |  ...                 |
                        | }                    |
                        | Encrypted Payload {  | Verify with CTX_NEW
                        |  ...                 |
                        |  Application Payload |
                        | }                    | Discard CTX_OLD
```

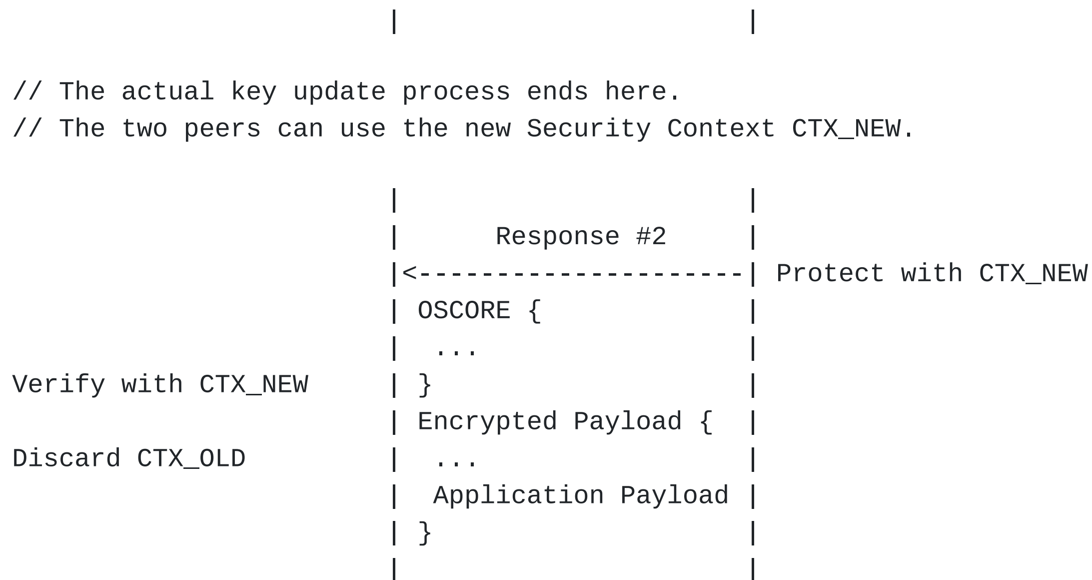


Figure 6: Example of the KUDOS reverse message flow

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 CTX_OLD as usual, the server generates a value N1 and provides the updateCtx() function with the input N = N1, X = X1, and CTX_OLD, in order to derive the temporary Security Context CTX_1.

Then, the server sends an OSCORE response to the client, protected with 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. Consistently with Section 3, the server includes its Sender Sequence Number as Partial IV in the response. After that, the server deletes CTX_1.

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

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

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

bit set to 1 and specifies N2 as 'nonce' and N1 as 'old_nonce'. After that, the client deletes CTX_1.

Upon receiving the OSCORE request, the server retrieves the values N1 from the 'old_nonce' field of the OSCORE Option, the value N2 from the 'nonce' field of the OSCORE Option, 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 CTX_OLD, in order to derive the new Security Context CTX_NEW. Finally, the server verifies the request by using CTX_NEW and deletes CTX_OLD.

From then on, the two peers can protect their message exchanges by using CTX_NEW. In particular, as shown in the example in Figure 6, the server can send an OSCORE response protected with CTX_NEW.

In case the client does not successfully verify the response, the same error handling specified in Section 8.4 of [RFC8613] applies. This does not result in deleting CTX_NEW. If the client successfully verifies the response using CTX_NEW, the client deletes CTX_OLD. Note that, if the verification of the response fails, the client may want to send again the normal OSCORE request to the server it initially sent (to /temp in the example above), in order to ensure the retrieval of the resource representation.

More generally, as soon as the client successfully verifies an incoming message protected with CTX_NEW, the client deletes CTX_OLD.

Note that the client achieves key confirmation only when receiving a message from the server as protected with 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 CTX_NEW and start a new KUDOS execution acting again as CoAP client, i.e., as initiator in the forward message flow (see 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 6, 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 from cryptographically matching 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.

During an ongoing KUDOS execution the client MUST NOT send any non-KUDOS requests to the server, even when NSTART is greater than 1 (see Section 4.7 of [RFC7252]).

4.4. Avoiding Deadlocks

This section defines how to avoid a deadlock in different scenarios.

4.4.1. Scenario 1

In this scenario, an execution of KUDOS fails at PEER_1 acting as initiator, but successfully completes at PEER_2 acting as responder. After that, PEER_1 still stores CTX_OLD, while PEER_2 stores CTX_OLD and the just derived CTX_NEW.

Then, PEER_1 starts a new KUDOS execution acting again as initiator, by sending the first KUDOS message as a CoAP request. This is protected with a temporary Security Context CTX_1, which is newly derived from the retained CTX_OLD, and from the new values X1 and N1 exchanged in the present KUDOS execution.

Upon receiving the first KUDOS message, PEER_2, acting again as responder, proceeds as follows.

1. PEER_2 attempts to verify the first KUDOS message by using a temporary Security Context CTX_1'. This is derived from the Security Context CTX_NEW established during the latest successfully completed KUDOS execution.

2. The message verification inevitably fails. If PEER_2 is acting as CoAP server, it MUST NOT reply with an unprotected 4.01 (Unauthorized) CoAP response yet.

3. PEER_2 MUST attempt to verify the first KUDOS message by using a temporary Security Context CTX_1. This is newly derived from the Security Context CTX_OLD retained after the latest successfully completed KUDOS execution, and from the values X1 and N1 exchanged in the present KUDOS execution.

If the message verification fails, PEER_2: i) retains CTX_OLD and CTX_NEW from the latest successfully completed KUDOS execution; ii) if acting as CoAP server, replies with an unprotected 4.01 (Unauthorized) CoAP response.

If the message verification succeeds, PEER_2: i) retains CTX_OLD from the latest successfully completed KUDOS execution; ii) replaces CTX_NEW from the latest successfully completed KUDOS execution with a new Security Context CTX_NEW', derived from CTX_OLD and from the values X1, X2, N1, and N2 exchanged in the present KUDOS execution; iii) replies with the second KUDOS message, which is protected with the just derived CTX_NEW'.

4.4.2. Scenario 2

In this scenario, an execution of KUDOS fails at PEER_1 acting as initiator, but successfully completes at PEER_2 acting as responder. After that, PEER_1 still stores CTX_OLD, while PEER_2 stores CTX_OLD and the just derived CTX_NEW.

Then, PEER_2 starts a new KUDOS execution, this time acting as initiator, by sending the first KUDOS message as a CoAP request. This is protected with a temporary Security Context CTX_1, which is newly derived from CTX_NEW established during the latest successfully completed KUDOS execution, as well as from the new values X1 and N1 exchanged in the present KUDOS execution.

Upon receiving the first KUDOS message, PEER_1, this time acting as responder, proceeds as follows.

1. PEER_1 attempts to verify the first KUDOS message by using a temporary Security Context CTX_1', which is derived from the retained Security Context CTX_OLD and from the values X1 and N1 exchanged in the present KUDOS execution.

2. The message verification inevitably fails. If PEER_1 is acting as CoAP server, it replies with an unprotected 4.01 (Unauthorized) CoAP response.

3. If PEER_2 does not receive the second KUDOS message for a pre-defined amount of time, or if it receives a 4.01 (Unauthorized) CoAP response when acting as CoAP client, then PEER_2 can start a new KUDOS execution for a maximum, pre-defined number of times.

   In this case, PEER_2 sends a new first KUDOS message protected with a temporary Security Context CTX_1', which is derived from the retained CTX_OLD, as well as from the new values X1 and N1 exchanged in the present KUDOS execution.

During this time, PEER_2 does not delete CTX_NEW established during the latest successfully completed KUDOS execution, and does not delete CTX_OLD unless it successfully verifies an incoming message protected with CTX_NEW.

4. Upon receiving such a new, first KUDOS message, PEER_1 verifies it by using the temporary Security Context CTX_1', which is derived from the Security Context CTX_OLD, and from the values X1 and N1 exchanged in the present KUDOS execution.

   If the message verification succeeds, PEER_1 derives an OSCORE Security Context CTX_NEW' from CTX_OLD and from the values X1, X2, N1, and N2 exchanged in the present KUDOS execution. Then, it replies with the second KUDOS message, which is protected with the latest, just derived CTX_NEW'.

5. Upon receiving such second KUDOS message, PEER_2 derives CTX_NEW' from the retained CTX_OLD and from the values X1, X2, N1, and N2 exchanged in the present KUDOS execution. Then, PEER_2 attempts to verify the KUDOS message using the just derived CTX_NEW'.

   If the message verification succeeds, PEER_2 deletes the retained CTX_OLD as well as the retained CTX_NEW established during the immediately previously, successfully completed KUDOS execution.

4.4.3. Scenario 3

When KUDOS is run in the reverse message flow (see Section 4.3.2), 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 does not 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 reverse message flow. 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.

4.5. 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.

4.5.1. 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. If that is not the case, the device is denoted as "non-CAPABLE".

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.

* A peer that is a non-CAPABLE device MUST support the no-FS mode.

* A peer that is a CAPABLE device MUST support the FS mode and the no-FS mode.

* As an exception to the nonces being generated as random values (see Section Section 4.3), a peer that is a CAPABLE device MAY use a value obtained from a monotonically incremented counter as nonce N1 or N2. This has privacy implications, which are described in Section Section 5. In such a case, the peer MUST enforce measures to ensure freshness of the nonce values. For example, the peer can use the same procedure described in Appendix B.1.1 of [RFC8613] for handling the OSCORE Sender Sequence Number values. These measures require to regularly store the used counter values in non-volatile memory, which makes non-CAPABLE devices unable to safely use counter values as nonce values.

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 are not 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.

4.5.1.1. 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.

Following a state loss (e.g., due to a reboot), a device MUST first complete a successful KUDOS execution (with either of the workflows) before exchanging OSCORE-protected application data with another peer. An exception is a CAPABLE device implementing a functionality for safely reusing old keying material, such as the one defined in Appendix B.1 of [RFC8613].

4.5.2. 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 a non-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. 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 be a non-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 a 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 a non-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 such latter value is 0 and the responder is a non-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 non-CAPABLE devices or only the peer acting as initiator is a non-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 non-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 non-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 consider the KUDOS execution unsuccessful and 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. This response is a KUDOS message, and it MUST have the 'd' bit and the 'p' bit set to 1.

    When receiving the error response, the initiator learns that the responder is a non-CAPABLE device (and hence not able to run KUDOS in FS mode), since the 'p' bit in the error response is set to 1, while the 'p' bit in the corresponding request was set to 0. Hence, the initiator MUST consider the KUDOS execution unsuccessful, and 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 MUST consider the KUDOS execution unsuccessful and MUST send 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, the initiator learns that the responder is a non-CAPABLE device (and hence not able

to run KUDOS in FS mode), since the 'p' bit in the request is set to 1, while the 'p' bit in the response previously sent as first KUDOS message was set to 0. Also, the initiator SHOULD NOT send any response to such a request, and the responder SHOULD NOT expect any such response.

In either case, both KUDOS peers delete the OSCORE Security Contexts CTX_1 and CTX_NEW. Also, both peers MUST retain CTX_OLD for use during the next KUDOS execution in the no-FS mode. This is in contrast with the typical behavior where CTX_OLD is deleted upon reception of a message protected with CTX_NEW.

4.6. 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, e.g., 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 4.6.1. 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.

]

4.6.1. 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 or equal 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.

4.7. Retention Policies

Applications MAY define policies that allow a peer to temporarily keep the old Security Context CTX_OLD beyond having established the new Security Context CTX_NEW and having achieved key confirmation, rather than simply overwriting CTX_OLD with 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.

A peer MUST NOT retain CTX_OLD beyond the establishment of CTX_NEW and the achievement of key confirmation, if any of the following conditions holds: CTX_OLD is expired; limits set for safe key usage have been reached [I-D.ietf-core-oscore-key-limits], for the Recipient Key of the Recipient Context of CTX_OLD.

4.8. 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 2). 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.

4.8.1. KUDOS Interleaved with Other Message Exchanges

During a KUDOS execution, a peer that is a CoAP Client must be ready to receive CoAP responses that are not KUDOS messages and that are protected with a different OSCORE Security Context than the one that was used to protect the corresponding request.

This can happen, for instance, when a CoAP client sends a request and, shortly after that, it executes KUDOS. In such a case, the CoAP request is protected with CTX_OLD, while the CoAP response from the server is protected with CTX_NEW. Another case is when incoming responses are Observe notifications protected with CTX_NEW, while the corresponding request from the CoAP client that started the observation was protected with CTX_OLD.

Another case is when running KUDOS in the reverse message flow, if the client uses NSTART > 1 and one of its requests triggers a KUDOS execution, i.e., the server replies with the first KUDOS message by acting as responder. The other requests would be latest served by the server after KUDOS has been completed.

4.8.2. Communication Overhead

Each of the two KUDOS messages displays a small communication overhead. This is determined by the following, additional information conveyed in the OSCORE option (see Section 4.1).

* The second byte of the OSCORE option.

* The byte 'x' of the OSCORE option.

* The nonce conveyed in the 'nonce' field of the OSCORE option. Its size ranges from 1 to 16 bytes as indicated in the 'x' byte, and is typically of 8 bytes.

Assuming nonces of the same size in both messages of the same KUDOS execution, this results in the following minimum, typical, and maximum communication overhead, when considering a nonce with size 1, 8, and 16 bytes, respectively. All the indicated values are in bytes.

	Forward message flow			Reverse message flow		
Nonce size	First KUDOS message	Second KUDOS message	Total	First KUDOS message	Second KUDOS message	Total
1	3	3	6	3	4	7
8	10	10	20	10	11	21
16	18	18	36	18	19	37

4.8.3. Well-Known KUDOS Resource

According to this specification, KUDOS is transferred in POST requests and 2.04 (Changed) responses. If a client wishes to execute the KUDOS procedure as initiator without triggering any application processing on the server, then the request sent as first KUDOS message must target a KUDOS resource, e.g., at the Uri-Path "/.well-known/kudos" (see Section 6.3), or at an alternative Uri-Path that can be discovered, e.g., by using a resource directory [RFC9176]. In order to discover a server's KUDOS resource, client applications can use the resource type "core.kudos" (see Section 6.4).

4.8.4. Rekeying when Using SCHC with OSCORE

In the interest of rekeying, the following points must be taken into account when using the Static Context Header Compression and

fragmentation (SCHC) framework [RFC8724] for compressing CoAP messages protected with OSCORE, as defined in [RFC8824].

Compression of the OSCORE Partial IV has implications for the frequency of rekeying. That is, if the Partial IV is compressed, the communicating peers must perform rekeying more often, as the available Partial IV space becomes smaller due to the compression. For instance, if only 3 bits of the Partial IV are sent, then the maximum PIV before having to rekey is only 2^3 - 1 = 7.

Furthermore, any time the SCHC context Rules are updated on an OSCORE endpoint, that endpoint must perform a rekeying (see Section 9 of [RFC8824]).

That is, the use of SCHC plays a role in triggering KUDOS executions and in affecting their cadence. Hence, the used SCHC Rules and their update policies should ensure that the KUDOS executions occurring as their side effect do not significantly impair the gain from message compression.

4.9. Signaling KUDOS support in EDHOC

The EDHOC protocol defines the transport of additional External Authorization Data (EAD) within an optional EAD field of the EDHOC messages (see Section 3.8 of [I-D.ietf-lake-edhoc]). An EAD field is composed of one or multiple EAD items, each of which specifies an identifying 'ead_label' encoded as a CBOR integer, and an optional 'ead_value' encoded as a CBOR bstr.

This document defines a new EDHOC EAD item KUDOS_EAD and registers its 'ead_label' in Section 6.2. By including this EAD item in an outgoing EDHOC message, a sender peer can indicate whether it supports KUDOS and in which modes, as well as query the other peer about its support. Note that peers do not have to use this EDHOC EAD item to be able to run KUDOS with each other, irrespective of the modes they support. The possible values of the 'ead_value' are as follows:

Name	Value	Description
ASK	h'' (0x40)	Used only in EDHOC message_1. It asks the recipient peer to specify in EDHOC message_2 whether it supports KUDOS.
NONE	h'00' (0x4100)	Used only in EDHOC message_2 and message_3. It specifies that the sender peer does not support KUDOS.
FULL	h'01' (0x4101)	Used only in EDHOC message_2 and message_3. It specifies that the sender peer supports KUDOS in FS mode and no-FS mode.
PART	h'02' (0x4102)	Used only in EDHOC message_2 and message_3. It specifies that the sender peer supports KUDOS in no-FS mode only.

When the KUDOS_EAD item is included in EDHOC message_1 with 'ead_value' ASK, a recipient peer that supports the KUDOS_EAD item MUST specify whether it supports KUDOS in EDHOC message_2.

When the KUDOS_EAD item is not included in EDHOC message_1 with 'ead_value' ASK, a recipient peer that supports the KUDOS_EAD item MAY still specify whether it supports KUDOS in EDHOC message_2.

When the KUDOS_EAD item is included in EDHOC message_2 with 'ead_value' FULL or PART, a recipient peer that supports the KUDOS_EAD item SHOULD specify whether it supports KUDOS in EDHOC message_3. An exception applies in case, based on application policies or other context information, the recipient peer that receives EDHOC message_2 already knows that the sender peer is supposed to have such knowledge.

When the KUDOS_EAD item is included in EDHOC message_2 with 'ead_value' NONE, a recipient peer that supports the KUDOS_EAD item MUST NOT specify whether it supports KUDOS in EDHOC message_3.

In the following cases, the recipient peer silently ignores the KUDOS_EAD item specified in the received EDHOC message, and does not include a KUDOS_EAD item in the next EDHOC message it sends (if any).

* The recipient peer does not support the KUDOS_EAD item.

* The KUDOS_EAD item is included in EDHOC message_1 with 'ead_value' different than ASK

* The KUDOS_EAD item is included in EDHOC message_2 or message_3 with 'ead_value' ASK.

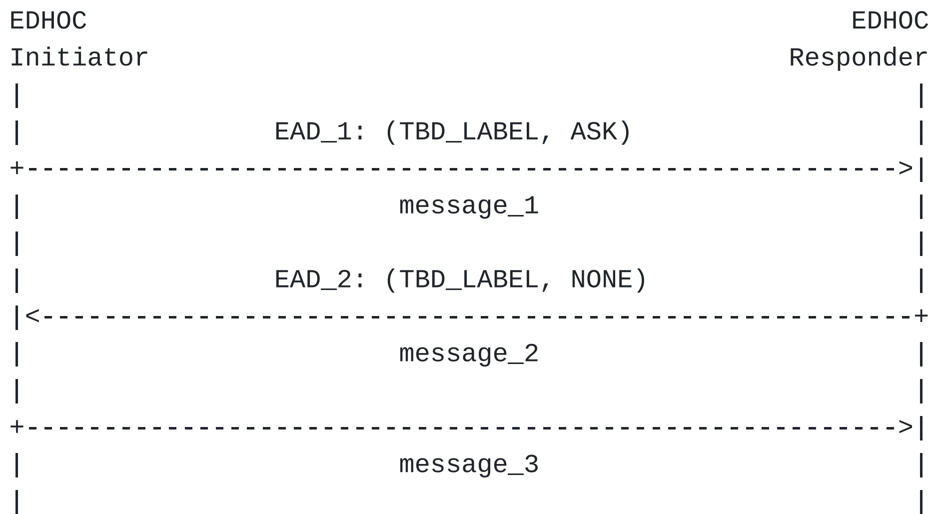
* The KUDOS_EAD item is included in EDHOC message_4.

That is, by specifying 'ead_value' ASK in EDHOC message_1, a peer A can indicate to the other peer B that it wishes to know if B supports KUDOS and in what mode(s). In the following EDHOC message_2, B indicates whether it supports KUDOS and in what mode(s), by specifying either NONE, FULL, or PART as 'ead_value'. Specifying the 'ead_value' FULL or PART in EDHOC message_2 also asks A to indicate whether it supports KUDOS in EDHOC message_3.

To further illustrate the functionality, two examples are presented below as EDHOC executions where only the new KUDOS_EAD item is shown when present, and assuming that no other EAD items are used by the two peers.

```
EDHOC                                                      EDHOC
Initiator                                              Responder
|                                                              |
|                EAD_1: (TBD_LABEL, ASK)                       |
+------------------------------------------------------------->|
|                        message_1                             |
|                                                              |
|                EAD_2: (TBD_LABEL, FULL)                      |
|<-------------------------------------------------------------+
|                        message_2                             |
|                                                              |
|                EAD_3: (TBD_LABEL, FULL)                      |
+------------------------------------------------------------->|
|                        message_3                             |
|                                                              |
```

In the example above, the Initiator asks the EDHOC Responder about its support for KUDOS ('ead_value' = ASK). In EDHOC message_2, the Responder indicates that it supports both the FS and no-FS mode of KUDOS ('ead_value' = FULL). Finally, in EDHOC message_3, the Initiator indicates that it also supports both the FS and no-FS mode of KUDOS ('ead_value' = FULL). After the EDHOC execution has successfully finished, both peers are aware that they both support KUDOS, in the FS and no-FS modes.



In this second example, the Initiator asks the EDHOC Responder about its support for KUDOS ('ead_value' = ASK). In EDHOC message_2, the Responder indicates that it does not support KUDOS at all ('ead_value' = NONE). Finally, in EDHOC message_3, the Initiator does not include the KUDOS_EAD item, since it already knows that using KUDOS with the other peer will not be possible. After the EDHOC execution has successfully finished, the Initiator is aware that the Responder does not support KUDOS, which the two peers are not going to use with each other.

5. 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.

As mentioned in Section 4.3, it is RECOMMENDED that the size for nonces N1 and N2 is 8 bytes. The application needs to set the size of each nonce such that the probability of its value being repeated is negligible. Note that the probability of collision of nonce values is heightened by the birthday paradox. However, considering a nonce size of 8 bytes there will be a collision on average after approximately 2^32 instances of Response #1 messages.

Overall, the size of the nonces N1 and N2 should be set such that the security level is harmonized with other components of the deployment. Considering the constraints of embedded implementations, there might be a need for allowing N1 and N2 values that are smaller in size. This is acceptable, provided that safety, reliability, and robustness within the system can still be assured. Although using

nonces that are smaller in size means that there will be a collision on average after fewer KUDOS messages have been sent, this should not pose significant problems even for a constrained server operating at a capacity of one request per second.

The nonces exchanged in the KUDOS messages are sent in the clear, so using random nonces is preferable for maintaining privacy. If instead a counter value is used, this can leak some information about the peers. Specifically, using counters will reveal the frequency of rekeying procedures performed.

[TODO: Add more considerations.]

6. 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.

6.1. 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
0	Extension-1 Flag	Set to 1 if the OSCORE Option specifies a second byte, which includes the OSCORE flag bits 8-15	[RFC-XXXX]
8	Extension-2 Flag	Set to 1 if the OSCORE Option specifies a third byte, which includes the OSCORE flag bits 16-23	[RFC-XXXX]
15	Nonce Flag	Set to 1 if nonce is present in the compressed COSE object	[RFC-XXXX]
16	Extension-3 Flag	Set to 1 if the OSCORE Option specifies a fourth byte, which includes the OSCORE flag bits 24-31	[RFC-XXXX]
24	Extension-4 Flag	Set to 1 if the OSCORE Option specifies a fifth byte, which includes the OSCORE flag bits 32-39	[RFC-XXXX]
32	Extension-5 Flag	Set to 1 if the OSCORE Option specifies a sixth byte, which includes the OSCORE flag bits 40-47	[RFC-XXXX]
40	Extension-6 Flag	Set to 1 if the OSCORE Option specifies a seventh byte, which includes the OSCORE flag bits 48-55	[RFC-XXXX]
48	Extension-7 Flag	Set to 1 if the OSCORE Option specifies an eigth byte, which includes the OSCORE flag bits 56-63	[RFC-XXXX]

In the same registry, IANA is asked to mark as 'Unassigned' the entry with Bit Position of 1, i.e., to update the entry as follows.

Bit Position	Name	Description	Reference
1	Unassigned		

6.2. EDHOC External Authorization Data Registry

IANA is asked to add the following entries to the "EDHOC External Authorization Data" registry defined in Section 10.5 of [I-D.ietf-lake-edhoc] within the "Ephemeral Diffie-Hellman Over COSE (EDHOC)" registry group.

Label	Description	Reference
TBD1	Indicates whether this peer supports KUDOS and in which mode(s)	[RFC-XXXX]

6.3. The Well-Known URI Registry

IANA is asked to add the 'kudos' well-known URI to the Well-Known URIs registry as defined by [RFC8615].

- *URI suffix: kudos
- *Change controller: IETF
- *Specification document(s): [RFC-XXXX]
- *Related information: None

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

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

- *Value: "core.kudos"
- *Description: KUDOS resource.
- *Reference: [RFC-XXXX]

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/info/rfc2119>.

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

[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>.

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.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-07, 31 May 2023, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-aead-limits-07>.

[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>.

[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>.

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

[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>.

[RFC8824] Minaburo, A., Toutain, L., and R. Andreasen, "Static Context Header Compression (SCHC) for the Constrained Application Protocol (CoAP)", RFC 8824, DOI 10.17487/RFC8824, June 2021, <https://www.rfc-editor.org/info/rfc8824>.

[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>.

[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>.

[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/info/rfc9200>.

[RFC9203] Palombini, F., Seitz, L., Selander, G., and M. Gunnarsson, "The Object Security for Constrained RESTful Environments (OSCORE) Profile of the Authentication and Authorization for Constrained Environments (ACE) Framework", RFC 9203, DOI 10.17487/RFC9203, August 2022, <https://www.rfc-editor.org/info/rfc9203>.

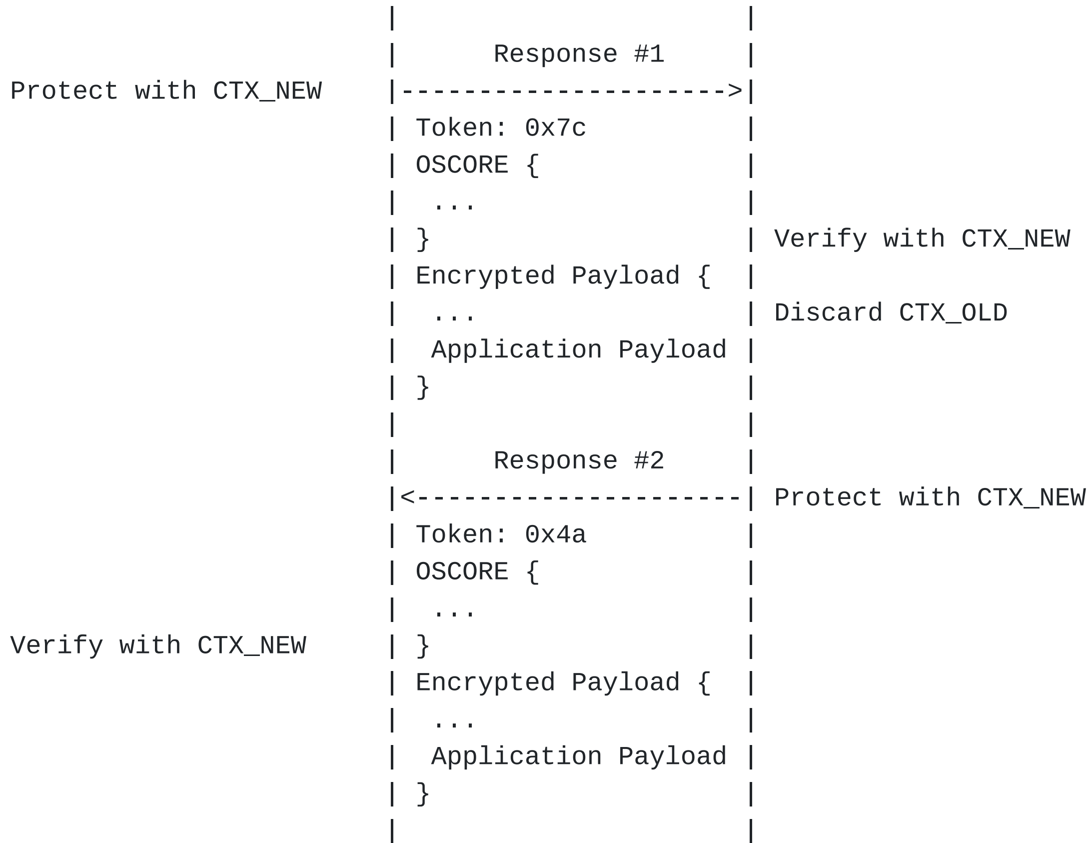
Appendix A. Forward Message Flow using two CoAP Requests

This section presents an example of KUDOS run in the forward message flow, with the client acting as KUDOS initiator, and both KUDOS messages being CoAP requests.

```
                  Client/Server          Client/Server
                   (initiator)            (responder)
                        |                      |
Generate N1             |                      |
                        |                      |
CTX_1 = updateCtx(      |                      |
        X1,             |                      |
        N1,             |                      |
        CTX_OLD)        |                      |
                        |                      |
                        |      Request #1      |
Protect with CTX_1      |--------------------->| /.well-known/kudos
                        | Token: 0x4a          |
                        | OSCORE {             |
                        |  ...                 |
                        |  Partial IV: 0       |
                        |  ...                 |
                        |  d flag: 1           | CTX_1 = updateCtx(
                        |  x: X1               |         X1,
                        |  nonce: N1           |         N1,
                        |  ...                 |         CTX_OLD)
                        | }                    |
                        | Encrypted Payload {  | Verify with CTX_1
                        |  ...                 |
                        | }                    | Generate N2
                        |                      |
                        |                      | CTX_NEW = updateCtx(
                        |                      |           Comb(X1,X2),
                        |                      |           Comb(N1,N2),
                        |                      |           CTX_OLD)
                        |                      |
                        |      Request #2      |
     /.well-known/kudos |<---------------------| Protect with CTX_NEW
                        | Token: 0x7c          |
                        | OSCORE {             |
                        |  ...                 |
CTX_NEW = updateCtx(    |  Partial IV: 0       |
          Comb(X1,X2),  |  ...                 |
          Comb(N1,N2),  |  d flag: 1           |
          CTX_OLD)      |  x: X2               |
                        |  nonce: N2           |
Verify with CTX_NEW     |  ...                 |
                        | }                    |
Discard CTX_OLD         | Encrypted Payload {  |
                        |  ...                 |
                        | }                    |
                        |                      |

// The actual key update process ends here.
```

// The two peers can use the new Security Context CTX_NEW.

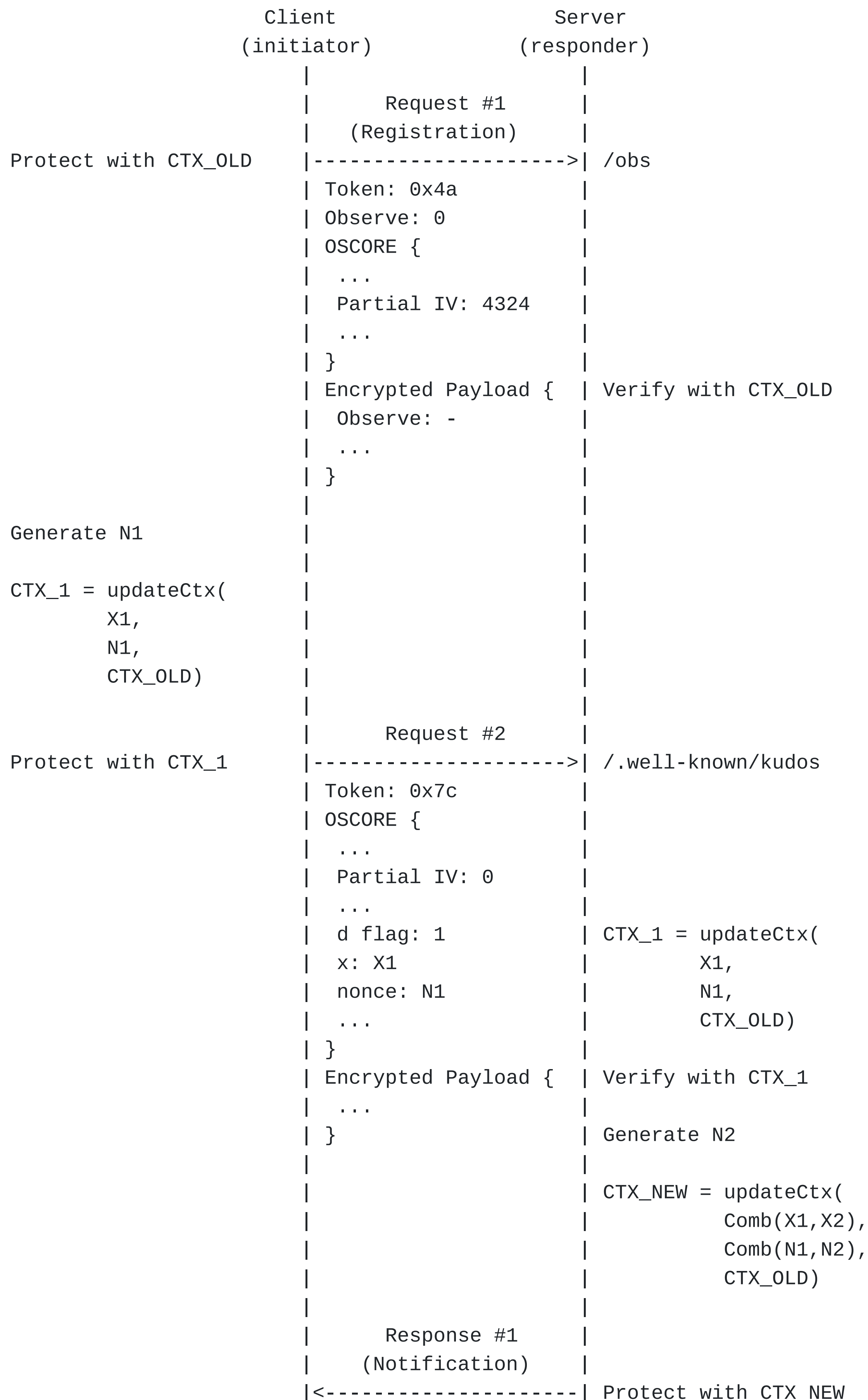


```
                        |                      |
                        |      Response #1     |
Protect with CTX_NEW    |--------------------->|
                        | Token: 0x7c          |
                        | OSCORE {             |
                        |  ...                 |
                        | }                    | Verify with CTX_NEW
                        | Encrypted Payload {  |
                        |  ...                 | Discard CTX_OLD
                        |  Application Payload |
                        | }                    |
                        |                      |
                        |      Response #2     |
                        |<---------------------| Protect with CTX_NEW
                        | Token: 0x4a          |
                        | OSCORE {             |
                        |  ...                 |
Verify with CTX_NEW     | }                    |
                        | Encrypted Payload {  |
                        |  ...                 |
                        |  Application Payload |
                        | }                    |
                        |                      |
```

Figure 7: Example of the KUDOS forward message flow where both KUDOS messages are requests.

Appendix B. Forward Message Flow with Response #1 unrelated to Request #1

This section presents an example of KUDOS run in the forward message flow, with the client acting as KUDOS initiator, and where the second KUDOS message Response #1 is not a response to the first KUDOS message Request #2, but rather an unrelated Observe notification as a response to the non-KUDOS message Request #1



Client
(initiator)
Server
(responder)
Request #1
(Registration)
Protect with CTX_OLD
/obs
Token: 0x4a
Observe: 0
OSCORE {
...
Partial IV: 4324
...
}
Encrypted Payload {
Verify with CTX_OLD
Observe: -
...
}
Generate N1
CTX_1 = updateCtx(
X1,
N1,
CTX_OLD)
Request #2
Protect with CTX_1
/.well-known/kudos
Token: 0x7c
OSCORE {
...
Partial IV: 0
...
d flag: 1
CTX_1 = updateCtx(
x: X1
X1,
nonce: N1
N1,
...
CTX_OLD)
}
Encrypted Payload {
Verify with CTX_1
...
}
Generate N2
CTX_NEW = updateCtx(
Comb(X1,X2),
Comb(N1,N2),
CTX_OLD)
Response #1
(Notification)
Protect with CTX_NEW

```
                        | Token: 0x4a            |
                        | Observe: 1             |
                        | OSCORE {               |
                        |  ...                   |
CTX_NEW = updateCtx(    |  Partial IV: 0         |
          Comb(X1,X2),  |  ...                   |
          Comb(N1,N2),  |  d flag: 1             |
          CTX_OLD)      |  x: X2                 |
                        |  nonce: N2             |
Verify with CTX_NEW     |  ...                   |
                        | }                      |
Discard CTX_OLD         | Encrypted Payload {    |
                        | Observe: -             |
                        |  ...                   |
                        | }                      |
                        |                        |

// The actual key update process ends here.
// The two peers can use the new Security Context CTX_NEW.

                        |      Response #2       |
                        |<-----------------------| Protect with CTX_NEW
                        | Token: 0x7c            |
                        | OSCORE {               |
                        |  ...                   |
Verify with CTX_NEW     | }                      |
                        | Encrypted Payload {    |
                        |  ...                   |
                        |  Application Payload   |
                        | }                      |
                        |                        |
```

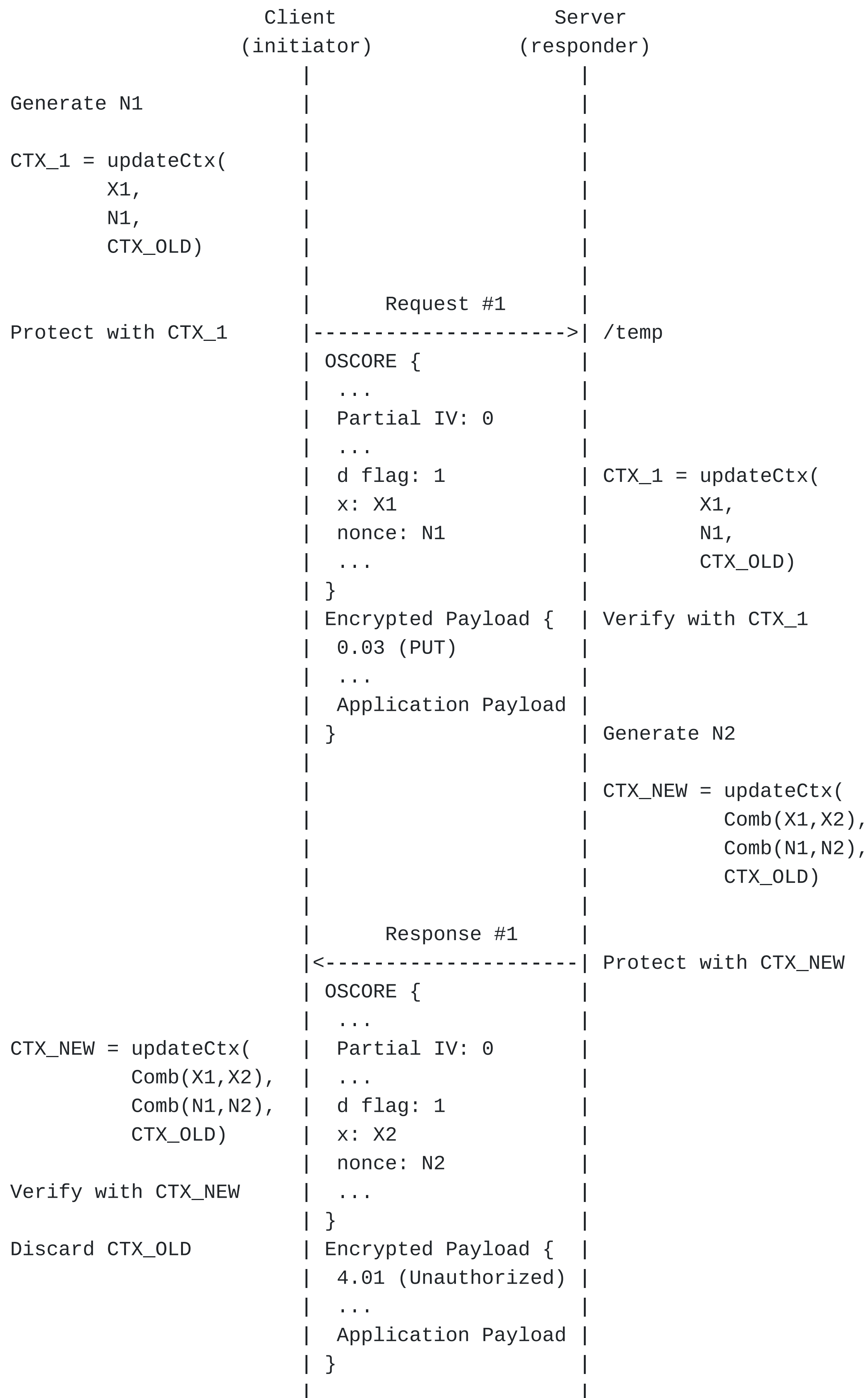
Figure 8: Example of the KUDOS forward message flow where the second KUDOS message Response #1 is not a response to Request #1.

Appendix C. Forward Message Flow Targeting a non-KUDOS Resource at Server

This section presents an example of KUDOS run in the forward message flow, with the client acting as KUDOS initiator, and with the KUDOS message Request #1 targeting a non-KUDOS resource at the Uri-Path "/temp". The server application has freshness requirements on the requests targeting the resource at "/temp".

Note the presence of an application payload in the KUDOS message Request #1 and in the non-KUDOS message Request #2, both of which are composed as PUT requests. That request method is part of the encrypted payload, since it is protected by OSCORE.

Also note the fact that the KUDOS message Response #1 is composed as a 4.01 (Unauthorized) response, while the non-KUDOS message Response #2 is composed as a 2.04 (Changed) repsonse. Those response codes are part of the encrypted payload, since they are protected by OSCORE.


Client
(initiator)
Server
(responder)
Generate N1
CTX_1 = updateCtx(
X1,
N1,
CTX_OLD)
Request #1
Protect with CTX_1
/temp
OSCORE {
...
Partial IV: 0
...
d flag: 1
x: X1
nonce: N1
...
}
CTX_1 = updateCtx(
X1,
N1,
CTX_OLD)
Encrypted Payload {
0.03 (PUT)
...
Application Payload
}
Verify with CTX_1
Generate N2
CTX_NEW = updateCtx(
Comb(X1,X2),
Comb(N1,N2),
CTX_OLD)
Response #1
Protect with CTX_NEW
OSCORE {
...
Partial IV: 0
...
d flag: 1
x: X2
nonce: N2
...
}
CTX_NEW = updateCtx(
Comb(X1,X2),
Comb(N1,N2),
CTX_OLD)
Verify with CTX_NEW
Discard CTX_OLD
Encrypted Payload {
4.01 (Unauthorized)
...
Application Payload
}

```
// The actual key update process ends here.
// The two peers can use the new Security Context CTX_NEW.
```

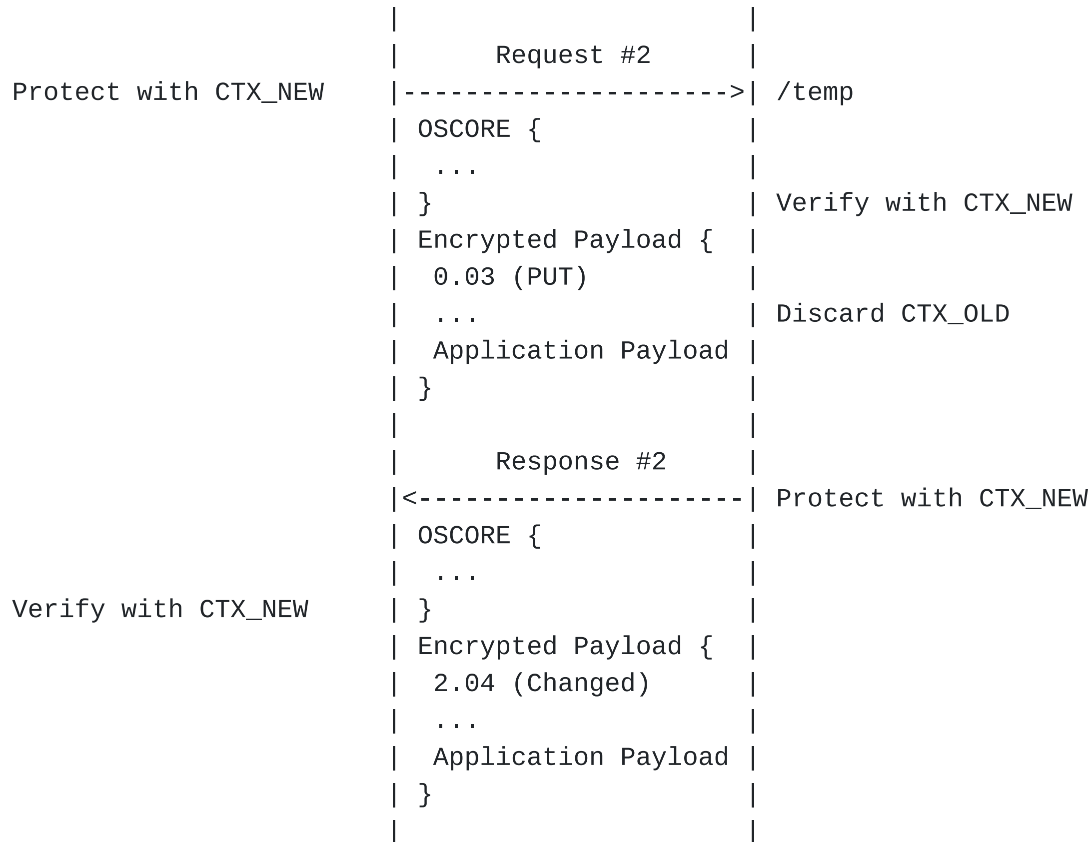


Figure 9: Example of the KUDOS forward message flow where the KUDOS message Request #1 targets a non-KUDOS resource.

Appendix D. Document Updates

RFC EDITOR: PLEASE REMOVE THIS SECTION.

D.1. Version -06 to -07

* Removed material about the ID update procedure, which has been split out into a separate draft.

* Allow non-random nonces for CAPABLE devices.

* Editorial improvements.

* Permit flexible message flow with KUDOS messages as any request/response.

* Enable sending KUDOS messages as regular application messages.

D.2. Version -05 to -06

* Mandate support for both the forward and reverse message flow.

* Mention the EDHOC and OSCORE profile of ACE as method for rekeying.

* Clarify definition of KUDOS (request/response) message.

* Further extend the OSCORE option to transport N1 in the second KUDOS message as a request.

* Mandate support for the no-FS mode on CAPABLE devices.

* Explain when KUDOS fails during selection of mode.

* Explicitly forbid using old keying material after reboot.

* Editorial improvements.

D.3. Version -04 to -05

* Note on client retransmissions if KUDOS execution fails in reverse message flow.

* Specify what information needs to be written to non-volatile memory to handle reboots.

* Extended recommendations and considerations on minimum size of nonces N1 & N2.

* Arbitrary maximum size of the Recipient-ID Option.

* Detailed lifecycle of the OSCORE IDs update procedure.

* Described examples of OSCORE IDs update procedure.

* Examples of OSCORE IDs update procedure integrated in KUDOS.

* Considerations about using SCHC for CoAP with OSCORE.

* Clarifications and editorial improvements.

D.4. Version -03 to -04

* Removed content about key usage limits.

* Use of "forward message flow" and "reverse message flow".

* Update to RFC 8613 extended to include protection of responses.

* Include EDHOC_KeyUpdate() in the methods for rekeying.

* Describe reasons for using the OSCORE ID update procedure.

* Clarifications on deletion of CTX_OLD and CTX_NEW.

* Added new section on preventing deadlocks.

* Clarified that peers can decide to run KUDOS at any point.

* Defined preservation of observations beyond OSCORE ID updates.

* Revised discussion section, including also communication overhead.

* Defined a well-known KUDOS resource and a KUDOS resource type.

* Editorial improvements.

D.5. Version -02 to -03

* Use of the OSCORE flag bit 0 to signal more flag bits.

* In UpdateCtx(), open for future key derivation different than HKDF.

* Simplified updateCtx() to use only Expand(); used to be METHOD 2.

* Included the Partial IV if the second KUDOS message is a response.

* Added signaling of support for KUDOS in EDHOC.

* Clarifications on terminology and reasons for rekeying.

* Updated IANA considerations.

* Editorial improvements.

D.6. 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.

D.7. 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, Carsten Bormann, Rafa Marin-Lopez, 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 projects SIFIS-Home (Grant agreement 952652) and ARCADIAN-IoT (Grant agreement 101020259).

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