Transport Layer Security
Internet-Draft
Updates: 9261, 8446 (if approved)
Intended status: Standards Track
Expires: 8 October 2026

H. Tschofenig
Siemens
M. T端xen
M端nster Univ. of Applied Sciences
T. Reddy
Nokia
S. Fries
Siemens
Y. Rosomakho
Zscaler

6 April 2026

Extended Key Update for Transport Layer Security (TLS) 1.3
draft-ietf-tls-extended-key-update-12

Abstract

TLS 1.3 ensures forward secrecy by performing an ephemeral Diffie-Hellman key exchange during the initial handshake, protecting past communications even if a party's long-term keys (typically a private key with a corresponding certificate) are later compromised. While the built-in KeyUpdate mechanism allows application traffic keys to be refreshed during a session, it does not incorporate fresh entropy from a new key exchange and therefore does not provide post-compromise security. This limitation can pose a security risk in long-lived sessions, such as those found in industrial IoT or telecommunications environments.

To address this, this specification defines an extended key update mechanism that performs a fresh Diffie-Hellman exchange within an active session, thereby ensuring post-compromise security. By forcing attackers to exfiltrate new key material repeatedly, this approach mitigates the risks associated with static key compromise. Regular renewal of session keys helps contain the impact of such compromises. The extension is applicable to both TLS 1.3 and DTLS 1.3.

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 8 October 2026.

Copyright Notice

Copyright (c) 2026 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

The Transport Layer Security (TLS) 1.3 protocol provides forward secrecy by using fresh ephemeral key exchange during the initial handshake. In the base protocol, this key exchange is performed with (EC)DHE. By registering new NamedGroup codepoints, TLS specifications also define hybrid and post-quantum key exchange mechanisms for the same purpose. This ensures that encrypted communication remains confidential even if an attacker later obtains a party's long-term private key, protecting against passive adversaries who record encrypted traffic for later decryption.

TLS 1.3 also includes a KeyUpdate mechanism that allows traffic keys to be refreshed during an established session. However, this mechanism does not provide post-compromise security, as it applies only a key derivation function to the previous application traffic key as input. While this design is generally sufficient for short-lived connections, it may present security limitations in scenarios where sessions persist for extended periods, such as in industrial IoT or telecommunications systems, where continuous availability is critical and session renegotiation or resumption is impractical.

Earlier versions of TLS supported session renegotiation, which allowed peers to negotiate fresh keying material, including performing new Diffie-Hellman exchanges during the session lifetime. Due to protocol complexity and known vulnerabilities, renegotiation was first restricted by [TLS-RENEGOTIATION] and ultimately removed in TLS 1.3. While the KeyUpdate message was introduced to offer limited rekeying functionality, it does not fulfill the same cryptographic role as renegotiation and cannot refresh the TLS main secret and consequently cannot derive new secrets from fresh key exchange input. This limitation applies regardless of whether the session was established with traditional (EC)DHE, a post-quantum/traditional (PQ/T) hybrid exchange, or a post-quantum KEM exchange.

Security guidance from national agencies, such as ANSSI (France [ANSSI]), recommends the periodic renewal of cryptographic keys during long-lived sessions to limit the impact of key compromise. This approach encourages designs that force an attacker to perform dynamic key exfiltration, as defined in [CONFIDENTIALITY]. Dynamic key exfiltration refers to attack scenarios where an adversary must repeatedly extract fresh keying material to maintain access to protected data, increasing operational cost and risk for the attacker. In contrast, static key exfiltration, where the TLS main secret is extracted once and reused, poses a greater long-term threat, especially when session keys are not refreshed with fresh key exchange input rather than key derivation.

This specification defines a TLS extension that introduces an extended key update mechanism for sessions established with traditional (EC)DHE, hybrid PQ/T hybrid key exchange, or post-quantum KEM exchange. Unlike the standard key update, this mechanism allows peers to inject fresh key exchange input from the negotiated mechanism into an active session. By periodically rerunning the negotiated key exchange, this extension enables the derivation of new traffic keys that are independent of main secrets from prior epochs. As noted in Appendix F of [TLS], this approach mitigates the risk of static key exfiltration and shifts the attacker burden toward dynamic key exfiltration.

The proposed extension is applicable to both TLS 1.3 [TLS] and DTLS 1.3 [DTLS]. For clarity, the term "TLS" is used throughout this document to refer to both protocols unless otherwise specified.

2. Terminology and Requirements Language

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

To distinguish the key update procedure defined in [TLS] from the key update procedure specified in this document, we use the terms "standard key update" and "extended key update", respectively.

For terminology regarding traditional and PQ/T hybrid algorithms please see [RFC9794].

In this document, we use the term post-compromise security, as defined in [CCG16]. We assume that an adversary may obtain access to the application traffic keys.

Unless otherwise specified, all references to traffic keys in this document refer to application traffic keys and because the Extended Key Update procedure occurs after the handshake phase has completed, no handshake traffic keys are involved.

In this document, send key refers to the [sender]_write_key, and receive key refers to the [receiver]_write_key. These keys are derived from the active client_application_traffic_secret_N and server_application_traffic_secret_N, as defined in (D)TLS 1.3 [TLS], and are replaced with new ones after each successful Extended Key Update.

3. Negotiating the Extended Key Update

Client and servers use the TLS flags extension [TLS-FLAGS] to indicate support for the functionality defined in this document. We call this flag "Extended_Key_Update" flag.

The "Extended_Key_Update" flag proposed by the client in the ClientHello (CH) MUST be acknowledged in the EncryptedExtensions (EE), if the server also supports the functionality defined in this document and is configured to use it.

If the "Extended_Key_Update" flag is not set, servers ignore any of the functionality specified in this document and applications that require post-compromise security will have to initiate a full handshake.

4. Extended Key Update Messages

If the client and server agree to use the extended key update mechanism, the standard key update MUST NOT be used. In this case, the extended key update fully replaces the standard key update functionality.

Implementations that receive a classic KeyUpdate message after successfully negotiating the Extended Key Update functionality MUST terminate the connection with an "unexpected_message" alert.

The extended key update messages are signaled in a new handshake message named ExtendedKeyUpdate (EKU), with an internal uint8 message subtype indicating its role. This specification defines three ExtendedKeyUpdate message subtypes:

* key_update_request (0)

* key_update_response (1)

* key_update_finish (2)

New ExtendedKeyUpdate message subtypes are assigned by IANA as described in Section 13.3.

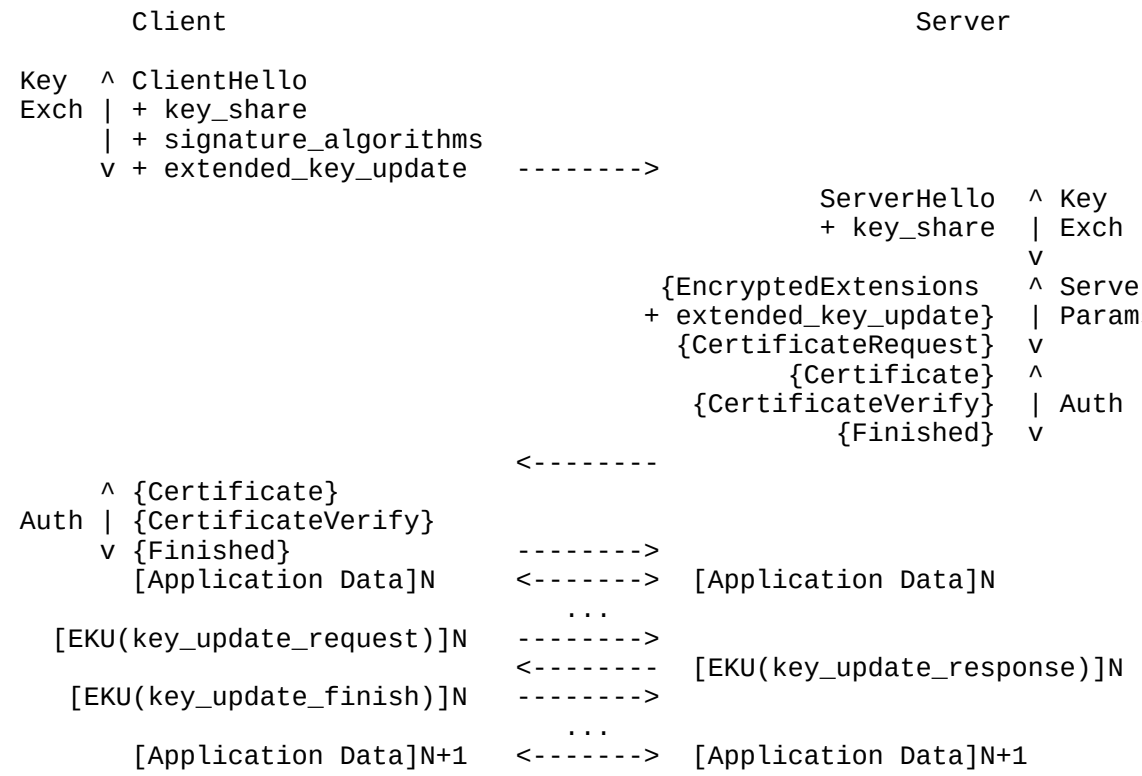
A TLS peer which receives a ExtendedKeyUpdate with an unexpected message subtype MUST abort the connection with an "unexpected_message" alert.

The extended key update process can be initiated by either peer after it has sent a Finished message. Implementations that receive an ExtendedKeyUpdate message prior to the sender having sent Finished MUST terminate the connection with an "unexpected_message" alert.

The KeyShareEntry carried in a key_update_request and in a key_update_response MUST use the group that was negotiated by the client and server during the initial handshake. An implementation that receives an algorithm other than previously negotiated MUST terminate the connection with an "illegal_parameter" alert.

Figure 1 shows the interaction graphically. First, support for the functionality in this specification is negotiated in the ClientHello and the EncryptedExtensions messages. Then, the ExtendedKeyUpdate exchange is sent to update the application traffic secrets.

The extended key update exchange is performed between the initiator and the responder; either the TLS client or the TLS server may act as initiator.



Legend:

+ Indicates noteworthy extensions sent in the previously noted message.

- Indicates optional or situation-dependent messages/extensions that are not always sent.

() Indicates messages protected using keys derived from a client_early_traffic_secret.

{} Indicates messages protected using keys derived from a [sender]_handshake_traffic_secret.

[]N Indicates messages protected using keys derived from [sender]_application_traffic_secret_N.

Figure 1: Extended Key Update Message Exchange in TLS 1.3.

The ExtendedKeyUpdate wire format is:

```
enum {
   key_update_request(0),
   key_update_response(1),
   key_update_finish(2),
   (255)
} ExtendedKeyUpdateType;

struct {
   ExtendedKeyUpdateType eku_type;
   select (eku_type) {
      case key_update_request: {
          KeyShareEntry key_share;
      }
      case key_update_response: {
          KeyShareEntry key_share;
      }
      case key_update_finish: {
          /* empty */
      }
   };
} ExtendedKeyUpdate;
```

Fields:

* eku_type: the subtype of the ExtendedKeyUpdate message.

* key_share: key share information. The contents of this field is determined by the specified group and its corresponding definition (see Section 4.2.8 of [TLS]).

5. TLS 1.3 Considerations

The following steps are taken by a TLS 1.3 implementation; the steps executed with DTLS 1.3 differ slightly.

1. The initiator sends ExtendedKeyUpdate(key_update_request) carrying a KeyShareEntry. While an extended key update is in progress, the initiator MUST NOT initiate another key update.

2. Upon receipt, the responder sends its own KeyShareEntry in a ExtendedKeyUpdate(key_update_response) message. While an extended key update is in progress, the responder MUST NOT initiate another key update. The responder MAY defer sending a response if system load or resource constraints prevent immediate processing. In such cases, the response MUST be sent once sufficient resources become available.

3. After the responder sends the ExtendedKeyUpdate(key_update_response) it MUST update its send keys.

4. Upon receipt of an ExtendedKeyUpdate(key_update_response) the initiator derives the new secrets from the exchanged key shares. The initiator then updates its receive keys and sends an empty ExtendedKeyUpdate(key_update_finish) message to complete the process. The initiator MUST NOT defer derivation of the secrets and sending the ExtendedKeyUpdate(key_update_finish) message as it would stall the communication.

5. After sending ExtendedKeyUpdate(key_update_finish), the initiator MUST update its send keys.

6. Upon receiving the initiator宴洰 ExtendedKeyUpdate(key_update_finish), the responder MUST update its receive keys.

Both initiator and responder MUST encrypt their ExtendedKeyUpdate messages using the old keys. The Responder MUST ensure that it has received ExtendedKeyUpdate(key_update_finish), encrypted with the old key, before accepting any messages encrypted with the new keys.

If TLS peers independently initiate the extended key update and the requests cross in flight, the ExtendedKeyUpdate(key_update_request) with the lower lexicographic order of the key_exchange value in KeyShareEntry MUST be ignored. This prevents each side from advancing keys by two generations. If the tie-break comparison yields equality (an event that should be impossible for genuine asymmetric key pairs), the endpoint MUST treat this as a protocol violation, send an "unexpected_message" alert, and close the connection.

The handshake framing uses a single HandshakeType for this message (see Figure 2).

```
struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* bytes in message */
    select (Handshake.msg_type) {
        case client_hello:          ClientHello;
        case server_hello:          ServerHello;
        case end_of_early_data:     EndOfEarlyData;
        case encrypted_extensions:  EncryptedExtensions;
        case certificate_request:   CertificateRequest;
        case certificate:           Certificate;
        case certificate_verify:    CertificateVerify;
        case finished:              Finished;
        case new_session_ticket:    NewSessionTicket;
        case key_update:            KeyUpdate;
        case extended_key_update:   ExtendedKeyUpdate;
    };
} Handshake;
```

Figure 2: TLS 1.3 Handshake Structure.

5.1. TLS 1.3 Extended Key Update Example

Figure 1 shows the high-level interaction between a TLS 1.3 client and server, while Figure 3 illustrates an example message exchange including the updated keys. Similar to the key_update message, the extended_key_update message can be sent by either peer after sending its Finished message.

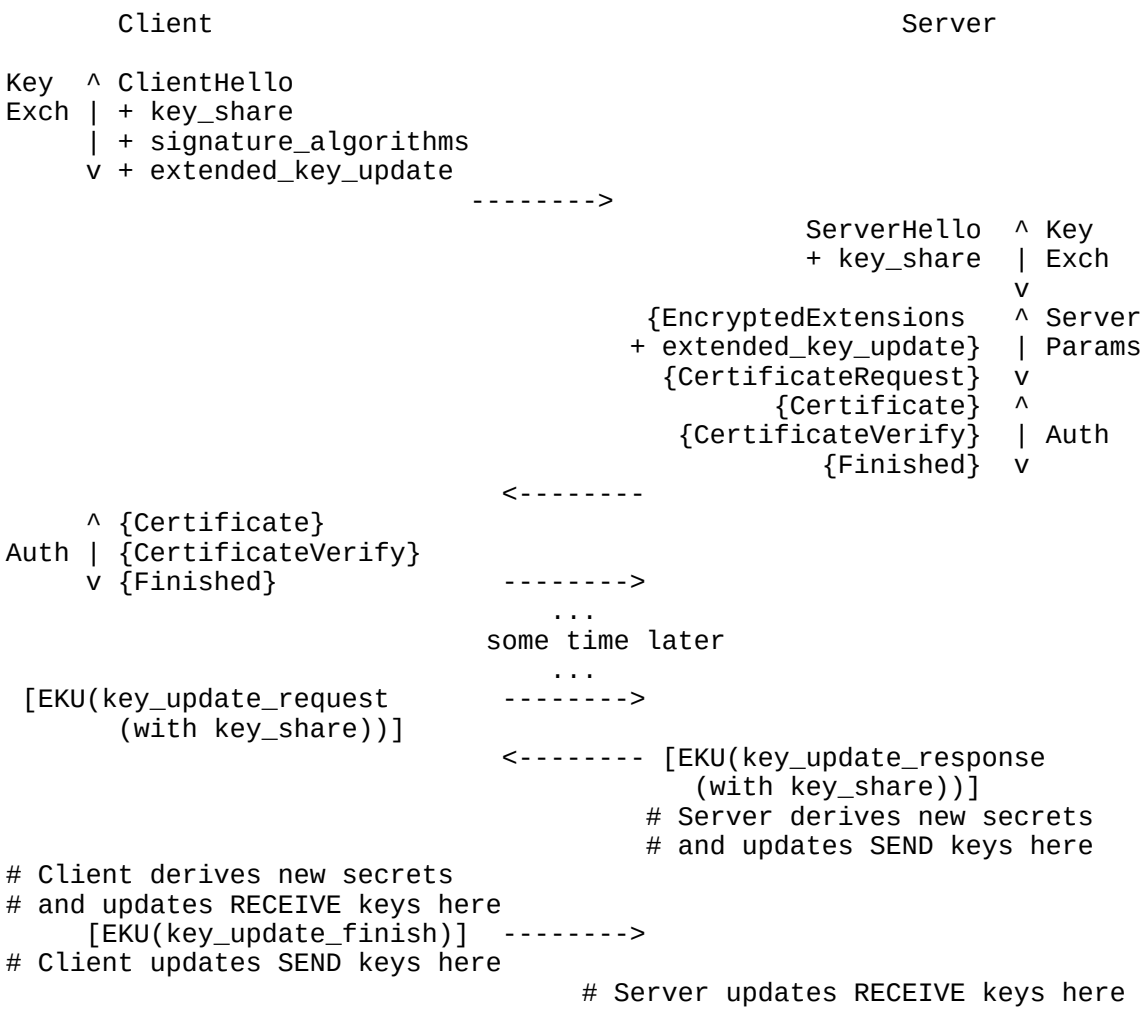


Figure 3: Extended Key Update Example.

6. DTLS 1.3 Considerations

Unlike TLS 1.3, DTLS 1.3 implementations must take into account that handshake messages are not transmitted over a reliable transport protocol.

EKU messages MUST be transmitted reliably, like other DTLS handshake messages. If necessary, EKU messages MAY be fragmented as described in Section 5.5 of [DTLS]. Due to the possibility of an ExtendedKeyUpdate messages being lost and thereby preventing the sender of that message from updating its keying material, receivers MUST retain the pre-update keying material until receipt and successful decryption of a message using the new keys.

Due to packet loss and/or reordering, DTLS 1.3 peers MAY receive records from an earlier epoch. If the necessary keys are available, implementations SHOULD attempt to process such records; however, they MAY choose to discard them.

The exchange has the following steps:

1. The initiator sends an ExtendedKeyUpdate(key_update_request) message, which contains a key share. While an extended key update is in progress, the initiator MUST NOT initiate further key updates. This message is subject to DTLS handshake retransmission, but delivery is only confirmed when the initiator either receives the corresponding ExtendedKeyUpdate(key_update_response) or an ACK.

2. Upon receipt, the responder sends its own KeyShareEntry in a ExtendedKeyUpdate(key_update_response) message. While an extended key update is in progress, the responder MUST NOT initiate further key updates. The responder MAY defer sending a response if system load or resource constraints prevent immediate processing. In such cases, the responder MUST acknowledge receipt of the key_update_request with an ACK and, once sufficient resources become available, retransmit the key_update_response until it is acknowledged by the initiator. key_update_response and ACK message MAY be received out of order; a late ACK MUST be ignored and MUST NOT affect the extended key update state machine.

3. On receipt of ExtendedKeyUpdate(key_update_response) the initiator derives a secret key based on the exchanged key shares. This message also serves as an implicit acknowledgment of the initiator’s ExtendedKeyUpdate(key_update_request), so no separate ACK is required. The initiator MUST update its receive keys and epoch value. The initiator MUST NOT defer derivation of the secrets.

4. The initiator transmits an ExtendedKeyUpdate(key_update_finish) message. This message is subject to DTLS retransmission until acknowledged.

5. The responder MUST acknowledge the received message by sending an ACK message.

6. After the responder receives the initiator’s ExtendedKeyUpdate(key_update_finish), the responder MUST update its send key and epoch value. With the receipt of that message, the responder MUST also update its receive keys.

7. On receipt of the ACK message, the initiator updates its send key and epoch value. If this ACK is not received, the initiator re-transmits its ExtendedKeyUpdate(key_update_finish) until ACK is received. The key update is complete once this ACK is processed by the initiator.

The handshake framing uses a single HandshakeType for this message (see Figure 4).

```
enum {
    client_hello(1),
    server_hello(2),
    new_session_ticket(4),
    end_of_early_data(5),
    encrypted_extensions(8),
    request_connection_id(9),
    new_connection_id(10),
    certificate(11),
    certificate_request(13),
    certificate_verify(15),
    finished(20),
    key_update(24),
    extended_key_update(TBD),  /* new */
    message_hash(254),
    (255)
} HandshakeType;

struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* bytes in message */
    uint16 message_seq;        /* DTLS-required field */
    uint24 fragment_offset;    /* DTLS-required field */
    uint24 fragment_length;    /* DTLS-required field */
    select (msg_type) {
        case client_hello:          ClientHello;
        case server_hello:          ServerHello;
        case end_of_early_data:     EndOfEarlyData;
        case encrypted_extensions:  EncryptedExtensions;
        case certificate_request:   CertificateRequest;
        case certificate:           Certificate;
        case certificate_verify:    CertificateVerify;
        case finished:              Finished;
        case new_session_ticket:    NewSessionTicket;
        case key_update:            KeyUpdate;
        case extended_key_update:   ExtendedKeyUpdate;
        case request_connection_id: RequestConnectionId;
        case new_connection_id:     NewConnectionId;
    } body;
} DTLSHandshake;
```

Figure 4: DTLS 1.3 Handshake Structure.

6.1. DTLS 1.3 Extended Key Update Example

The following example illustrates a successful extended key update, including how the epochs change during the exchange.

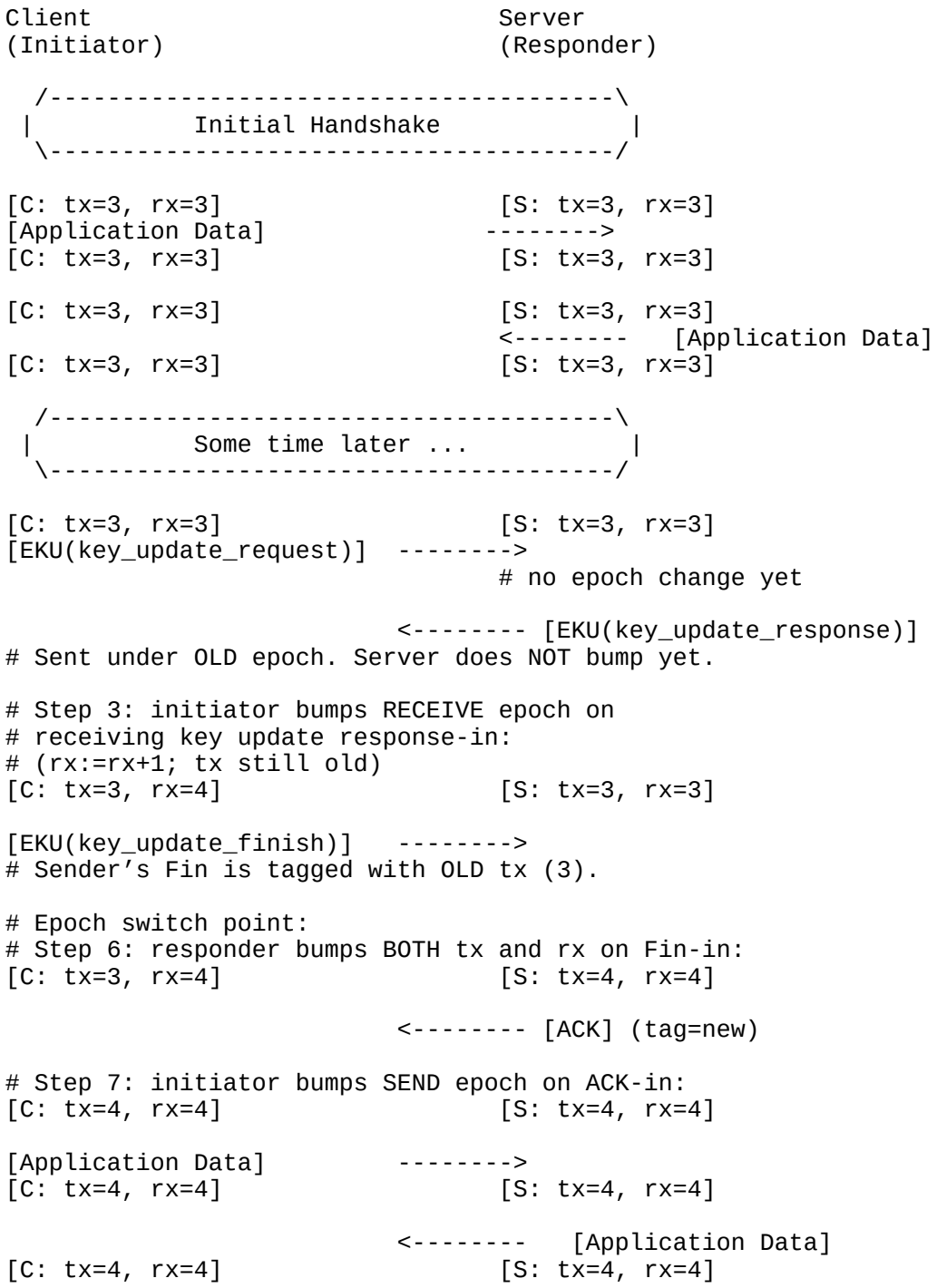

Client
(Initiator)
Server
(Responder)
Initial Handshake
[C: tx=3, rx=3]
[Application Data]
[S: tx=3, rx=3]
[C: tx=3, rx=3]
[S: tx=3, rx=3]
[C: tx=3, rx=3]
[S: tx=3, rx=3]
[Application Data]
[C: tx=3, rx=3]
[S: tx=3, rx=3]
Some time later ...
[C: tx=3, rx=3]
[S: tx=3, rx=3]
[EKU(key_update_request)]
no epoch change yet
[EKU(key_update_response)]
Sent under OLD epoch. Server does NOT bump yet.
Step 3: initiator bumps RECEIVE epoch on
receiving key update response-in:
(rx:=rx+1; tx still old)
[C: tx=3, rx=4]
[S: tx=3, rx=3]
[EKU(key_update_finish)]
Sender’s Fin is tagged with OLD tx (3).
Epoch switch point:
Step 6: responder bumps BOTH tx and rx on Fin-in:
[C: tx=3, rx=4]
[S: tx=4, rx=4]
[ACK] (tag=new)
Step 7: initiator bumps SEND epoch on ACK-in:
[C: tx=4, rx=4]
[S: tx=4, rx=4]
[Application Data]
[C: tx=4, rx=4]
[S: tx=4, rx=4]
[Application Data]
[C: tx=4, rx=4]
[S: tx=4, rx=4]

Figure 5: Example DTLS 1.3 Extended Key Update: Message Exchange.

Figure 6 shows the steps, the message in flight, and the epoch changes on both sides. The A/B -> X/Y notation indicates the change of epoch values for tx/rx before and after the message transmission.

Message	Client tx/rx	Server tx/rx	
APP ---------->	3/3 -> 3/3	3/3 -> 3/3	
<---------- APP	3/3 -> 3/3	3/3 -> 3/3	
req ---------->	3/3 -> 3/3	3/3 -> 3/3	
<---------- resp	3/3 -> 3/4	3/3 -> 3/3	<- step 3
Fin ---------->	3/4 -> 3/4	3/3 -> 4/4	<- step 6
<---------- ACK	3/4 -> 4/4	4/4 -> 4/4	<- step 7
APP ---------->	4/4 -> 4/4	4/4 -> 4/4	
<---------- APP	4/4 -> 4/4	4/4 -> 4/4	

Figure 6: Example DTLS 1.3 Extended Key Update: Epoch Changes.

7. Updating Traffic Keys

When the extended key update message exchange is completed both peers have successfully updated their application traffic keys. The key derivation function described in this document is used to perform this update.

The design of the key derivation function for computing the next generation of application_traffic_secret is motivated by the desire to include

* a secret derived from the (EC)DHE exchange (or from a PQ/T hybrid or post-quantum KEM exchange),

* a secret that allows the new key exchange to be cryptographically bound to the previously established secret,

* a transcript hash that is updated after each Extended Key Update exchange by hashing together the previous transcript hash value with the current ExtendedKeyUpdate(key_update_request) and ExtendedKeyUpdate(key_update_response) messages, which contain the key shares, thereby binding the encapsulated shared secret ciphertext to the IKM in the case of PQ/T hybrid or post-quantum-only key exchange and cryptographically binding the newly derived secrets to the prior handshake transcript and all preceding EKU exchanges, as well as to the current EKU exchange, and

* new label strings to distinguish it from the key derivation used in TLS 1.3.

The transcript_hash_N denotes the transcript hash value associated with generation N. During each Extended Key Update exchange, the transcript hash value for the next generation is computed as follows:

```
transcript_hash_N+1 = Transcript-Hash(transcript_hash_N ||
                                      EKU(key_update_request) ||
                                      EKU(key_update_response))
```

Once transcript_hash_N+1 has been computed, transcript_hash_N can be deleted. No prior transcript hash values need to be retained for future EKU exchanges. transcript_hash_0 denotes the transcript hash of the initial TLS handshake, covering all messages from the ClientHello up to and including the client Finished message.

Figure 7 shows the key derivation hierarchy.

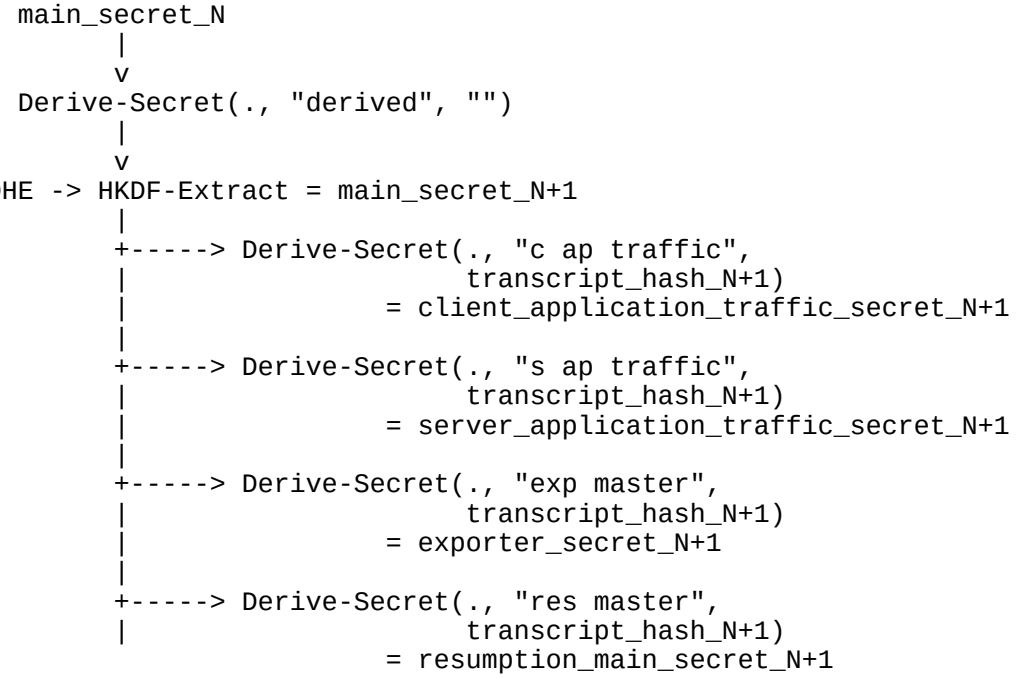


Figure 7: Key Derivation Hierarchy.

During the initial handshake, the Main Secret is generated (see Section 7.1 of [TLS]). Since the main_secret is discarded during the key derivation procedure, a derived value is stored. This stored

value then serves as the input salt to the first key update procedure that incorporates the ephemeral (EC)DHE- established value as input keying material (IKM) to produce main_secret_N+1. The derived value from this new main secret serves as input salt to the subsequent key update procedure, which also incorporates a fresh ephemeral (EC)DHE value as IKM. This process is repeated for each additional key update procedure.

The traffic keys are re-derived from client_application_traffic_secret_N+1 and server_application_traffic_secret_N+1, as described in Section 7.3 of [TLS].

Once client_/server_application_traffic_secret_N+1 and its associated traffic keys have been computed, implementations SHOULD delete client_/server_application_traffic_secret_N and its associated traffic keys as soon as possible. Note: The client_/ server_application_traffic_secret_N and its associated traffic keys can only be deleted by the responder after receiving the ExtendedKeyUpdate(key_update_finish) message.

Once client_/server_application_traffic_secret_N+1 and the corresponding traffic keys are in use, all subsequent records, including alerts and post-handshake messages MUST be protected using those keys.

When using this extension, it is important to consider its interaction with PSK-based resumption using PSKs established via the NewSessionTicket mechanism defined in [TLS].

EKU provides post-compromise recovery for the TLS connection in which it is performed. The recovery guarantees depend on the assumed compromise model. In typical deployment environments, PSKs established via the NewSessionTicket mechanism are generated and reside in memory within the rich operating system. Although such PSKs may subsequently be stored in secure storage, they are exposed during this initial processing window, and compromise of endpoint memory during this period is sufficient to reveal them. Consequently, later protection using secure storage does not prevent their prior compromise. These PSKs are sufficient to establish new authenticated TLS connections. Even if an implementation invalidates previously issued PSKs upon completion of the EKU exchange, an attacker that has already obtained such a PSK may initiate and complete a resumed session prior to that invalidation. In such environments, EKU does not prevent the use of previously issued PSKs.

Accordingly, endpoints that enable EKU MUST disable resumption using PSKs established via the NewSessionTicket mechanism.

8. Post-Quantum Cryptography Considerations

Hybrid key exchange refers to the simultaneous use of multiple key exchange algorithms, with the resulting shared secret derived by combining the outputs of each. The goal of this approach is to maintain security even if all but one of the component algorithms are later found to be vulnerable.

The transition to post-quantum cryptography has motivated the adoption of hybrid key exchanges in TLS, as described in [TLS-HYBRID]. Specific hybrid groups have been registered in [TLS-ECDHE-MLKEM]. When hybrid key exchange is used, the key_exchange field of each KeyShareEntry in the initial handshake is formed by concatenating the key_exchange fields of the constituent algorithms. This same approach is reused during the Extended Key Update, when new key shares are exchanged.

[TLS-MLKEM] registers the lattice-based ML-KEM algorithm and its variants, such as ML-KEM-512, ML-KEM-768 and ML-KEM-1024. The KEM encapsulation key or KEM ciphertext is represented as a 'KeyShareEntry' field. This same approach is reused during the Extended Key Update, when new key shares are exchanged.

9. SSLKEYLOGFILE Update

As a successful extended key update exchange invalidates previous secrets, SSLKEYLOGFILE [TLS-KEYLOGFILE] needs to be populated with new entries. As a result, two additional secret labels are utilized in the SSLKEYLOGFILE:

1. CLIENT_TRAFFIC_SECRET_N+1: identifies the client_application_traffic_secret_N+1 in the key schedule

2. SERVER_TRAFFIC_SECRET_N+1: identifies the server_application_traffic_secret_N+1 in the key schedule

3. EXPORTER_SECRET_N+1: identifies the exporter_secret_N+1 in the key schedule

Similar to other entries in the SSLKEYLOGFILE, the label is followed by the 32-byte value of the Random field from the ClientHello message that established the TLS connection, and the corresponding secret encoded in hexadecimal.

SSLKEYLOGFILE entries for the extended key update MUST NOT be produced if SSLKEYLOGFILE was not used for other secrets in the handshake.

Note that each successful Extended Key Update invalidates all previous SSLKEYLOGFILE secrets including past iterations of CLIENT_TRAFFIC_SECRET_, SERVER_TRAFFIC_SECRET_ and EXPORTER_SECRET_.

10. Exporter

10.1. Post-Compromise Security for the Initial Exporter Secret

The TLS 1.3 Key Schedule, see Figure 5 of [TLS], derives the exporter_secret from the main secret. This exporter_secret is static for the lifetime of the connection and is not updated by a standard key update.

A core design goal of this specification is not met if the exporter_secret does not change. Therefore, this document defines an exporter interface that derives a fresh exporter secret whenever new application traffic keys are updated through the EKU.

If the initial exporter secret for this new interface were identical to exporter_secret, then compromising exporter_secret at any point during the lifetime of the connection would enable an attacker to recompute all exporter outputs derived from it. This would break post-compromise security for exported keying material.

Therefore, the initial exporter secret used by the exporter interface defined in this document, i.e., the exporter output available prior to the first Extended Key Update, MUST be distinct from the exporter_secret. This separation ensures that compromise of the TLS exporter interface does not compromise outputs derived from the exporter interface defined in this document.

Prior to the first Extended Key Update, the exporter interface provides an initial exporter secret, denoted exporter_secret_0. This secret is derived from the TLS main secret and the handshake transcript, but is cryptographically independent of the TLS exporter_secret. It is computed as follows:

```
exporter_secret_0 =
Derive-Secret(Main Secret,
"exporter eku",
Transcript-Hash(ClientHello..server Finished))
```

Applications that require post-compromise security MUST use the exporter interface defined in this document. This exporter interface is independent of the TLS exporter defined in Section 7.5 of [TLS], which continues to use a static exporter_secret for the lifetime of the connection for compatiblity with "legacy" applications.

10.2. Exporter Usage After Extended Key Update

Protocols such as DTLS-SRTP and DTLS-over-SCTP rely on TLS or DTLS for key establishment, but reuse portions of the derived keying material for their own specific purposes. These protocols use the TLS exporter defined in Section 7.5 of [TLS]. Exporters are also used for deriving authentication related values such as nonces, as described in [RFC9729].

Once the Extended Key Update mechanism is complete, such protocols would need to use the newly derived exporter secret to generate Exported Keying Material (EKM) to protect packets. The "sk" derived in the Section 7 will be used as the "Secret" in the exporter function, defined in Section 7.5 of [TLS], to generate EKM, ensuring that the exported keying material is aligned with the updated security context. The newly derived exporter secret is cryptographically independent of previous exporter secrets.

When a new exporter secret becomes active following a successful Extended Key Update, the TLS or DTLS implementation would have to provide an asynchronous notification to the application indicating that:

* A new epoch has become active, and the (D)TLS implementation can include the corresponding epoch identifier. Applications receiving an epoch identifier can use it to request keying material for that specific epoch through an epoch-aware exporter interface. In TLS, this identifier represents a local logical counter that may differ between peers.

Applications are notified that a new epoch is active only after both peers have completed the Extended Key Update exchange and switched to the new traffic keys.

* In TLS, the initiator triggers notification after Step 5 in Section 5, and the responder triggers notification after Step 4 in Section 5.

* In DTLS, the initiator triggers notification to the application after Step 7 in Section 6, and the responder triggers notification after Step 9 in Section 6.

The corresponding EKM is obtained by the application through the TLS/DTLS exporter interface using its chosen label and context values as defined in Section 4 of [RFC5705].

To prevent desynchronization, the application will have to retain both the previous and the newly derived exporter secrets for a short period. For TLS, the previous exporter secret would be discarded once data derived from the new exporter has been successfully processed, and no records protected with the old exporter secret are expected to arrive. For DTLS, the previous exporter secret needs to be retained until the retention timer expires for the prior epoch, to allow for processing of packets that may arrive out of order. The retention policy for exporter secrets is application-specific. For example, in DTLS-SRTP, the application might retain the previous exporter secret until its replay window no longer accepts packets protected with keys derived from that secret, as described in Section 3.3.2 of [RFC3711].

11. Use of Post-Handshake Authentication and Exported Authenticators with Extended Key Update

EKU provides fresh traffic secrets, but EKU alone does not authenticate that both endpoints derived the same updated keys. An attacker that temporarily compromises an endpoint may later act as an active MitM capable of interfering with the EKU exchange. Such an attacker can cause the peers to transition to divergent traffic secrets without detection, but cannot compromise the endpoint to derive secrets after the new epoch is established. To confirm that both peers transitioned to the same new key state, TLS 1.3 provides two mechanisms: Post-Handshake Certificate-Based Client Authentication and Exported Authenticators [RFC9261].

11.1. Post-Handshake Certificate-Based Client Authentication

When Post-Handshake Certificate-Based Client Authentication (Section 4.6.2 of [TLS]) is performed after an Extended Key Update (EKU) is complete, the Handshake Context used for the transcript hash is updated. It consists of transcript_hash_N+1 concatenated with the CertificateRequest message. The Finished message is computed using a MAC key derived from the Base Key of the new epoch (client_application_traffic_secret_N+1). This confirms that both peers are operating with the same updated traffic keys and completes an authenticated transition after the EKU.

11.2. Exported Authenticators

This document updates Section 5.1 of [RFC9261] to specify that, after an Extended Key Update has completed, the Handshake Context and Finished MAC Key used for Exported Authenticators MUST be derived from the exporter secret associated with the current epoch. Implementations that support the epoch-aware Exported Authenticators interface MUST provide a means for applications to request the

generation or validation of Exported Authenticators using the exporter secret for a specific epoch.

The Handshake Context and Finished MAC Key used in both the CertificateVerify message (Section 5.2.2 of [RFC9261]) and the Finished message (Section 5.2.3 of [RFC9261]) are derived from the exporter secret associated with the current epoch. If a MitM interferes with the EKU exchange and causes the peers to derive different traffic and exporter secrets, their Handshake Contexts and Finished MAC Keys will differ. As a result, validation procedures specified in Section 5.2.4 of [RFC9261] will fail, thereby detecting the divergence of key state between peers.

A new optional API SHOULD be defined to permit applications to request or verify Exported Authenticators for a specific exporter epoch. As discussed in Section 7 of [RFC9261], this can, as an exception, be implemented at the application layer when the epoch-aware TLS exporter is available. The APIs defined in [RFC9261] remain unchanged, so existing applications continue to operate without modification. The epoch-aware API accepts an epoch identifier; when present, the (D)TLS implementation MUST derive the Handshake Context and Finished MAC Key from the exporter secret associated with that epoch. When Exported Authenticators are generated using the epoch-aware Exported Authenticators interface, the epoch identifier used for their derivation can be conveyed in the certificate_request_context field, allowing the peer, particularly in DTLS where records may be reordered, to determine the correct exporter secret for validation.

11.3. Interaction of Extended Key Update and Post-Handshake Authentication

EKU and post-handshake authentication may both occur during the lifetime of a (D)TLS connection. Post-Handshake Certificate-Based Client Authentication (PHA) is bound to the handshake transcript and computes its Finished message as specified in Section 4.4 of [TLS], using a key derived from the application traffic secret that is active at the time the CertificateRequest is sent. Therefore, specific ordering constraints are required to preserve cryptographic consistency.

11.3.1. Post-Handshake Certificate-Based Client Authentication

An endpoint MUST NOT complete an EKU exchange in a manner that transitions to new application traffic secrets while a PHA exchange is in progress.

The following constraints apply to both TLS 1.3 and DTLS 1.3:

* An endpoint MUST NOT initiate PHA while an EKU exchange is in progress.

* If PHA has been initiated and the corresponding authentication exchange has not yet completed, neither endpoint MUST initiate an EKU exchange.

* In a cross-flight condition, if a (D)TLS client sends an EKU request and, before receiving a response, receives a CertificateRequest from the (D)TLS server, the endpoints MUST defer completion of the EKU exchange and proceed with the post-handshake authentication exchange. The endpoints MUST NOT transition to new application traffic secrets until the authentication exchange has completed.

In DTLS, deferred EKU request is acknowledged as specified in Section 6.

11.3.2. Exported Authenticators

Because the exporter interface defined in this document is epoch-aware, the exporter secret used for an Exported Authenticator exchange is explicitly determined by the epoch selected by the application.

As a result, cross-flight exchanges of EKU and AuthenticatorRequest messages do not introduce cryptographic ambiguity. Therefore, no serialization requirement is imposed between EKU and Exported Authenticator exchanges.

12. Security Considerations

This section discusses additional security and operational aspects introduced by the Extended Key Update mechanism. All security considerations of TLS 1.3 [TLS] and DTLS 1.3 [DTLS] continue to apply.

12.1. Scope of Key Compromise

Extended Key Update (EKU) assumes a transient compromise of the current application traffic keys, rather than a persistent attacker with ongoing access to key material. The EKU procedure does not rely on long-term private keys, which may be stored in a secure element (e.g., a Hardware Security Module (HSM)) or within the rich OS. Moreover, in security-critical scenarios, these long-term private keys are typically stored separately from the primary secret or the traffic keys.

Two threat scenarios are relevant:

1. The long-term private key remains secure, while application traffic keys in the rich operating system are temporarily exposed. EKU addresses this case for the current TLS connection.

2. If the long-term private key in the rich OS is compromised, EKU can still protect the current TLS connection by updating the main secret and traffic keys. However, all future TLS connections are at risk, as the attacker can impersonate the endpoint using the stolen private key.

Extended Key Update can restore confidentiality only if the attacker no longer has access to either peer. If an adversary retains access to current application traffic keys and can act as a man-in-the-middle during the Extended Key Update, then the update cannot restore security unless Section 11 is used.

If one of the mechanisms defined in Section 11 is not used, the attacker can impersonate each endpoint, substitute EKU messages, and maintain control of the communication. When Post-handshake Certificate-Based Client Authentication or the modified Exported Authenticator mechanism is used, the authentication messages are bound to the keys established after the EKU. Any modification or substitution of EKU messages therefore becomes detectable, preventing this attack.

If a compromise occurs before the handshake completes, the ephemeral key exchange, client_handshake_traffic_secret, server_handshake_traffic_secret, and the initial client_/server_application_traffic_secret could be exposed. In that case, only the initial handshake messages and the application data encrypted under the initial client_/server_application_traffic_secret can be decrypted until the Extended Key Update procedure completes. The Extended Key Update procedure derives fresh application_traffic_secrets from a new ephemeral key exchange, ensuring that all subsequent application data remains confidential.

12.2. Post-Compromise Security

Extended Key Update provides post-compromise security for long-lived TLS sessions. To ensure post-compromise security guarantees:

* Each update MUST use freshly generated ephemeral key-exchange material. Implementations MUST NOT reuse ephemeral key-exchange material across updates or across TLS sessions.

12.3. Denial-of-Service (DoS)

The Extended Key Update mechanism increases computational and state-management overhead. A malicious peer could attempt to exhaust CPU or memory resources by initiating excessive update requests.

Implementations SHOULD apply the following mitigations:

* Limit the frequency of accepted Extended Key Update requests per session.

* A peer that has sent an Extended Key Update MUST NOT initiate another until the previous update completes. If a peer violates this rule, the receiving peer MUST treat it as a protocol violation, send an "unexpected_message" alert, and terminate the connection.

12.4. Operational Guidance

Deployments SHOULD evaluate Extended Key Update performance under load and fault conditions, such as high-frequency or concurrent updates. TLS policies SHOULD define explicit rate limits that balance post-compromise security benefits against potential DoS exposure.

13. IANA Considerations

13.1. TLS Flags

IANA is requested to add the following entry to the "TLS Flags" extension registry [TLS-Ext-Registry]:

* Value: TBD1

* Flag Name: extended_key_update

* Messages: CH, EE

* Recommended: Y

* Reference: [This document]

13.2. TLS HandshakeType

IANA is requested to add the following entry to the "TLS HandshakeType" registry [TLS-Param-Registry]:

* Value: TBD2

* Description: extended_key_update

* DTLS-OK: Y

* Reference: [This document]

13.3. ExtendedKeyUpdate Message Subtypes Registry

IANA is requested to create a new registry "TLS ExtendedKeyUpdate Message Subtypes", within the existing "Transport Layer Security (TLS) Parameters" registry group [TLS-Param-Registry]. This new registry reserves types used for Extended Key Update entries. The initial contents of this registry are as follows.

Value	Description	DTLS-OK	Reference
0	key_update_request	Y	This document
1	key_update_response	Y	This document
2	key_update_finish	Y	This document
3-255	Unassigned		

Table 1

New assignments in the "TLS ExtendedKeyUpdate Types" registry will be administered by IANA through Specification Required procedure [RFC8126]. The role of the designated expert is described in Section 17 of [RFC8447]. The designated expert [RFC8126] ensures that the specification is publicly available. It is sufficient to have an Internet-Draft (that is posted and never published as an RFC) or to cite a document from another standards body, industry consortium, or any other location. An expert may provide more in-depth reviews, but their approval should not be taken as an endorsement of the ExtendedKeyUpdate Message Subtype.

13.4. SSLKEYLOGFILE labels

IANA is requested to add the following entries to the "TLS SSLKEYLOGFILE Labels" extension registry [TLS-Ext-Registry]:

Value	Description	Reference	Comment
CLIENT_TRAFFIC_SECRET_N+1	Secret protecting client records after Extended Key Update	This document	N represents iteration of Extended Key Update
SERVER_TRAFFIC_SECRET_N+1	Secret protecting server records after Extended Key Update	This document	N represents iteration of Extended Key Update
EXPORTER_SECRET_N+1	Exporter secret after Extended Key Update	This document	N represents iteration of Extended Key Update

Table 2

14. References

14.1. Normative References

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

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

[RFC3711] Baugher, M., McGrew, D., Naslund, M., Carrara, E., and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)", RFC 3711, DOI 10.17487/RFC3711, March 2004, <https://www.rfc-editor.org/rfc/rfc3711>.

[RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, DOI 10.17487/RFC5705, March 2010, <https://www.rfc-editor.org/rfc/rfc5705>.

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

[RFC9261] Sullivan, N., "Exported Authenticators in TLS", RFC 9261, DOI 10.17487/RFC9261, July 2022, <https://www.rfc-editor.org/rfc/rfc9261>.

[RFC9729] Schinazi, D., Oliver, D., and J. Hoyland, "The Concealed HTTP Authentication Scheme", RFC 9729, DOI 10.17487/RFC9729, February 2025, <https://www.rfc-editor.org/rfc/rfc9729>.

[TLS] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-rfc8446bis-14, 13 September 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-rfc8446bis-14>.

[TLS-FLAGS] Nir, Y., "A Flags Extension for TLS 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-tlsflags-17, 17 March 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-tlsflags-17>.

14.2. Informative References

[ANSSI] ANSSI, "Recommendations for securing networks with IPsec, Technical Report", August 2015, <https://cyber.gouv.fr/sites/default/files/2012/09/NT_IPsec_EN.pdf>.

[CCG16] IEEE, "On Post-compromise Security", August 2016, <https://doi.org/10.1109/csf.2016.19>.

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

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

[RFC8447] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <https://www.rfc-editor.org/rfc/rfc8447>.

[RFC9794] Driscoll, F., Parsons, M., and B. Hale, "Terminology for Post-Quantum Traditional Hybrid Schemes", RFC 9794, DOI 10.17487/RFC9794, June 2025, <https://www.rfc-editor.org/rfc/rfc9794>.

[TLS-ECDHE-MLKEM] Kwiatkowski, K., Kampanakis, P., Westerbaan, B., and D. Stebila, "Post-quantum hybrid ECDHE-MLKEM Key Agreement for TLSv1.3", Work in Progress, Internet-Draft, draft-ietf-tls-ecdhe-mlkem-04, 8 February 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-ecdhe-mlkem-04>.

[TLS-Ext-Registry] IANA, "Transport Layer Security (TLS) Extensions", November 2023, <https://www.iana.org/assignments/tls-extensiontype-values>.

[TLS-HYBRID] Stebila, D., Fluhrer, S., and S. Gueron, "Hybrid key exchange in TLS 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-hybrid-design-16, 7 September 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-hybrid-design-16>.

[TLS-KEYLOGFILE] Thomson, M., Rosomakho, Y., and H. Tschofenig, "The SSLKEYLOGFILE Format for TLS", Work in Progress, Internet-Draft, draft-ietf-tls-keylogfile-05, 9 June 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-keylogfile-05>.

[TLS-MLKEM] Connolly, D., "ML-KEM Post-Quantum Key Agreement for TLS 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-mlkem-07, 12 February 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-mlkem-07>.

[TLS-Param-Registry]
IANA, "Transport Layer Security (TLS) Parameters", November 2023, <https://www.iana.org/assignments/tls-parameters>.

[TLS-RENEGOTIATION]
Rescorla, E., Ray, M., Dispensa, S., and N. Oskov, "Transport Layer Security (TLS) Renegotiation Indication Extension", RFC 5746, DOI 10.17487/RFC5746, February 2010, <https://www.rfc-editor.org/rfc/rfc5746>.

Appendix A. Acknowledgments

We would like to thank the members of the "TSVWG DTLS for SCTP Requirements Design Team" for their discussion. The members, in no particular order, were:

* Marcelo Ricardo Leitner
* Zaheduzzaman Sarker
* Magnus Westerlund
* John Mattsson
* Claudio Porfiri
* Xin Long
* Michael Tテシxen
* Hannes Tschofenig
* K Tirumaleswar Reddy
* Bertrand Rault

Additionally, we would like to thank the chairs of the Transport and Services Working Group (tsvwg) Gorry Fairhurst and Marten Seemann as well as the responsible area director Martin Duke.

Finally, we would like to thank Martin Thomson, Ilari Liusvaara, Benjamin Kaduk, Scott Fluhrer, Dennis Jackson, David Benjamin, Matthijs van Duin, Rifaat Shekh-Yusef, Joe Birr-Pixton, Eric Rescorla, and Thom Wiggers for their review comments.

Appendix B. State Machines

The sections below describe the state machines for the extended key update operation for TLS 1.3 and DTLS 1.3.

The state machine diagrams in the Appendix are provided for illustrative purposes only to aid understanding of the protocol flow. They are not normative. In case of any discrepancy between the Appendix diagrams and the protocol behavior specified in the main body of this document, the text in the draft takes precedence.

For editorial reasons we abbreviate the protocol message types:

* Req - ExtendedKeyUpdate(request)

* Resp - ExtendedKeyUpdate(response)

* Fin - ExtendedKeyUpdate(key_update_finish)

* ACK - Acknowledgement message from Section 7 of [DTLS]

* APP - application data payloads

In the (D)TLS 1.3 state machines discussed below, the terms SEND and RECEIVE keys refer to the send and receive key variables defined in Section 2.

The TLS state machine diagrams use the following notation:

* Numbered labels (1), (2), 寰ヲ denote normal protocol transitions in the order they occur.

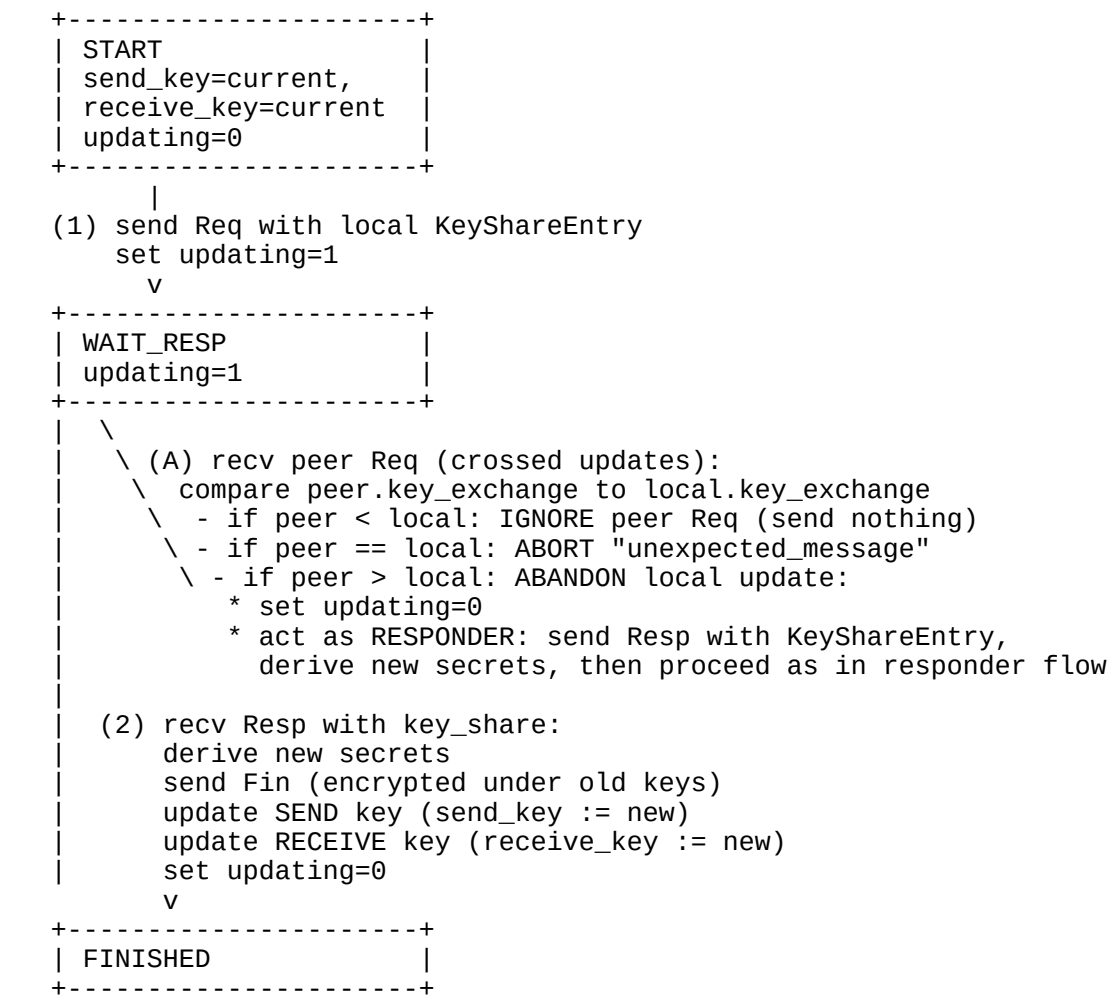
* Lettered labels (A), (B), 寰ヲ denote exceptional or conditional transitions.

* Self-loops indicate that the state does not change as a result of the event.

B.1. TLS 1.3 State Machines

This section describes the initiator and responder state machines.

B.1.1. Initiator State Machine



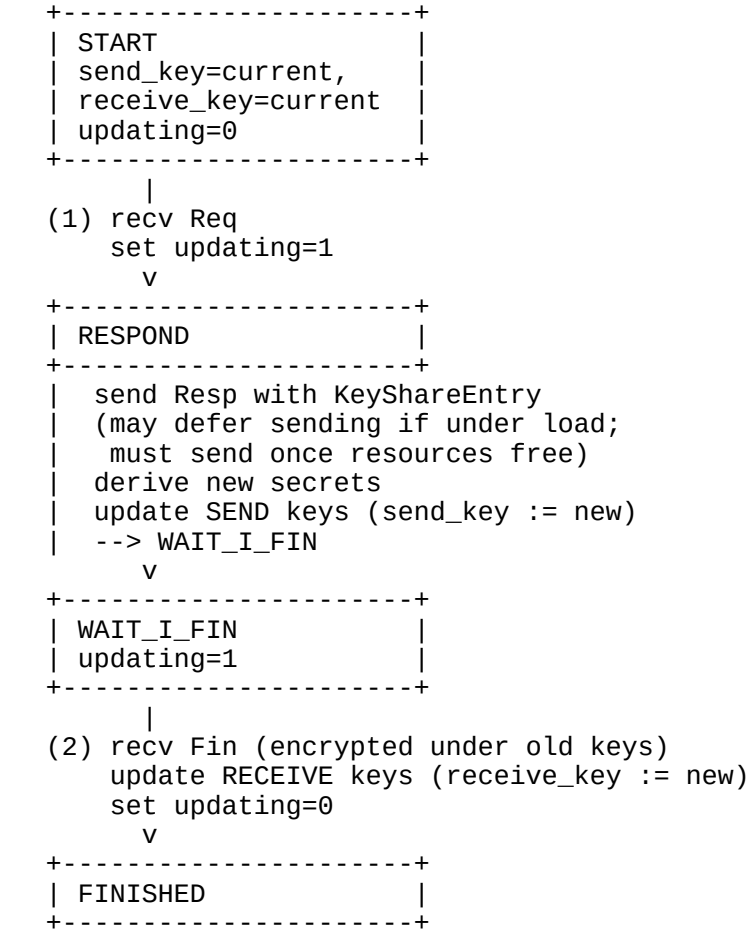
Notes:

* Fin message is sent under old keys.

* If a classic KeyUpdate arrives (EKU negotiated), ABORT "unexpected_message".

* Crossed-requests: ignore the request with LOWER lexicographic key_exchange; if equal, abort.

B.1.2. Responder State Machine



Notes:

* Responder may defer Resp under load (no status reply), then must send it once resources are free.

* If a classic KeyUpdate arrives (EKU negotiated), ABORT "unexpected_message".

B.2. DTLS 1.3 State Machines

This section describes the initiator and responder state machines.

B.2.1. Terms and Abbreviations

The following variables and abbreviations are used in the state machine diagrams.

* rx - current, accepted receive epoch.
* tx - current transmit epoch used for tagging outgoing messages.
* E - initial epoch value.
* updating - true while a key-update handshake is in progress.
* old_rx - the previous receive epoch remembered during retention.
* retain_old - when true, receiver accepts tags old_rx and rx.
* tag=... - the TX-epoch value written on an outgoing message.
* e==... - the epoch tag carried on an incoming message (what the peer sent).
* FINISHED / START / WAIT_RESP / WAIT_I_FIN / WAIT_R_FIN / ACTIVATE RETENTION / RESPOND / WAIT_ACK - diagram states; FINISHED denotes the steady state after success.

Crossed requests. If both peers independently initiate the extended key update and the key_update_request messages cross in flight, compare the KeyShareEntry.key_exchange values. The request with the lower lexicographic value must be ignored. If the values are equal, the endpoint must abort with an "unexpected_message" alert. If the peer’s value is higher than the local one, the endpoint abandons its in-flight update and processes the peer’s request as responder.

B.2.2. State Machine (Initiator)

The initiator starts in the START state with matching epochs (rx := E; tx := E). It sends a Req and enters WAIT_RESP (updating := 1). While waiting, APP data may be sent at any time (tagged with the current tx) and received according to the APP acceptance rule below.

Once the responder returns Resp with a tag matching the current rx, the initiator derives new key material. It then sends Fin still tagged with the old tx. The initiator activates retention mode: the old epoch is remembered, the receive epoch is incremented, and application data is accepted under both epochs for a transition period. Initiator moves to WAIT_ACK.

If a peer key_update_request arrives while in WAIT_RESP (crossed updates), apply the crossed-request rule above. If the peer's key_exchange is higher, abandon the local update (updating := 0) and continue as responder: send key_update_response, derive new secrets, then proceed with the responder flow. If lower, ignore the peer's request; if equal, abort with "unexpected_message".

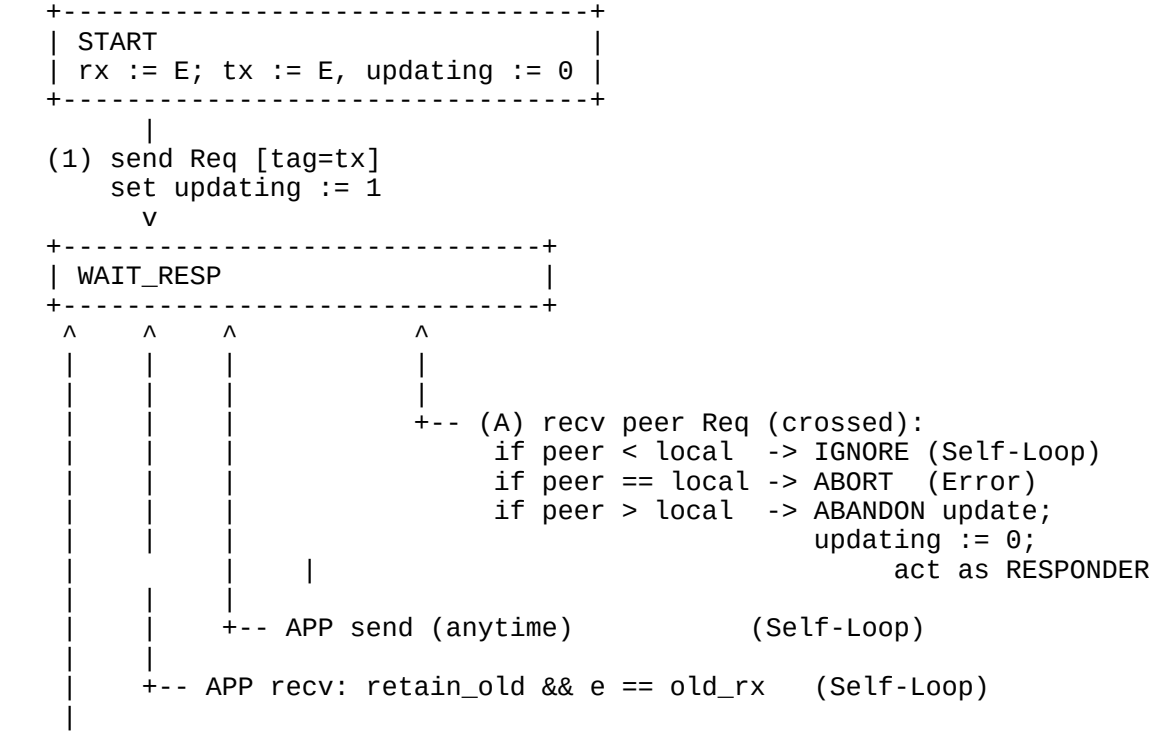
Upon receiving the responder's ACK matching the updated epoch, the initiator completes the transition by synchronizing transmit and receive epochs (tx := rx), disabling retention, and clearing the update flag. The state machine returns to FINISHED, ready for subsequent updates.

Throughout the process:

* Duplicate messages are tolerated (for retransmission handling).

* Temporary epoch mismatches are permitted while an update is in progress.

* Application data flows continuously, subject to epoch acceptance rules.



B.2.3. State Machine (Responder)

The responder starts in the START state with synchronized transmit and receive epochs (rx := E; tx := E) and no update in progress. Application data can be transmitted at any time using the sender's current transmit epoch. A receiver must accept application data if the epoch tag on the DTLS record equals the receiver's current receive epoch. If the receiver has retention active (retain_old == true), the receiver must also accept DTLS records whose epoch tag equals the remembered previous epoch.

Upon receiving an ExtendedKeyUpdate(key_update_request) (Req), the responder transitions to RESPOND. The responder may defer sending ExtendedKeyUpdate(key_update_response) under load; in that case it must acknowledge the request with an ACK and retransmit the response until it is acknowledged by the initiator, as specified in DTLS considerations. When sent, ExtendedKeyUpdate(key_update_response) is tagged with the current tx.

After an ExtendedKeyUpdate(key_update_finish) (Fin) is received with the correct epoch, the responder:

1. activates retention (old_rx := rx; retain_old := 1),

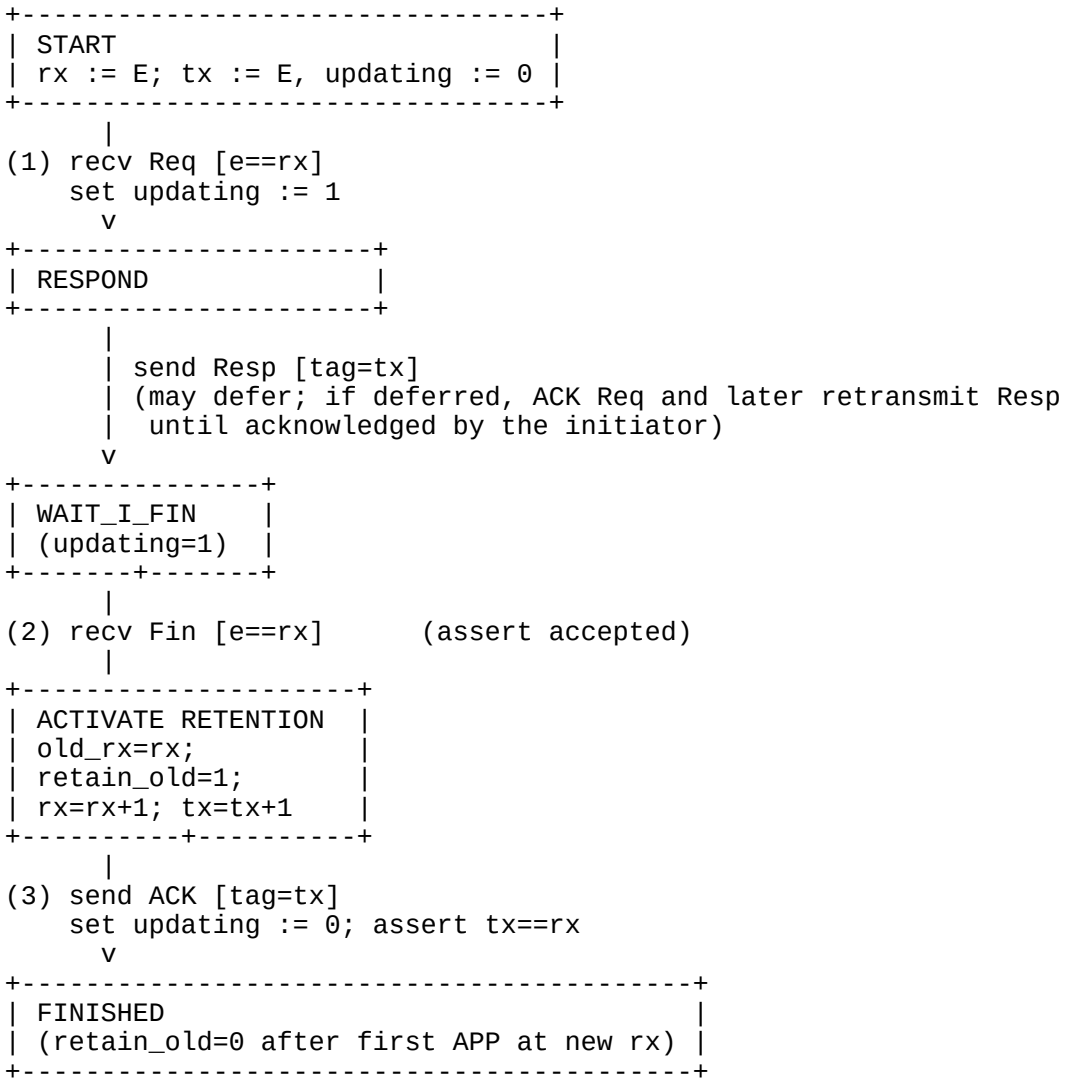
2. increments both epochs (rx++, tx++),

3. sends ACK tagged with the new tx (which now equals the new rx),

4. clears updating and enters FINISHED.

Retention at the responder ends automatically on the first APP received under the new rx (then retain_old := 0). APP traffic is otherwise permitted at any time; reordering is tolerated by the acceptance rule.

APP acceptance rule (receiver): accept if e == rx or (retain_old && e == old_rx). If retain_old is set and an APP with the new rx arrives, clear retain_old.



Appendix C. Overview of Security Goals

A complete security analysis of the EKU is outside the scope of this document. This appendix provides an informal description of the primary security goals that EKU is designed to achieve.

C.1. Post-Compromise Security (PCS)

Extended Key Update supports post-compromise security under the assumptions described in Section 12.1. If an attacker temporarily compromises an endpoint and obtains the traffic keys in use before an Extended Key Update takes place, but the compromise does not persist after the EKU completes, the attacker cannot derive the new keying material established by EKU. This property follows from the use of fresh ephemeral key exchange material during each Extended Key Update, which produces new traffic keys that are independent of the previous ones. This property provides only best-effort post-compromise security, as it assumes the attacker is not acting as a MiTM during the Extended Key Update.

As a result, confidentiality of application data encrypted after the Extended Key Update is preserved even if the earlier traffic keys were exposed.

C.2. Key Freshness and Cryptographic Independence

Each Extended Key Update derives new traffic keys from ephemeral key exchange material. This ensures strong separation between successive traffic keys:

* The new traffic keys established by an Extended Key Update are independent of the previous traffic keys.

* Compromise of one of traffic keys does not allow recovery of any earlier or later traffic keys.

* Application data protected under one of the traffic keys cannot be decrypted using keys from another.

C.3. Elimination of Standard KeyUpdate

Once Extended Key Update has been negotiated for a session, peers rely exclusively on EKU rather than the standard TLS 1.3 KeyUpdate mechanism. Relying solely on Extended Key Update helps maintain PCS properties throughout the lifetime of the TLS session.

C.4. Detecting Divergent Key State

As described in Section 11, both Post-handshake Certificate-Based Client Authentication and Exported Authenticators can be used after an Extended Key Update to confirm that both endpoints derived the same traffic keys. Because the authentication messages produced by these mechanisms depend on values derived from the updated traffic keys, any divergence in those traffic keys causes validation to fail,

revealing interference by an active attacker.

Authors' Addresses

Hannes Tschofenig
Siemens
Email: hannes.tschofenig@gmx.net

Michael Tテシxen
Mテシnster Univ. of Applied Sciences
Email: tuexen@fh-muenster.de

Tirumaleswar Reddy
Nokia
Email: kondtir@gmail.com

Steffen Fries
Siemens
Email: steffen.fries@siemens.com

Yaroslav Rosomakho
Zscaler
Email: yrosomakho@zscaler.com