Transport Layer Security
Internet-Draft
Intended status: Standards Track
Expires: 6 May 2026

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

2 November 2025

Extended Key Update for Transport Layer Security (TLS) 1.3
draft-ietf-tls-extended-key-update-07

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 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/.

Copyright Notice

Table of Contents

1. Introduction

The Transport Layer Security (TLS) 1.3 protocol provides forward secrecy by using an ephemeral Diffie-Hellman (DHE) key exchange during the initial handshake. 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 secret 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 long-term secrets or derive new secrets from fresh DHE input.

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 a long-term 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. Unlike the standard key update, this mechanism allows peers to perform a fresh Diffie-Hellman exchange within an active session using one of the groups negotiated during the initial handshake. By periodically rerunning (EC)DHE, this extension enables the derivation of new traffic secrets that are independent of prior key material. 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.

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 secrets but is unable to compromise the long-term secret.

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)

* new_key_update (2)

New ExtendedKeyUpdate message subtypes are assigned by IANA as described in Section 12.3.

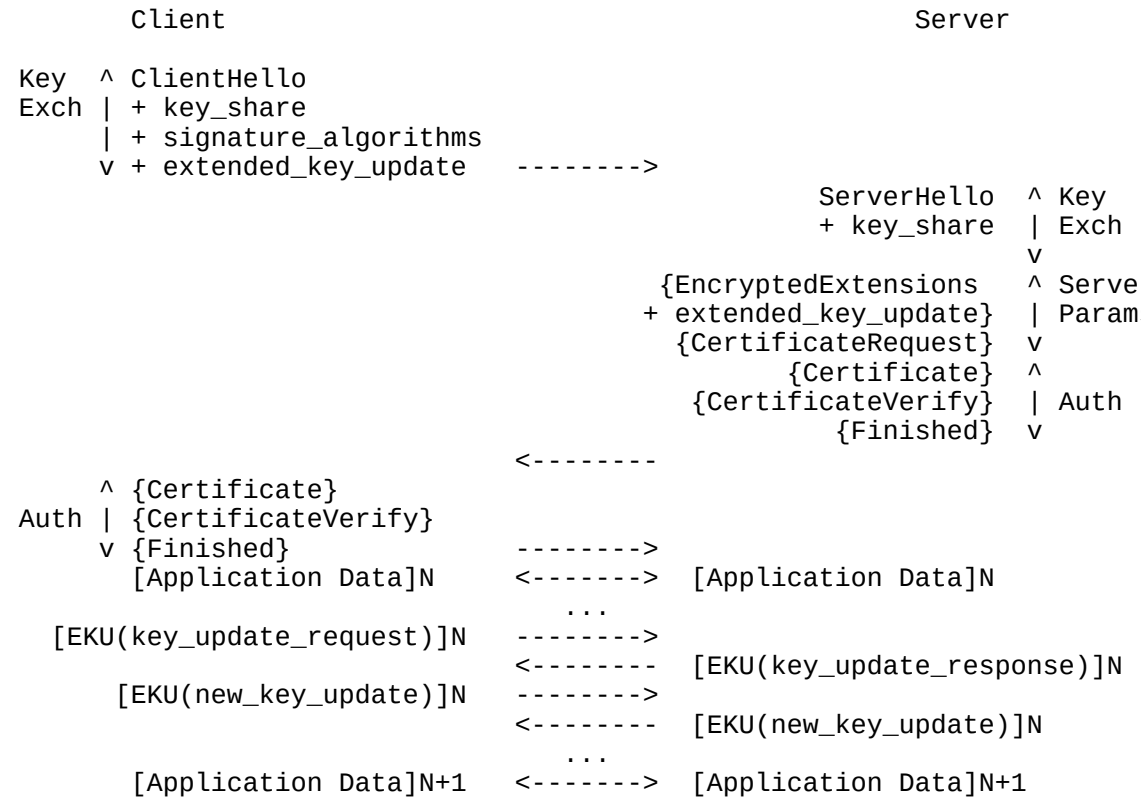
A TLS peer which receives a ExtendedKeyUpdate with an unexpected message subtype MUST abort the connection with an "unexpected_message" alert.

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),
   new_key_update(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 new_key_update: {
          /* 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. Upon receipt of an ExtendedKeyUpdate(key_update_response) the initiator derives the new secrets from the exchanged key shares. The initiator then sends an empty ExtendedKeyUpdate(new_key_update) message to trigger the switch to the new keys.

4. After the initiator sends ExtendedKeyUpdate(new_key_update) it MUST update its send keys. Upon receipt of this message, the responder MUST update its receive keys and then send ExtendedKeyUpdate(new_key_update), after which it MUST update its send keys.

5. After receiving the responder宴洀 ExtendedKeyUpdate(new_key_update), the initiator MUST update its receive keys.

Both sender and receiver MUST encrypt their ExtendedKeyUpdate(new_key_update) messages with the old keys. Both sides MUST ensure that the new_key_update encrypted with the old key is received before accepting any messages encrypted with the new key.

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

While Figure 1 shows the high-level interaction between a TLS 1.3 client and server, this section shows an example message exchange with information about the updated keys added.

There are two phases:

1. The support for the functionality in this specification is negotiated in the ClientHello and the EncryptedExtensions messages.

2. Once the initial handshake is completed, a key update can be triggered.

Figure 3 provides an overview of the exchange starting with the initial negotiation followed by the key update.

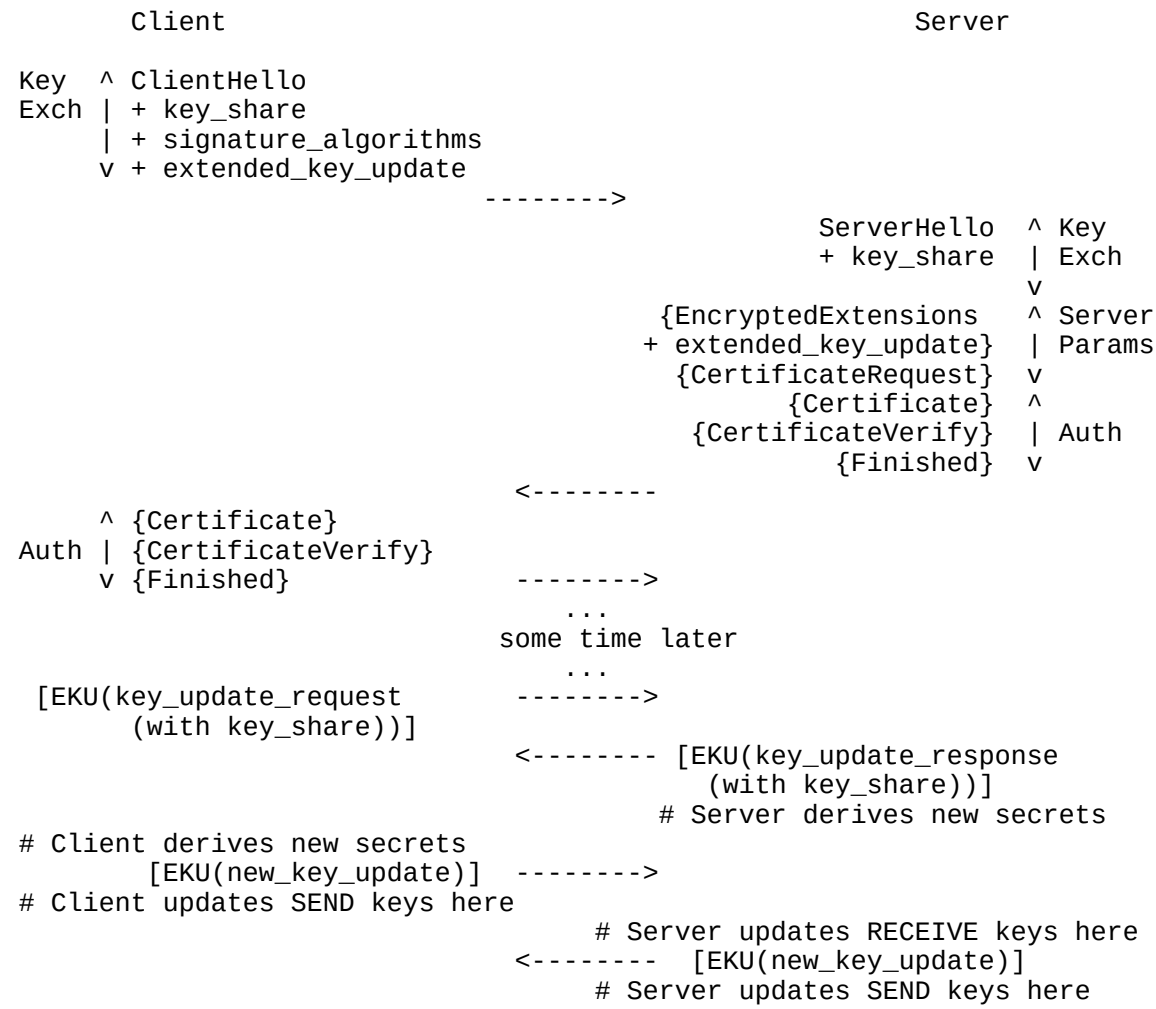


Figure 3: Extended Key Update Example.

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.

Due to the possibility of an ExtendedKeyUpdate(new_key_update) message 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.

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宴洰 ExtendedKeyUpdate(key_update_request), so no separate ACK is required.

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

5. Upon receiving ExtendedKeyUpdate(new_key_update), the responder MUST update its receive keys and epoch value.

6. The responder acknowledges the received message by sending its own ExtendedKeyUpdate(new_key_update).

7. After the initiator receives the responder寰洅 ExtendedKeyUpdate(new_key_update), the initiator MUST update its send key and epoch value. With the receipt of that message, the initiator MUST also update its receive keys.

8. The initiator MUST acknowledge the responder寰洅 ExtendedKeyUpdate(new_key_update) with an ACK message.

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

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

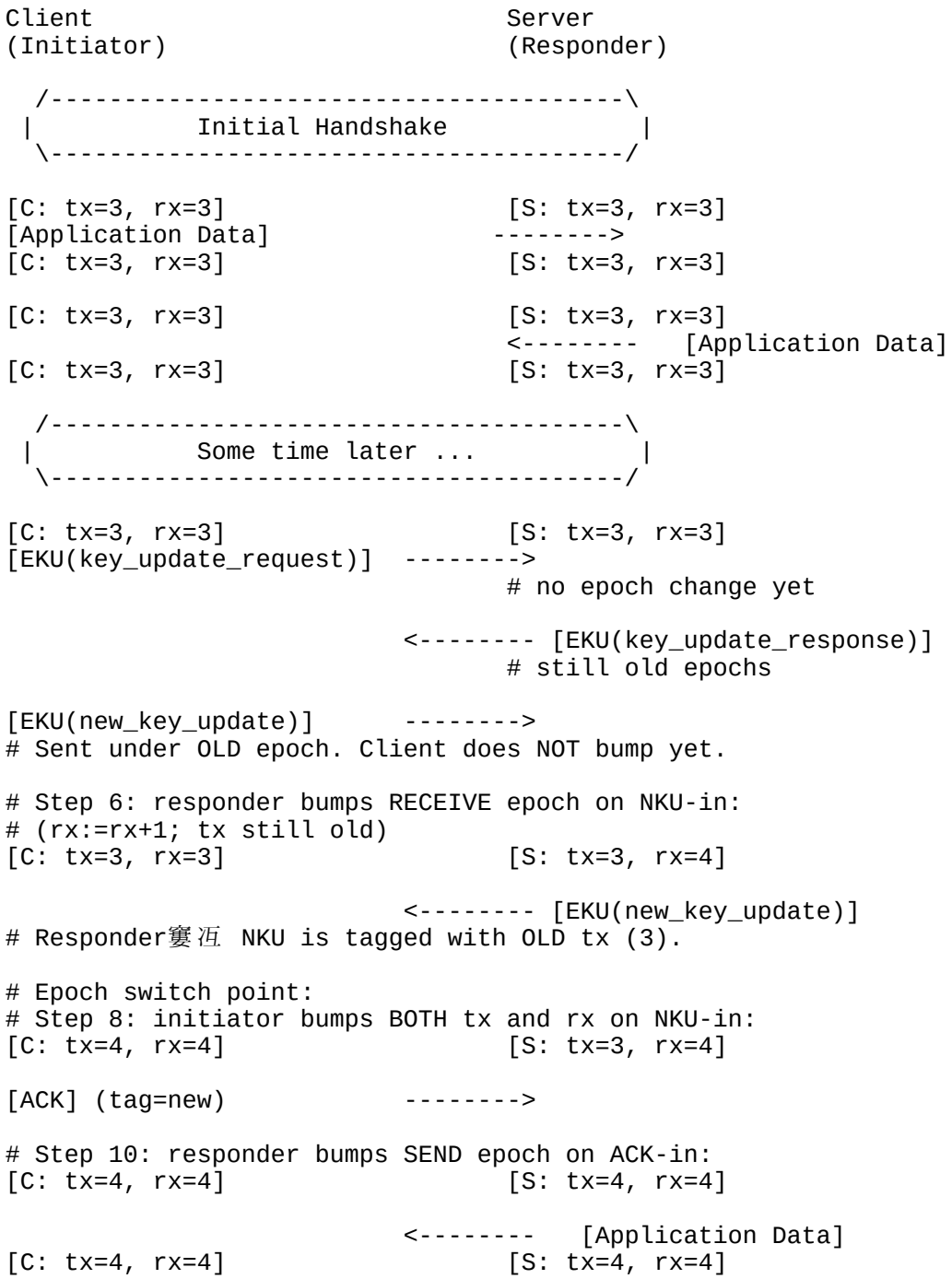
```
enum {
    client_hello(1),
    server_hello(2),
    new_session_ticket(4),
    end_of_early_data(5),
    encrypted_extensions(8),
    request_connection_id(9),
    new_connection_id(10),
    certificate(11),
    certificate_request(13),
    certificate_verify(15),
    finished(20),
    key_update(24),
    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]
[S: tx=3, rx=3]
[Application Data]
[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)]
still old epochs
[EKU(new_key_update)]
Sent under OLD epoch. Client does NOT bump yet.
Step 6: responder bumps RECEIVE epoch on NKU-in:
(rx:=rx+1; tx still old)
[C: tx=3, rx=3]
[S: tx=3, rx=4]
[EKU(new_key_update)]
Responder寰洉 NKU is tagged with OLD tx (3).
Epoch switch point:
Step 8: initiator bumps BOTH tx and rx on NKU-in:
[C: tx=4, rx=4]
[S: tx=3, rx=4]
[ACK] (tag=new)
Step 10: responder 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 -> 3/3	3/3 -> 3/3	
NKU ---------->	3/3 -> 3/3	3/3 -> 3/4	<- step 6
<---------- NKU	3/3 -> 4/4	3/4 -> 3/4	<- step 8
ACK ---------->	4/4 -> 4/4	3/4 -> 4/4	<- step 10
<---------- 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 Secrets

When the extended key update message exchange is completed both peers have successfully updated their application traffic secrets. 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

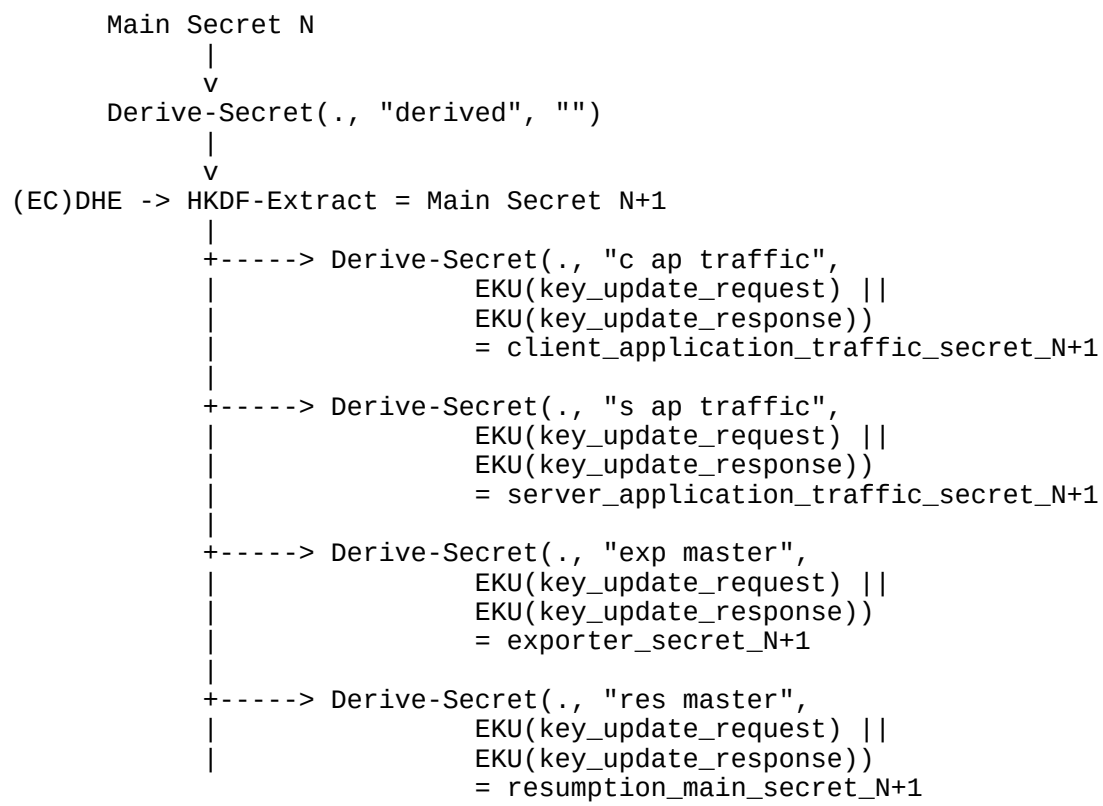
* a secret derived from the (EC)DHE exchange (or from the hybrid key exchange / PQ-KEM exchange),

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

* the concatenation of the ExtendedKeyUpdate(key_update_request) and the ExtendedKeyUpdate(key_update_response) messages, which contain the key shares, binding the encapsulated shared secret ciphertext to IKM in case of hybrid key exchange, providing MAL-BIND-K-CT security (see [CDM23]), and

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

The following diagram shows the 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 master 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 after receiving the ExtendedKeyUpdate(new_key_update) message.

When using this extension, it is important to consider its interaction with ticket-based session resumption. If resumption occurs without a new (EC)DH exchange that provides forward secrecy, an attacker could potentially revert the security context to an earlier state, thereby negating the benefits of the extended key update. To preserve the security guarantees provided by key updates, endpoints MUST either invalidate any session tickets issued prior to the key update or ensure that resumption always involves a fresh (EC)DH exchange.

If session tickets cannot be stored securely, developers SHOULD consider disabling ticket-based resumption in their deployments. While this approach may impact performance, it provides improved security properties.

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.

The specification in [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

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.

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 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].

Delivering the derived EKM in this notification allows applications that depend on exporter-based keying material to install new application-layer keys in synchronization with the epoch transition.

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. 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.13 [DTLS] continue to apply.

11.1. Scope of Key Compromise

Extended Key Update assumes a transient compromise of the current application traffic keys, not a persistent attacker with ongoing access to key material. The procedure itself does not rely on long-term private keys; those are assumed to remain secure, as they are stored in a secure element, such as a Trusted Execution Environment (TEE), Hardware Security Module (HSM), or Trusted Platform Module (TPM). In contrast, application traffic keys are stored within the rich operating system, where short-term exposure due to memory disclosure or transient compromise may occur. Post-compromise security can be re-established, provided the compromise is no longer active when an Extended Key Update is performed.

Extended Key Update can restore confidentiality only if the attacker no longer has access to either peer and cannot interfere with the Extended Key Update procedure. 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. In that case, the attacker can impersonate each endpoint and substitute key shares, maintaining control of the communication. Therefore, Extended Key Update provides recovery only in the case where the compromise has ended before the procedure begins.

If a compromise occurs before the handshake completes, the ephemeral key exchange, client_handshake_traffic_secret, and server_handshake_traffic_secret could be exposed. In that case, only the initial handshake messages and the application data encrypted with these secrets 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.

11.2. 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.

11.3. 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.

11.4. 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.

12. IANA Considerations

12.1. 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]

12.2. TLS HandshakeType

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

* Value: TBD2
* Description: extended_key_update
* DTLS-OK: Y
* Reference: [This document]

12.3. 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 [TLS-Ext-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	new_key_update	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.

12.4. 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

13. References

13.1. 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>.

[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-16, 14 September 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-tlsflags-16>.

13.2. 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>.

[CDM23] ACM, "Keeping Up with the KEMs: Stronger Security Notions for KEMs and automated analysis of KEM-based protocols", November 2023, <https://eprint.iacr.org/2023/1933.pdf>.

[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>.

[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-01, 29 September 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-ecdhe-mlkem-01>.

[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-04, 22 July 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-mlkem-04>.

[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.

For editorial reasons we abbreviate the protocol message types:

* Req - ExtendedKeyUpdate(request)
* Resp - ExtendedKeyUpdate(response)
* NKU - ExtendedKeyUpdate(new_key_update)

* 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.

B.1. TLS 1.3 State Machines

This section describes the initiator and responder state machines.

B.1.1. Initiator State Machine


START
send_key=current,
receive_key=current
updating=0
(1) send Req with local KeyShareEntry
set updating=1
WAIT_RESP
updating=1
(A) recv peer Req (crossed updates):
compare peer.key_exchange to local.key_exchange
- if peer < local: IGNORE peer Req (send nothing)
- if peer == local: ABORT "unexpected_message"
- if peer > local: ABANDON local update:
* set updating=0
* act as RESPONDER: send Resp with KeyShareEntry,
derive new secrets, then proceed as in responder flow
(B) recv Resp with key_share:
derive new secrets
send NKU (encrypted under old keys)
update SEND keys (send_key := new)
--> WAIT_R_NKU
SENT_NKU / WAIT_R_NKU
send_key=new,
receive_key=current,
updating=1
(5) recv NKU (encrypted under old keys)
update RECEIVE keys (receive_key := new)
set updating=0
FINISHED

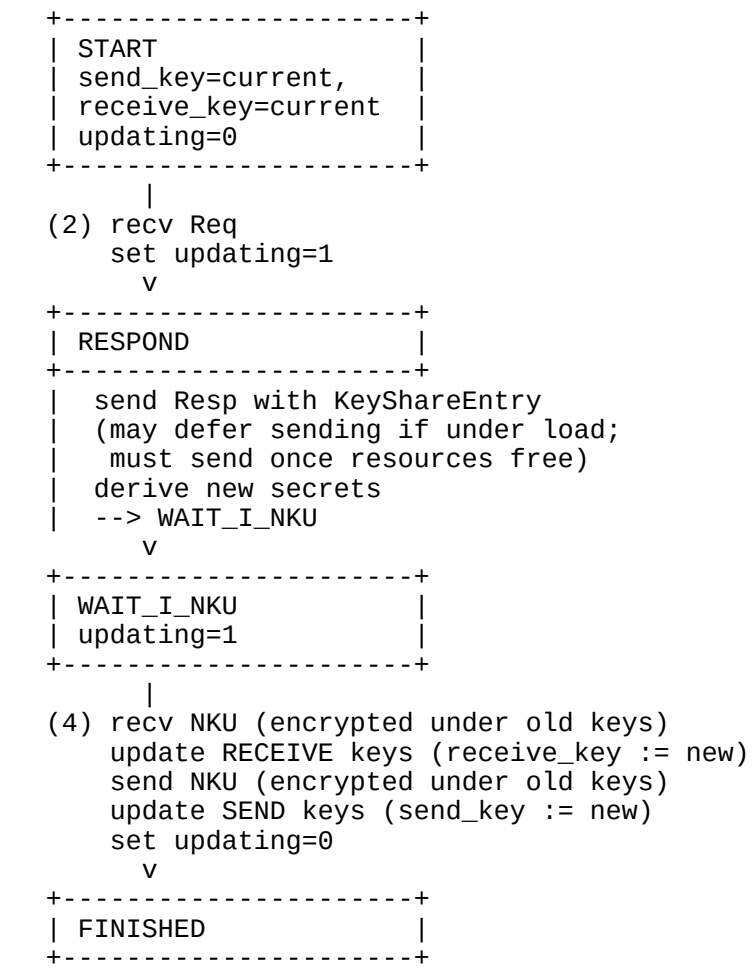
Notes:

* Both NKU messages are sent under old keys. The old-key NKU must be received before accepting traffic under new 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:

* No "accept/reject" or status in Resp; wire format has only a KeyShareEntry.

* 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_NKU / WAIT_R_NKU / 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.

No status in responses. ExtendedKeyUpdate(key_update_response) carries only a KeyShareEntry; there is no accept/reject/status field in the wire format. Implementations either proceed normally or abort on error; there is no benign "reject" reply.

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 NKU still tagged with the old tx, moving to WAIT_R_NKU.

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 NKU (tag equals the current rx, meaning the responder is still tagging with its old tx), the initiator:

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

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

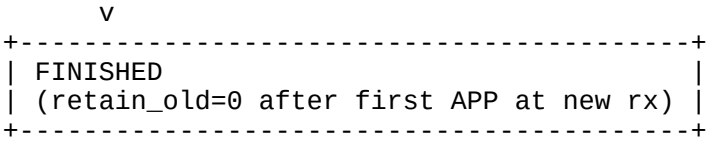
3. sends ACK tagged with the new tx (which now equals the new rx),

4. clears updating and enters FINISHED.

Retention at the initiator 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.


START
rx := E; tx := E, updating := 0
(1) send Req [tag=tx]
set updating := 1
WAIT_RESP
(A) recv peer Req (crossed):
if peer < local -> IGNORE (Self-Loop)
if peer == local -> ABORT (Error)
if peer > local -> ABANDON update;
updating := 0;
act as RESPONDER
APP send (anytime) (Self-Loop)
APP recv: retain_old && e == old_rx (Self-Loop)
APP recv: e == rx (Self-Loop)
recv Resp [e == rx]
derive secrets; send NKU [tag=old tx]
WAIT_R_NKU
APP send/recv allowed
(6) recv NKU [e==rx]
(Responder still tags old tx)
ACTIVATE RETENTION
old_rx=rx;
retain_old=1;
rx=rx+1; tx=tx+1
(8) send ACK [tag=tx]
set updating := 0; assert tx==rx



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 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, key_update_response is tagged with the current tx. After sending the response, the responder enters WAIT_I_NKU.

When a new_key_update (NKU) is received with the correct epoch, the responder 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. The responder then sends its own NKU tagged with the old transmit epoch and moves to WAIT_ACK.

Finally, upon receipt of an ACK matching the updated epoch, the responder 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.

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