TLS
Internet-Draft
Obsoletes: 6347 (if approved)
Intended status: Standards Track
Expires: May 9, 2019

E. Rescorla
RTFM, Inc.
H. Tschofenig
Arm Limited
N. Modadugu
Google, Inc.

November 05, 2018

The Datagram Transport Layer Security (DTLS) Protocol Version 1.3
draft-ietf-tls-dtls13-30

Abstract

This document specifies Version 1.3 of the Datagram Transport Layer Security (DTLS) protocol. DTLS 1.3 allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

The DTLS 1.3 protocol is intentionally based on the Transport Layer Security (TLS) 1.3 protocol and provides equivalent security guarantees. Datagram semantics of the underlying transport are preserved by the DTLS protocol.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH

The source for this draft is maintained in GitHub. Suggested changes should be submitted as pull requests at https://github.com/tlswg/dtls13-spec. Instructions are on that page as well. Editorial changes can be managed in GitHub, but any substantive change should be discussed on the TLS mailing list.

The primary goal of the TLS protocol is to provide privacy and data integrity between two communicating peers. The TLS protocol is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol. However, TLS must run over a reliable transport channel - typically TCP [RFC0793].

There are applications that utilize UDP [RFC0768] as a transport and to offer communication security protection for those applications the

Datagram Transport Layer Security (DTLS) protocol has been designed. DTLS is deliberately designed to be as similar to TLS as possible, both to minimize new security invention and to maximize the amount of code and infrastructure reuse.

DTLS 1.0 [RFC4347] was originally defined as a delta from TLS 1.1 [RFC4346] and DTLS 1.2 [RFC6347] was defined as a series of deltas to TLS 1.2 [RFC5246]. There is no DTLS 1.1; that version number was skipped in order to harmonize version numbers with TLS. This specification describes the most current version of the DTLS protocol aligning with the efforts around TLS 1.3 [TLS13].

Implementations that speak both DTLS 1.2 and DTLS 1.3 can interoperate with those that speak only DTLS 1.2 (using DTLS 1.2 of course), just as TLS 1.3 implementations can interoperate with TLS 1.2 (see Appendix D of [TLS13] for details). While backwards compatibility with DTLS 1.0 is possible the use of DTLS 1.0 is not recommended as explained in Section 3.1.2 of RFC 7525 [RFC7525].

2. Conventions and Terminology

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

The following terms are used:

- client: The endpoint initiating the DTLS connection.

- connection: A transport-layer connection between two endpoints.

- endpoint: Either the client or server of the connection.

- handshake: An initial negotiation between client and server that establishes the parameters of their transactions.

- peer: An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is remote to the primary subject of discussion.

- receiver: An endpoint that is receiving records.

- sender: An endpoint that is transmitting records.

- session: An association between a client and a server resulting from a handshake.

- server: The endpoint which did not initiate the DTLS connection.

The reader is assumed to be familiar with the TLS 1.3 specification since this document is defined as a delta from TLS 1.3. As in TLS 1.3 the HelloRetryRequest has the same format as a ServerHello message but for convenience we use the term HelloRetryRequest throughout this document as if it were a distinct message.

Figures in this document illustrate various combinations of the DTLS protocol exchanges and the symbols have the following meaning:

- '+' 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 [sender]_handshake_traffic_secret.

- '[]' indicates messages protected using keys derived from traffic_secret_N.

3. DTLS Design Rationale and Overview

The basic design philosophy of DTLS is to construct "TLS over datagram transport". Datagram transport does not require nor provide reliable or in-order delivery of data. The DTLS protocol preserves this property for application data. Applications such as media streaming, Internet telephony, and online gaming use datagram transport for communication due to the delay-sensitive nature of transported data. The behavior of such applications is unchanged when the DTLS protocol is used to secure communication, since the DTLS protocol does not compensate for lost or re-ordered data traffic.

TLS cannot be used directly in datagram environments for the following five reasons:

1. TLS does not allow independent decryption of individual records. Because the integrity check indirectly depends on a sequence number, if record N is not received, then the integrity check on record N+1 will be based on the wrong sequence number and thus will fail. DTLS solves this problem by adding explicit sequence numbers.

2. The TLS handshake is a lock-step cryptographic handshake. Messages must be transmitted and received in a defined order; any

other order is an error. This is incompatible with reordering and message loss.

3. Not all TLS 1.3 handshake messages (such as the NewSessionTicket message) are acknowledged. Hence, a new acknowledgment message has to be added to detect message loss.

4. Handshake messages are potentially larger than any given datagram, thus creating the problem of IP fragmentation.

5. Datagram transport protocols, like UDP, are susceptible to abusive behavior effecting denial of service attacks against nonparticipants, and require a return-routability check with the help of cookies to be integrated into the handshake. A detailed discussion of countermeasures can be found in Section 5.1.

3.1. Packet Loss

DTLS uses a simple retransmission timer to handle packet loss. Figure 1 demonstrates the basic concept, using the first phase of the DTLS handshake:

```
         Client                                   Server
         ------                                   ------
         ClientHello           ------>

                                 X<-- HelloRetryRequest
                                                  (lost)

         [Timer Expires]

         ClientHello           ------>
         (retransmit)
```

Figure 1: DTLS retransmission example

Once the client has transmitted the ClientHello message, it expects to see a HelloRetryRequest or a ServerHello from the server. However, if the server's message is lost, the client knows that either the ClientHello or the response from the server has been lost and retransmits. When the server receives the retransmission, it knows to retransmit.

The server also maintains a retransmission timer and retransmits when that timer expires.

Note that timeout and retransmission do not apply to the HelloRetryRequest since this would require creating state on the

server. The HelloRetryRequest is designed to be small enough that it will not itself be fragmented, thus avoiding concerns about interleaving multiple HelloRetryRequests.

3.1.1. Reordering

In DTLS, each handshake message is assigned a specific sequence number within that handshake. When a peer receives a handshake message, it can quickly determine whether that message is the next message it expects. If it is, then it processes it. If not, it queues it for future handling once all previous messages have been received.

3.1.2. Message Size

TLS and DTLS handshake messages can be quite large (in theory up to 2^24-1 bytes, in practice many kilobytes). By contrast, UDP datagrams are often limited to less than 1500 bytes if IP fragmentation is not desired. In order to compensate for this limitation, each DTLS handshake message may be fragmented over several DTLS records, each of which is intended to fit in a single IP datagram. Each DTLS handshake message contains both a fragment offset and a fragment length. Thus, a recipient in possession of all bytes of a handshake message can reassemble the original unfragmented message.

3.2. Replay Detection

DTLS optionally supports record replay detection. The technique used is the same as in IPsec AH/ESP, by maintaining a bitmap window of received records. Records that are too old to fit in the window and records that have previously been received are silently discarded. The replay detection feature is optional, since packet duplication is not always malicious, but can also occur due to routing errors. Applications may conceivably detect duplicate packets and accordingly modify their data transmission strategy.

4. The DTLS Record Layer

The DTLS record layer is different from the TLS 1.3 record layer.

1. The DTLSCiphertext structure omits the superfluous version number and type fields.

2. DTLS adds an explicit epoch and sequence number to the TLS record header. This sequence number allows the recipient to correctly verify the DTLS MAC. However, the number of bits used for the

epoch and sequence number fields in the DTLSCiphertext structure have been reduced.

3. The DTLSCiphertext structure has a variable length header.

Note that the DTLS 1.3 record layer is different from the DTLS 1.2 record layer.

DTLSPlaintext records are used to send unprotected records and DTLSCiphertext records are used to send protected records.

The DTLS record formats are shown below. Unless explicitly stated the meaning of the fields is unchanged from previous TLS / DTLS versions.

```
struct {
    ContentType type;
    ProtocolVersion legacy_record_version;
    uint16 epoch = 0                                 // DTLS field
    uint48 sequence_number;                          // DTLS field
    uint16 length;
    opaque fragment[DTLSPlaintext.length];
} DTLSPlaintext;

struct {
     opaque content[DTLSPlaintext.length];
     ContentType type;
     uint8 zeros[length_of_padding];
} DTLSInnerPlaintext;

struct {
    opaque unified_hdr[variable];
    opaque encrypted_record[length];
} DTLSCiphertext;
```

Figure 2: DTLS 1.3 Record Format

unified_hdr: The unified_hdr is a field of variable length, as shown in Figure 3.

encrypted_record: Identical to the encrypted_record field in a TLS 1.3 record.

The DTLSCiphertext header is tightly bit-packed, as shown below:

```
0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|0|0|1|C|S|L|E E|
+-+-+-+-+-+-+-+-+
| Connection ID |   Legend:
| (if any,      |
/  length as    /   C   - CID present
|  negotiated)  |   S   - Sequence number length
+-+-+-+-+-+-+-+-+   L   - Length present
|  8 or 16 bit  |   E   - Epoch
|Sequence Number|
+-+-+-+-+-+-+-+-+
| 16 bit Length |
| (if present)  |
+-+-+-+-+-+-+-+-+
```

Figure 3: DTLS 1.3 CipherText Header

C: The C bit is set if the connection ID is present.

S: The size of the sequence number. 0 means an 8-bit sequence number, 1 means 16-bit.

L: The L bit is set if the length is present.

E: The low order two bits of the epoch.

connection ID: Variable length connection ID. The connection ID concept is described in [DTLS-CID]. An example can be found in Section 9.1.

sequence number: The low order 8 or 16 bits of the record sequence number. This value is 16 bits if the S bit is set to 1, and 8 bits if the S bit is 0.

length: Identical to the length field in a TLS 1.3 record.

As with previous versions of DTLS, multiple DTLSPlaintext and DTLSCiphertext records can be included in the same underlying transport datagram.

Figure 4 illustrates different record layer header types.

```
    0 1 2 3 4 5 6 7       0 1 2 3 4 5 6 7        0 1 2 3 4 5 6 7
   +-+-+-+-+-+-+-+-+     +-+-+-+-+-+-+-+-+     +-+-+-+-+-+-+-+-+
   | Content Type  |     |0|0|1|C|1|E|E|1|     |0|0|1|0|0|E|E|0|
   +-+-+-+-+-+-+-+-+     +-+-+-+-++-+-+-+     +-+-+-+-+-+-+-+-+
   |   16 bit      |     |    16 bit     |     |8-bit Seq. No. |
   |   Version     |     |Sequence Number|     +-+-+-+-+-+-+-+-+
   +-+-+-+-+-+-+-+-+     +-+-+-+-+-+-+-+-+     |               |
   |   16 bit      |     |               |     |   Encrypted   |
   |    Epoch      |     / Connection ID /     /   Record      /
   +-+-+-+-+-+-+-+-+     |               |     |               |
   |               |     +-+-+-+-+-+-+-+-+     +-+-+-+-+-+-+-+-+
   |               |     |   16 bit      |
   |   48 bit      |     |   Length      |       DTLSCiphertext
   |Sequence Number|     +-+-+-+-+-+-+-+-+         Structure
   |               |     |               |         (minimal)
   |               |     |  Encrypted    |
   +-+-+-+-+-+-+-+-+     /  Record       /
   |    16 bit     |     |               |
   |    Length     |     +-+-+-+-+-+-+-+-+
   +-+-+-+-+-+-+-+-+
   |               |      DTLSCiphertext
   |               |        Structure
   /   Fragment    /          (full)
   |               |
   +-+-+-+-+-+-+-+-+

    DTLSPlaintext
      Structure
```

Figure 4: Header Examples

The length field may be omitted and therefore the record consumes the entire rest of the datagram in the lower level transport. In this case it is not possible to have multiple DTLSCiphertext format records without length fields in the same datagram.

Omitting the length field MUST only be used for data which is protected with one of the application_traffic_secret values, and not for messages protected with either [sender]_handshake_traffic_sercret or [sender]_early_traffic_secret values. When using an [sender]_application_traffic_secret for message protection, Implementations MAY include the length field at their discretion.

The entire header value shown above is used as it appears on the wire as the additional data value for the AEAD function.

4.1. Determining the Header Format

Implementations can distinguish the two header formats by examining the first byte:

- If the first byte is alert(21), handshake(22), or ack(proposed, 25), the record MUST be interpreted as a DTLSPlaintext record.

- If the first byte is any other other value, then receivers MUST check to see if the leading bits of the first byte are 001. If so, the implementation MUST process the record as DTLSCiphertext; the true content type will be inside the protected portion.

- Otherwise, the record MUST be rejected as if it had failed deprotection, as described in Section 4.5.2.

4.2. Sequence Number and Epoch

4.2.1. Processing Guidelines

DTLS uses an explicit sequence number, rather than an implicit one, carried in the sequence_number field of the record. Sequence numbers are maintained separately for each epoch, with each sequence_number initially being 0 for each epoch.

The epoch number is initially zero and is incremented each time keying material changes and a sender aims to rekey. More details are provided in Section 6.1.

Because DTLS records may be reordered, a record from epoch 1 may be received after epoch 2 has begun. In general, implementations SHOULD discard packets from earlier epochs, but if packet loss causes noticeable problems implementations MAY choose to retain keying material from previous epochs for up to the default MSL specified for TCP [RFC0793] to allow for packet reordering. (Note that the intention here is that implementers use the current guidance from the IETF for MSL, as specified in [RFC0793] or successors not that they attempt to interrogate the MSL that the system TCP stack is using.) Until the handshake has completed, implementations MUST accept packets from the old epoch.

Conversely, it is possible for records that are protected with the new epoch to be received prior to the completion of a handshake. For instance, the server may send its Finished message and then start transmitting data. Implementations MAY either buffer or discard such packets, though when DTLS is used over reliable transports (e.g., SCTP [RFC4960]), they SHOULD be buffered and processed once the handshake completes. Note that TLS's restrictions on when packets

may be sent still apply, and the receiver treats the packets as if they were sent in the right order. In particular, it is still impermissible to send data prior to completion of the first handshake.

Note that some care needs to be taken during the handshake to ensure that retransmitted messages use the right epoch and keying material.

Implementations MUST either abandon an association or re-key prior to allowing the sequence number to wrap.

Implementations MUST NOT allow the epoch to wrap, but instead MUST establish a new association, terminating the old association.

4.2.2. Reconstructing the Sequence Number and Epoch

When receiving protected DTLS records message, the recipient does not have a full epoch or sequence number value and so there is some opportunity for ambiguity. Because the full epoch and sequence number are used to compute the per-record nonce, failure to reconstruct these values leads to failure to deprotect the record, and so implementations MAY use a mechanism of their choice to determine the full values. This section provides an algorithm which is comparatively simple and which implementations are RECOMMENDED to follow.

If the epoch bits match those of the current epoch, then implementations SHOULD reconstruct the sequence number by computing the full sequence number which is numerically closest to one plus the sequence number of the highest successfully deprotected record.

During the handshake phase, the epoch bits unambiguously indicate the correct key to use. After the handshake is complete, if the epoch bits do not match those from the current epoch implementations SHOULD use the most recent past epoch which has matching bits, and then reconstruct the sequence number as described above.

4.2.3. Sequence Number Encryption

In DTLS 1.3, when records are encrypted, record sequence numbers are also encrypted. The basic pattern is that the underlying encryption algorithm used with the AEAD algorithm is used to generate a mask which is then XORed with the sequence number.

When the AEAD is based on AES, then the Mask is generated by computing AES-ECB on the first 16 bytes of the ciphertext:

```
Mask = AES-ECB(sn_key, Ciphertext[0..15])
```

When the AEAD is based on ChaCha20, then the mask is generated by treating the first 12 bytes of the ciphertext as the Nonce and the next 4 bytes as the counter:

```
Mask = ChaCha20(sn_key, Ciphertext[0..12], Ciphertext[13..15])
```

The sn_key is computed as follows:

```
[sender]_sn_key  = HKDF-Expand-Label(Secret, "sn" , "", key_length)
```

[sender] denotes the sending side. The Secret value to be used is described in Section 7.3 of [TLS13].

The encrypted sequence number is computed by XORing the leading bytes of the Mask with the sequence number. Decryption is accomplished by the same process.

In some (rare) cases the ciphertext may be less than 16 bytes. This cannot happen with most of the DTLS AEAD algorithms because the authentication tag itself is 16 bytes, however some algorithms such as TLS_AES_128_CCM_8_SHA256 have a shorter authentication tag, and in combination with a short plaintext, the result might be less than 16 bytes. In this case, implementations MUST pad the plaintext out (using the conventional record padding mechanism) in order to make a suitable-length ciphertext.

Note that sequence number encryption is only applied to the DTLSCiphertext structure and not to the DTLSPlaintext structure, which also contains a sequence number.

4.3. Transport Layer Mapping

DTLS messages MAY be fragmentented into multiple DTLS records. Each DTLS record MUST fit within a single datagram. In order to avoid IP fragmentation, clients of the DTLS record layer SHOULD attempt to size records so that they fit within any PMTU estimates obtained from the record layer.

Multiple DTLS records MAY be placed in a single datagram. They are simply encoded consecutively. The DTLS record framing is sufficient to determine the boundaries. Note, however, that the first byte of the datagram payload MUST be the beginning of a record. Records MUST NOT span datagrams.

DTLS records, as defined in this document, do not contain any association identifiers and applications must arrange to multiplex between associations. With UDP, the host/port number is used to look up the appropriate security association for incoming records.

However, the Connection ID extension defined in [DTLS-CID] adds an association identifier to DTLS records.

Some transports, such as DCCP [RFC4340], provide their own sequence numbers. When carried over those transports, both the DTLS and the transport sequence numbers will be present. Although this introduces a small amount of inefficiency, the transport layer and DTLS sequence numbers serve different purposes; therefore, for conceptual simplicity, it is superior to use both sequence numbers.

Some transports provide congestion control for traffic carried over them. If the congestion window is sufficiently narrow, DTLS handshake retransmissions may be held rather than transmitted immediately, potentially leading to timeouts and spurious retransmission. When DTLS is used over such transports, care should be taken not to overrun the likely congestion window. [RFC5238] defines a mapping of DTLS to DCCP that takes these issues into account.

4.4. PMTU Issues

In general, DTLS's philosophy is to leave PMTU discovery to the application. However, DTLS cannot completely ignore PMTU for three reasons:

- The DTLS record framing expands the datagram size, thus lowering the effective PMTU from the application's perspective.

- In some implementations, the application may not directly talk to the network, in which case the DTLS stack may absorb ICMP [RFC1191] "Datagram Too Big" indications or ICMPv6 [RFC4443] "Packet Too Big" indications.

- The DTLS handshake messages can exceed the PMTU.

In order to deal with the first two issues, the DTLS record layer SHOULD behave as described below.

If PMTU estimates are available from the underlying transport protocol, they should be made available to upper layer protocols. In particular:

- For DTLS over UDP, the upper layer protocol SHOULD be allowed to obtain the PMTU estimate maintained in the IP layer.

- For DTLS over DCCP, the upper layer protocol SHOULD be allowed to obtain the current estimate of the PMTU.

- For DTLS over TCP or SCTP, which automatically fragment and reassemble datagrams, there is no PMTU limitation. However, the upper layer protocol MUST NOT write any record that exceeds the maximum record size of 2^14 bytes.

The DTLS record layer SHOULD allow the upper layer protocol to discover the amount of record expansion expected by the DTLS processing.

If there is a transport protocol indication (either via ICMP or via a refusal to send the datagram as in Section 14 of [RFC4340]), then the DTLS record layer MUST inform the upper layer protocol of the error.

The DTLS record layer SHOULD NOT interfere with upper layer protocols performing PMTU discovery, whether via [RFC1191] or [RFC4821] mechanisms. In particular:

- Where allowed by the underlying transport protocol, the upper layer protocol SHOULD be allowed to set the state of the DF bit (in IPv4) or prohibit local fragmentation (in IPv6).

- If the underlying transport protocol allows the application to request PMTU probing (e.g., DCCP), the DTLS record layer SHOULD honor this request.

The final issue is the DTLS handshake protocol. From the perspective of the DTLS record layer, this is merely another upper layer protocol. However, DTLS handshakes occur infrequently and involve only a few round trips; therefore, the handshake protocol PMTU handling places a premium on rapid completion over accurate PMTU discovery. In order to allow connections under these circumstances, DTLS implementations SHOULD follow the following rules:

- If the DTLS record layer informs the DTLS handshake layer that a message is too big, it SHOULD immediately attempt to fragment it, using any existing information about the PMTU.

- If repeated retransmissions do not result in a response, and the PMTU is unknown, subsequent retransmissions SHOULD back off to a smaller record size, fragmenting the handshake message as appropriate. This standard does not specify an exact number of retransmits to attempt before backing off, but 2-3 seems appropriate.

4.5. Record Payload Protection

Like TLS, DTLS transmits data as a series of protected records. The rest of this section describes the details of that format.

4.5.1. Anti-Replay

Each DTLS record contains a sequence number to provide replay protection. Sequence number verification SHOULD be performed using the following sliding window procedure, borrowed from Section 3.4.3 of [RFC4303].

The received packet counter for a session MUST be initialized to zero when that session is established. For each received record, the receiver MUST verify that the record contains a sequence number that does not duplicate the sequence number of any other record received during the lifetime of the session. This check SHOULD happen after deprotecting the packet; otherwise the packet discard might itself serve as a timing channel for the sequence number.

Duplicates are rejected through the use of a sliding receive window. (How the window is implemented is a local matter, but the following text describes the functionality that the implementation must exhibit.) A minimum window size of 32 MUST be supported, but a window size of 64 is preferred and SHOULD be employed as the default. Another window size (larger than the minimum) MAY be chosen by the receiver. (The receiver does not notify the sender of the window size.)

The "right" edge of the window represents the highest validated sequence number value received on the session. Records that contain sequence numbers lower than the "left" edge of the window are rejected. Packets falling within the window are checked against a list of received packets within the window. An efficient means for performing this check, based on the use of a bit mask, is described in Section 3.4.3 of [RFC4303]. If the received record falls within the window and is new, or if the packet is to the right of the window, then the packet is new.

The window MUST NOT be updated until the packet has been deprotected successfully.

4.5.2. Handling Invalid Records

Unlike TLS, DTLS is resilient in the face of invalid records (e.g., invalid formatting, length, MAC, etc.). In general, invalid records SHOULD be silently discarded, thus preserving the association; however, an error MAY be logged for diagnostic purposes.

Implementations which choose to generate an alert instead, MUST generate error alerts to avoid attacks where the attacker repeatedly probes the implementation to see how it responds to various types of error. Note that if DTLS is run over UDP, then any implementation which does this will be extremely susceptible to denial-of-service (DoS) attacks because UDP forgery is so easy. Thus, this practice is NOT RECOMMENDED for such transports, both to increase the reliability of DTLS service and to avoid the risk of spoofing attacks sending traffic to unrelated third parties.

If DTLS is being carried over a transport that is resistant to forgery (e.g., SCTP with SCTP-AUTH), then it is safer to send alerts because an attacker will have difficulty forging a datagram that will not be rejected by the transport layer.

5. The DTLS Handshake Protocol

DTLS 1.3 re-uses the TLS 1.3 handshake messages and flows, with the following changes:

1. To handle message loss, reordering, and fragmentation modifications to the handshake header are necessary.

2. Retransmission timers are introduced to handle message loss.

3. A new ACK content type has been added for reliable message delivery of handshake messages.

Note that TLS 1.3 already supports a cookie extension, which is used to prevent denial-of-service attacks. This DoS prevention mechanism is described in more detail below since UDP-based protocols are more vulnerable to amplification attacks than a connection-oriented transport like TCP that performs return-routability checks as part of the connection establishment.

DTLS implementations do not use the TLS 1.3 "compatibility mode" described in [TLS13], Section D.4. DTLS servers MUST NOT echo the "session_id" value from the client and endpoints MUST NOT send ChangeCipherSpec messages. Note however that implementations MUST ignore ChangeCipherSpec messages received in unprotected records.

With these exceptions, the DTLS message formats, flows, and logic are the same as those of TLS 1.3.

5.1. Denial-of-Service Countermeasures

Datagram security protocols are extremely susceptible to a variety of DoS attacks. Two attacks are of particular concern:

1. An attacker can consume excessive resources on the server by transmitting a series of handshake initiation requests, causing the server to allocate state and potentially to perform expensive cryptographic operations.

2. An attacker can use the server as an amplifier by sending connection initiation messages with a forged source of the victim. The server then sends its response to the victim machine, thus flooding it. Depending on the selected ciphersuite this response message can be quite large, as it is the case for a Certificate message.

In order to counter both of these attacks, DTLS borrows the stateless cookie technique used by Photuris [RFC2522] and IKE [RFC7296]. When the client sends its ClientHello message to the server, the server MAY respond with a HelloRetryRequest message. The HelloRetryRequest message, as well as the cookie extension, is defined in TLS 1.3. The HelloRetryRequest message contains a stateless cookie generated using the technique of [RFC2522]. The client MUST retransmit the ClientHello with the cookie added as an extension. The server then verifies the cookie and proceeds with the handshake only if it is valid. This mechanism forces the attacker/client to be able to receive the cookie, which makes DoS attacks with spoofed IP addresses difficult. This mechanism does not provide any defense against DoS attacks mounted from valid IP addresses.

The DTLS 1.3 specification changes the way how cookies are exchanged compared to DTLS 1.2. DTLS 1.3 re-uses the HelloRetryRequest message and conveys the cookie to the client via an extension. The client receiving the cookie uses the same extension to place the cookie subsequently into a ClientHello message. DTLS 1.2 on the other hand used a separate message, namely the HelloVerifyRequest, to pass a cookie to the client and did not utilize the extension mechanism. For backwards compatibility reason the cookie field in the ClientHello is present in DTLS 1.3 but is ignored by a DTLS 1.3 compliant server implementation.

The exchange is shown in Figure 5. Note that the figure focuses on the cookie exchange; all other extensions are omitted.

```
      Client                                  Server
      ------                                  ------
      ClientHello           ------>

                            <----- HelloRetryRequest
                                    + cookie

      ClientHello           ------>
       + cookie

      [Rest of handshake]
```

Figure 5: DTLS exchange with HelloRetryRequest containing the "cookie" extension

The cookie extension is defined in Section 4.2.2 of [TLS13]. When sending the initial ClientHello, the client does not have a cookie yet. In this case, the cookie extension is omitted and the legacy_cookie field in the ClientHello message SHOULD be set to a zero length vector (i.e., a single zero byte length field) and MUST be ignored by a server negotiating DTLS 1.3.

When responding to a HelloRetryRequest, the client MUST create a new ClientHello message following the description in Section 4.1.2 of [TLS13].

If the HelloRetryRequest message is used, the initial ClientHello and the HelloRetryRequest are included in the calculation of the transcript hash. The computation of the message hash for the HelloRetryRequest is done according to the description in Section 4.4.1 of [TLS13].

The handshake transcript is not reset with the second ClientHello and a stateless server-cookie implementation requires the transcript of the HelloRetryRequest to be stored in the cookie or the internal state of the hash algorithm, since only the hash of the transcript is required for the handshake to complete.

When the second ClientHello is received, the server can verify that the cookie is valid and that the client can receive packets at the given IP address. If the client's apparent IP address is embedded in the cookie, this prevents an attacker from generating an acceptable ClientHello apparently from another user.

One potential attack on this scheme is for the attacker to collect a number of cookies from different addresses where it controls endpoints and then reuse them to attack the server. The server can defend against this attack by changing the secret value frequently,

thus invalidating those cookies. If the server wishes to allow legitimate clients to handshake through the transition (e.g., a client received a cookie with Secret 1 and then sent the second ClientHello after the server has changed to Secret 2), the server can have a limited window during which it accepts both secrets. [RFC7296] suggests adding a key identifier to cookies to detect this case. An alternative approach is simply to try verifying with both secrets. It is RECOMMENDED that servers implement a key rotation scheme that allows the server to manage keys with overlapping lifetime.

Alternatively, the server can store timestamps in the cookie and reject cookies that were generated outside a certain interval of time.

DTLS servers SHOULD perform a cookie exchange whenever a new handshake is being performed. If the server is being operated in an environment where amplification is not a problem, the server MAY be configured not to perform a cookie exchange. The default SHOULD be that the exchange is performed, however. In addition, the server MAY choose not to do a cookie exchange when a session is resumed. Clients MUST be prepared to do a cookie exchange with every handshake.

If a server receives a ClientHello with an invalid cookie, it MUST NOT respond with a HelloRetryRequest. Restarting the handshake from scratch, without a cookie, allows the client to recover from a situation where it obtained a cookie that cannot be verified by the server. As described in Section 4.1.4 of [TLS13], clients SHOULD also abort the handshake with an "unexpected_message" alert in response to any second HelloRetryRequest which was sent in the same connection (i.e., where the ClientHello was itself in response to a HelloRetryRequest).

5.2. DTLS Handshake Message Format

In order to support message loss, reordering, and message fragmentation, DTLS modifies the TLS 1.3 handshake header:

```
enum {
    client_hello(1),
    server_hello(2),
    new_session_ticket(4),
    end_of_early_data(5),
    encrypted_extensions(8),
    certificate(11),
    certificate_request(13),
    certificate_verify(15),
    finished(20),
    key_update(24),
    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 (HandshakeType) {
        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;
    } body;
} Handshake;
```

The first message each side transmits in each association always has message_seq = 0. Whenever a new message is generated, the message_seq value is incremented by one. When a message is retransmitted, the old message_seq value is re-used, i.e., not incremented. From the perspective of the DTLS record layer, the retransmission is a new record. This record will have a new DTLSPlaintext.sequence_number value.

DTLS implementations maintain (at least notionally) a next_receive_seq counter. This counter is initially set to zero. When a handshake message is received, if its message_seq value matches next_receive_seq, next_receive_seq is incremented and the message is processed. If the sequence number is less than

next_receive_seq, the message MUST be discarded. If the sequence number is greater than next_receive_seq, the implementation SHOULD queue the message but MAY discard it. (This is a simple space/ bandwidth tradeoff).

In addition to the handshake messages that are deprecated by the TLS 1.3 specification DTLS 1.3 furthermore deprecates the HelloVerifyRequest message originally defined in DTLS 1.0. DTLS 1.3-compliant implements MUST NOT use the HelloVerifyRequest to execute a return-routability check. A dual-stack DTLS 1.2/DTLS 1.3 client MUST, however, be prepared to interact with a DTLS 1.2 server.

5.3. ClientHello Message

The format of the ClientHello used by a DTLS 1.3 client differs from the TLS 1.3 ClientHello format as shown below.

```
uint16 ProtocolVersion;
opaque Random[32];

uint8 CipherSuite[2];    /* Cryptographic suite selector */

struct {
    ProtocolVersion legacy_version = { 254,253 }; // DTLSv1.2
    Random random;
    opaque legacy_session_id<0..32>;
    opaque legacy_cookie<0..2^8-1>;                  // DTLS
    CipherSuite cipher_suites<2..2^16-2>;
    opaque legacy_compression_methods<1..2^8-1>;
    Extension extensions<8..2^16-1>;
} ClientHello;
```

legacy_version: In previous versions of DTLS, this field was used for version negotiation and represented the highest version number supported by the client. Experience has shown that many servers do not properly implement version negotiation, leading to "version intolerance" in which the server rejects an otherwise acceptable ClientHello with a version number higher than it supports. In DTLS 1.3, the client indicates its version preferences in the "supported_versions" extension (see Section 4.2.1 of [TLS13]) and the legacy_version field MUST be set to {254, 253}, which was the version number for DTLS 1.2.

random: Same as for TLS 1.3.

legacy_session_id: Same as for TLS 1.3.

legacy_cookie: A DTLS 1.3-only client MUST set the legacy_cookie field to zero length.

cipher_suites: Same as for TLS 1.3.

legacy_compression_methods: Same as for TLS 1.3.

extensions: Same as for TLS 1.3.

5.4. Handshake Message Fragmentation and Reassembly

Each DTLS message MUST fit within a single transport layer datagram. However, handshake messages are potentially bigger than the maximum record size. Therefore, DTLS provides a mechanism for fragmenting a handshake message over a number of records, each of which can be transmitted separately, thus avoiding IP fragmentation.

When transmitting the handshake message, the sender divides the message into a series of N contiguous data ranges. These ranges MUST NOT be larger than the maximum handshake fragment size and MUST jointly contain the entire handshake message. The ranges MUST NOT overlap. The sender then creates N handshake messages, all with the same message_seq value as the original handshake message. Each new message is labeled with the fragment_offset (the number of bytes contained in previous fragments) and the fragment_length (the length of this fragment). The length field in all messages is the same as the length field of the original message. An unfragmented message is a degenerate case with fragment_offset=0 and fragment_length=length.

When a DTLS implementation receives a handshake message fragment, it MUST buffer it until it has the entire handshake message. DTLS implementations MUST be able to handle overlapping fragment ranges. This allows senders to retransmit handshake messages with smaller fragment sizes if the PMTU estimate changes.

Note that as with TLS, multiple handshake messages may be placed in the same DTLS record, provided that there is room and that they are part of the same flight. Thus, there are two acceptable ways to pack two DTLS messages into the same datagram: in the same record or in separate records.

5.5. End Of Early Data

The DTLS 1.3 handshake has one important difference from the TLS 1.3 handshake: the EndOfEarlyData message is omitted both from the wire and the handshake transcript: because DTLS records have epochs, EndOfEarlyData is not necessary to determine when the early data is complete, and because DTLS is lossy, attackers can trivially mount

the deletion attacks that EndOfEarlyData prevents in TLS. Servers SHOULD aggressively age out the epoch 1 keys upon receiving the first epoch 1 record and SHOULD NOT accept epoch 1 data after the first epoch 3 record is received.

5.6. DTLS Handshake Flights

DTLS messages are grouped into a series of message flights, according to the diagrams below.

```
Client                                             Server

ClientHello                                      +----------+
 + key_share*                                    | Flight 1 |
 + pre_shared_key*      -------->                +----------+

                                                 +----------+
                        <--------        HelloRetryRequest  | Flight 2 |
                                          + cookie       +----------+


ClientHello                                      +----------+
 + key_share*                                    | Flight 3 |
 + pre_shared_key*      -------->                +----------+
 + cookie

                                               ServerHello
                                              + key_share*
                                         + pre_shared_key*  +----------+
                                     {EncryptedExtensions}  | Flight 4 |
                                     {CertificateRequest*}  +----------+
                                            {Certificate*}
                                      {CertificateVerify*}
                        <--------                {Finished}
                                       [Application Data*]


 {Certificate*}                                  +----------+
 {CertificateVerify*}                            | Flight 5 |
 {Finished}             -------->                +----------+
 [Application Data]

                                                 +----------+
                        <--------                    [ACK]  | Flight 6 |
                                       [Application Data*]  +----------+

 [Application Data]     <------->      [Application Data]
```

Figure 6: Message flights for a full DTLS Handshake (with cookie exchange)

```
   ClientHello                                          +----------+
    + pre_shared_key                                    | Flight 1 |
    + key_share*         -------->                      +----------+


                                             ServerHello
                                        + pre_shared_key  +----------+
                                            + key_share*  | Flight 2 |
                                   {EncryptedExtensions}  +----------+
                         <--------              {Finished}
                                     [Application Data*]
                                                          +----------+
   {Finished}            -------->                        | Flight 3 |
   [Application Data*]                                    +----------+

                                                          +----------+
                         <--------                   [ACK]  | Flight 4 |
                                     [Application Data*]  +----------+

   [Application Data]    <------->      [Application Data]
```

Figure 7: Message flights for resumption and PSK handshake (without cookie exchange)

```
 Client                                            Server

  ClientHello
   + early_data
   + psk_key_exchange_modes                                +----------+
   + key_share*                                            | Flight 1 |
   + pre_shared_key                                        +----------+
  (Application Data*)     -------->

                                                      ServerHello
                                                 + pre_shared_key
                                                     + key_share*  +----------+
                                            {EncryptedExtensions}  | Flight 2 |
                                                       {Finished}  +----------+
                        <--------       [Application Data*]


                                                                   +----------+
  (EndOfEarlyData)                                                 | Flight 3 |
  {Finished}            -------->                                  +----------+
  [Application Data*]
                                                                   +----------+
                        <--------                           [ACK]  | Flight 4 |
                                              [Application Data*]  +----------+

  [Application Data]    <------->      [Application Data]
```

Figure 8: Message flights for the Zero-RTT handshake

```
 Client                                            Server

                                                                   +----------+
                        <--------        [NewSessionTicket] | Flight 1 |
                                                                   +----------+

                                                                   +----------+
 [ACK]                  -------->                                  | Flight 2 |
                                                                   +----------+
```

Figure 9: Message flights for the new session ticket message

Note: The application data sent by the client is not included in the timeout and retransmission calculation.

5.7. Timeout and Retransmission

5.7.1. State Machine

DTLS uses a simple timeout and retransmission scheme with the state machine shown in Figure 10. Because DTLS clients send the first message (ClientHello), they start in the PREPARING state. DTLS servers start in the WAITING state, but with empty buffers and no retransmit timer.

```
                                +-----------+
                                | PREPARING |
                   +----------> |           |
                   |            |           |
                   |            +-----------+
                   |                  |
                   |                  | Buffer next flight
                   |                  |
                   |                 \|/
                   |            +-----------+
                   |            |           |
                   |            |  SENDING  |<------------------+
                   |            |           |                   |
                   |            +-----------+                   |
           Receive |                  |                         |
              next |                  | Send flight or partial  |
            flight |                  | flight                  |
                   |  +---------------+                         |
                   |  |               | Set retransmit timer    |
                   |  |              \|/                        |
                   |  |         +-----------+                   |
                   |  |         |           |                   |
                   +--)---------|  WAITING  |-------------------+
                   |  |  +----->|           |   Timer expires   |
                   |  |  |      +-----------+                   |
                   |  |  |          |  |   |                    |
                   |  |  |          |  |   |                    |
                   |  |  +----------+  |   +--------------------+
                   |  | Receive record |   Read retransmit or ACK
           Receive |  |  Send ACK      |
              last |  |                |
            flight |  |                | Receive ACK
                   |  |                | for last flight
                  \|/\|/               |
                                       |
               +-----------+           |
               |           | <---------+
               | FINISHED  |
```

```
               |           |
               +-----------+
                   |  /|\
                   |   |
                   |   |
                   +---+

             Server read retransmit
                 Retransmit ACK
```

Figure 10: DTLS timeout and retransmission state machine

The state machine has four basic states: PREPARING, SENDING, WAITING, and FINISHED.

In the PREPARING state, the implementation does whatever computations are necessary to prepare the next flight of messages. It then buffers them up for transmission (emptying the buffer first) and enters the SENDING state.

In the SENDING state, the implementation transmits the buffered flight of messages. If the implementation has received one or more ACKs (see Section 7) from the peer, then it SHOULD omit any messages or message fragments which have already been ACKed. Once the messages have been sent, the implementation then enters the FINISHED state if this is the last flight in the handshake. Or, if the implementation expects to receive more messages, it sets a retransmit timer and then enters the WAITING state.

There are four ways to exit the WAITING state:

1. The retransmit timer expires: the implementation transitions to the SENDING state, where it retransmits the flight, resets the retransmit timer, and returns to the WAITING state.

2. The implementation reads a ACK from the peer: upon receiving an ACK for a partial flight (as mentioned in Section 7.1), the implementation transitions to the SENDING state, where it retransmits the unacked portion of the flight, resets the retransmit timer, and returns to the WAITING state. Upon receiving an ACK for a complete flight, the implementation cancels all retransmissions and either remains in WAITING, or, if the ACK was for the final flight, transitions to FINISHED.

3. The implementation reads a retransmitted flight from the peer: the implementation transitions to the SENDING state, where it retransmits the flight, resets the retransmit timer, and returns to the WAITING state. The rationale here is that the receipt of

a duplicate message is the likely result of timer expiry on the peer and therefore suggests that part of one's previous flight was lost.

4. The implementation receives some or all next flight of messages: if this is the final flight of messages, the implementation transitions to FINISHED. If the implementation needs to send a new flight, it transitions to the PREPARING state. Partial reads (whether partial messages or only some of the messages in the flight) may also trigger the implementation to send an ACK, as described in Section 7.1.

Because DTLS clients send the first message (ClientHello), they start in the PREPARING state. DTLS servers start in the WAITING state, but with empty buffers and no retransmit timer.

In addition, for at least twice the default Maximum Segment Lifetime (MSL) defined for [RFC0793], when in the FINISHED state, the server MUST respond to retransmission of the client's second flight with a retransmit of its ACK.

Note that because of packet loss, it is possible for one side to be sending application data even though the other side has not received the first side's Finished message. Implementations MUST either discard or buffer all application data packets for the new epoch until they have received the Finished message for that epoch. Implementations MAY treat receipt of application data with a new epoch prior to receipt of the corresponding Finished message as evidence of reordering or packet loss and retransmit their final flight immediately, shortcutting the retransmission timer.

5.7.2. Timer Values

Though timer values are the choice of the implementation, mishandling of the timer can lead to serious congestion problems; for example, if many instances of a DTLS time out early and retransmit too quickly on a congested link. Implementations SHOULD use an initial timer value of 100 msec (the minimum defined in RFC 6298 [RFC6298]) and double the value at each retransmission, up to no less than the RFC 6298 maximum of 60 seconds. Application specific profiles, such as those used for the Internet of Things environment, may recommend longer timer values. Note that a 100 msec timer is recommend rather than the 3-second RFC 6298 default in order to improve latency for time-sensitive applications. Because DTLS only uses retransmission for handshake and not dataflow, the effect on congestion should be minimal.

Implementations SHOULD retain the current timer value until a transmission without loss occurs, at which time the value may be reset to the initial value. After a long period of idleness, no less than 10 times the current timer value, implementations may reset the timer to the initial value. One situation where this might occur is when a rehandshake is used after substantial data transfer.

5.8. CertificateVerify and Finished Messages

CertificateVerify and Finished messages have the same format as in TLS 1.3. Hash calculations include entire handshake messages, including DTLS-specific fields: message_seq, fragment_offset, and fragment_length. However, in order to remove sensitivity to handshake message fragmentation, the CertificateVerify and the Finished messages MUST be computed as if each handshake message had been sent as a single fragment following the algorithm described in Section 4.4.3 and Section 4.4.4 of [TLS13], respectively.

5.9. Alert Messages

Note that Alert messages are not retransmitted at all, even when they occur in the context of a handshake. However, a DTLS implementation which would ordinarily issue an alert SHOULD generate a new alert message if the offending record is received again (e.g., as a retransmitted handshake message). Implementations SHOULD detect when a peer is persistently sending bad messages and terminate the local connection state after such misbehavior is detected.

5.10. Establishing New Associations with Existing Parameters

If a DTLS client-server pair is configured in such a way that repeated connections happen on the same host/port quartet, then it is possible that a client will silently abandon one connection and then initiate another with the same parameters (e.g., after a reboot). This will appear to the server as a new handshake with epoch=0. In cases where a server believes it has an existing association on a given host/port quartet and it receives an epoch=0 ClientHello, it SHOULD proceed with a new handshake but MUST NOT destroy the existing association until the client has demonstrated reachability either by completing a cookie exchange or by completing a complete handshake including delivering a verifiable Finished message. After a correct Finished message is received, the server MUST abandon the previous association to avoid confusion between two valid associations with overlapping epochs. The reachability requirement prevents off-path/ blind attackers from destroying associations merely by sending forged ClientHellos.

Note: it is not always possible to distinguish which association a given packet is from. For instance, if the client performs a handshake, abandons the connection, and then immediately starts a new handshake, it may not be possible to tell which connection a given protected record is for. In these cases, trial decryption MAY be necessary, though implementations could also use some sort of connection identifier, such as the one specified in [I-D.rescorla-tls-dtls-connection-id].

6. Example of Handshake with Timeout and Retransmission

The following is an example of a handshake with lost packets and retransmissions.

```
   Client                                                Server
   ------                                                ------

    Record 0                  -------->
    ClientHello
    (message_seq=0)
     +cookie

                                X<-----                Record 0
                                (lost)              ServerHello
                                                (message_seq=1)
                                            EncryptedExtensions
                                                (message_seq=2)
                                                    Certificate
                                                (message_seq=3)


                              <--------                Record 1
                                              CertificateVerify
                                                (message_seq=4)
                                                       Finished
                                                (message_seq=5)

    Record 1                  -------->
    ACK [1]


                              <--------                Record 2
                                                    ServerHello
                                                (message_seq=1)
                                            EncryptedExtensions
                                                (message_seq=2)
                                                    Certificate
                                                (message_seq=3)

    Record 2                  -------->
    Certificate
    (message_seq=2)
    CertificateVerify
    (message_seq=3)
    Finished
    (message_seq=4)

                              <--------              Record 3
                                                      ACK [2]
```

Figure 11: Example DTLS exchange illustrating message loss

6.1. Epoch Values and Rekeying

A recipient of a DTLS message needs to select the correct keying material in order to process an incoming message. With the possibility of message loss and re-order an identifier is needed to determine which cipher state has been used to protect the record payload. The epoch value fulfills this role in DTLS. In addition to the key derivation steps described in Section 7 of [TLS13] triggered by the states during the handshake a sender may want to rekey at any time during the lifetime of the connection and has to have a way to indicate that it is updating its sending cryptographic keys.

This version of DTLS assigns dedicated epoch values to messages in the protocol exchange to allow identification of the correct cipher state:

- epoch value (0) is used with unencrypted messages. There are three unencrypted messages in DTLS, namely ClientHello, ServerHello, and HelloRetryRequest.

- epoch value (1) is used for messages protected using keys derived from client_early_traffic_secret. This includes early data sent by the client and the EndOfEarlyData message.

- epoch value (2) is used for messages protected using keys derived from [sender]_handshake_traffic_secret. Messages transmitted during the initial handshake, such as EncryptedExtensions, CertificateRequest, Certificate, CertificateVerify, and Finished belong to this category. Note, however, post-handshake are protected under the appropriate application traffic key and are not included in this category.

- epoch value (3) is used for payloads protected using keys derived from the initial traffic_secret_0. This may include handshake messages, such as post-handshake messages (e.g., a NewSessionTicket message).

- epoch value (4 to 2^16-1) is used for payloads protected using keys from the traffic_secret_N (N>0).

Using these reserved epoch values a receiver knows what cipher state has been used to encrypt and integrity protect a message. Implementations that receive a payload with an epoch value for which no corresponding cipher state can be determined MUST generate a "unexpected_message" alert. For example, client incorrectly uses epoch value 5 when sending early application data in a 0-RTT exchange. A server will not be able to compute the appropriate keys and will therefore have to respond with an alert.

Note that epoch values do not wrap. If a DTLS implementation would need to wrap the epoch value, it MUST terminate the connection.

The traffic key calculation is described in Section 7.3 of [TLS13].

Figure 12 illustrates the epoch values in an example DTLS handshake.

```
Client                                             Server
------                                             ------

 ClientHello
 (epoch=0)
                            -------->

                            <--------       HelloRetryRequest
                                                    (epoch=0)

 ClientHello                -------->
 (epoch=0)

                            <--------             ServerHello
                                                    (epoch=0)
                                        {EncryptedExtensions}
                                                    (epoch=2)
                                                {Certificate}
                                                    (epoch=2)
                                          {CertificateVerify}
                                                    (epoch=2)
                                                   {Finished}
                                                    (epoch=2)

 {Certificate}              -------->
 (epoch=2)
 {CertificateVerify}
 (epoch=2)
 {Finished}
 (epoch=2)

                            <--------                   [ACK]
                                                    (epoch=3)

 [Application Data]         -------->
 (epoch=3)

                            <--------      [Application Data]
                                                    (epoch=3)

                         Some time later ...
```

```
                  (Post-Handshake Message Exchange)

                             <--------      [NewSessionTicket]
                                                     (epoch=3)

    [ACK]                    -------->
    (epoch=3)

                          Some time later ...
                            (Rekeying)

                             <--------      [Application Data]
                                                     (epoch=4)
    [Application Data]       -------->
    (epoch=4)
```

Figure 12: Example DTLS exchange with epoch information

7. ACK Message

The ACK message is used by an endpoint to indicate handshake-containing the TLS records it has received from the other side. ACK is not a handshake message but is rather a separate content type, with code point TBD (proposed, 25). This avoids it consuming space in the handshake message sequence. Note that ACKs can still be piggybacked on the same UDP datagram as handshake records.

```
struct {
    uint64 record_numbers<0..2^16-1>;
} ACK;
```

record_numbers: a list of the records containing handshake messages in the current flight which the endpoint has received, in numerically increasing order. ACKs only cover the current outstanding flight (this is possible because DTLS is generally a lockstep protocol). Thus, an ACK from the server would not cover both the ClientHello and the client's Certificate. Implementations can accomplish this by clearing their ACK list upon receiving the start of the next flight.

ACK records MUST be sent with an epoch that is equal to or higher than the record which is being acknowledged. Implementations SHOULD simply use the current key.

7.1. Sending ACKs

When an implementation receives a partial flight, it SHOULD generate an ACK that covers the messages from that flight which it has received so far. Implementations have some discretion about when to generate ACKs, but it is RECOMMENDED that they do so under two circumstances:

- When they receive a message or fragment which is out of order, either because it is not the next expected message or because it is not the next piece of the current message. Implementations MUST NOT send ACKs for handshake messages which they discard as out-of-order, because otherwise those messages will not be retransmitted.

- When they have received part of a flight and do not immediately receive the rest of the flight (which may be in the same UDP datagram). A reasonable approach here is to set a timer for 1/4 the current retransmit timer value when the first record in the flight is received and then send an ACK when that timer expires.

In addition, implementations MUST send ACKs upon receiving all of any flight which they do not respond to with their own messages. Specifically, this means the client's final flight of the main handshake, the server's transmission of the NewSessionTicket, and KeyUpdate messages. ACKs SHOULD NOT be sent for other complete flights because they are implicitly acknowledged by the receipt of the next flight, which generally immediately follows the flight. Each NewSessionTicket or KeyUpdate is an individual flight; in particular, a KeyUpdate sent in response to a KeyUpdate with update_requested does not implicitly acknowledge that message. Implementations MAY ACK the records corresponding to each transmission of that flight or simply ACK the most recent one.

ACKs MUST NOT be sent for other records of any content type other than handshake or for records which cannot be unprotected.

Note that in some cases it may be necessary to send an ACK which does not contain any record numbers. For instance, a client might receive an EncryptedExtensions message prior to receiving a ServerHello. Because it cannot decrypt the EncryptedExtensions, it cannot safely ACK it (as it might be damaged). If the client does not send an ACK, the server will eventually retransmit its first flight, but this might take far longer than the actual round trip time between client and server. Having the client send an empty ACK shortcuts this process.

7.2. Receiving ACKs

When an implementation receives an ACK, it SHOULD record that the messages or message fragments sent in the records being ACKed were received and omit them from any future retransmissions. Upon receipt of an ACK for only some messages from a flight, an implementation SHOULD retransmit the remaining messages or fragments. Note that this requires implementations to track which messages appear in which records. Once all the messages in a flight have been acknowledged, the implementation MUST cancel all retransmissions of that flight. As noted above, the receipt of any packet responding to a given flight MUST be taken as an implicit ACK for the entire flight.

8. Key Updates

As with TLS 1.3, DTLS 1.3 implementations send a KeyUpdate message to indicate that they are updating their sending keys. As with other handshake messages with no built-in response, KeyUpdates MUST be acknowledged. In order to facilitate epoch reconstruction Section 4.2.2 implementations MUST NOT send with the new keys or send a new KeyUpdate until the previous KeyUpdate has been acknowledged (this avoids having too many epochs in active use).

Due to loss and/or re-ordering, DTLS 1.3 implementations may receive a record with an older epoch than the current one (the requirements above preclude receiving a newer record). They SHOULD attempt to process those records with that epoch (see Section 4.2.2 for information on determining the correct epoch), but MAY opt to discard such out-of-epoch records.

Although KeyUpdate MUST be ACKed, it is possible for the ACK to be lost, in which case the sender of the KeyUpdate will retransmit it. Implementations MUST retain the ability to ACK the KeyUpdate for up to 2MSL. It is RECOMMENDED that they do so by retaining the pre-update keying material, but they MAY do so by responding to messages which appear to be out-of-epoch with a canned ACK message; in this case, implementations SHOULD rate limit how often they send such ACKs.

9. Connection ID Updates

If the client and server have negotiated the "connection_id" extension [DTLS-CID], either side can send a new connection ID which it wishes the other side to use in a NewConnectionId message.

```
enum {
    cid_immediate(0), cid_spare(1), (255)
} ConnectionIdUsage;

opaque ConnectionId<0..2^8-1>;

struct {
    ConnectionIds cids<0..2^16-1>;
    ConnectionIdUsage usage;
} NewConnectionId;
```

cid Indicates the set of CIDs which the sender wishes the peer to use.

usage Indicates whether the new CIDs should be used immediately or are spare. If usage is set to "cid_immediate", then one of the new CID MUST be used immediately for all future records. If it is set to "cid_spare", then either existing or new CID MAY be used.

Endpoints SHOULD use receiver-provided CIDs in the order they were provided. Endpoints MUST NOT have more than one NewConnectionId message outstanding.

If the client and server have negotiated the "connection_id" extension, either side can request a new CID using the RequestConnectionId message.

```
struct {
  uint8 num_cids;
} RequestConnectionId;
```

num_cids The number of CIDs desired.

Endpoints SHOULD respond to RequestConnectionId by sending a NewConnectionId with usage "cid_spare" containing num_cid CIDs soon as possible. Endpoints MUST NOT send a RequestConnectionId message when an existing request is still unfulfilled; this implies that endpoints needs to request new CIDs well in advance. An endpoint MAY ignore requests, which it considers excessive (though they MUST be ACKed as usual).

Endpoints MUST NOT send either of these messages if they did not negotiate a connection ID. If an implementation receives these messages when connection IDs were not negotiated, it MUST abort the connection with an unexpected_message alert.

9.1. ID Example

Below is an example exchange for DTLS 1.3 using a single connection id in each direction.

Note: The connection_id extension is defined in [DTLS-CID], which is used in ClientHello and ServerHello messages.

```
   Client                                             Server
   ------                                             ------

   ClientHello
   (connection_id=5)
                               -------->


                               <--------      HelloRetryRequest
                                                       (cookie)

   ClientHello                 -------->
   (connection_id=5)
     +cookie

                               <--------            ServerHello
                                            (connection_id=100)
                                            EncryptedExtensions
                                                        (cid=5)
                                                    Certificate
                                                        (cid=5)
                                              CertificateVerify
                                                        (cid=5)
                                                       Finished
                                                        (cid=5)

   Certificate                -------->
   (cid=100)
   CertificateVerify
   (cid=100)
   Finished
   (cid=100)
                              <--------                     Ack
                                                        (cid=5)

   Application Data           ========>
   (cid=100)
                              <========        Application Data
                                                        (cid=5)
```

Figure 13: Example DTLS 1.3 Exchange with Connection IDs

10. Application Data Protocol

Application data messages are carried by the record layer and are fragmented and encrypted based on the current connection state. The messages are treated as transparent data to the record layer.

11. Security Considerations

Security issues are discussed primarily in [TLS13].

The primary additional security consideration raised by DTLS is that of denial of service. DTLS includes a cookie exchange designed to protect against denial of service. However, implementations that do not use this cookie exchange are still vulnerable to DoS. In particular, DTLS servers that do not use the cookie exchange may be used as attack amplifiers even if they themselves are not experiencing DoS. Therefore, DTLS servers SHOULD use the cookie exchange unless there is good reason to believe that amplification is not a threat in their environment. Clients MUST be prepared to do a cookie exchange with every handshake.

Unlike TLS implementations, DTLS implementations SHOULD NOT respond to invalid records by terminating the connection.

If implementations process out-of-epoch records as recommended in Section 8, then this creates a denial of service risk since an adversary could inject packets with fake epoch values, forcing the recipient to compute the next-generation application_traffic_secret using the HKDF-Expand-Label construct to only find out that the message was does not pass the AEAD cipher processing. The impact of this attack is small since the HKDF-Expand-Label only performs symmetric key hashing operations. Implementations which are concerned about this form of attack can discard out-of-epoch records.

The security and privacy properties of the connection ID for DTLS 1.3 builds on top of what is described in [DTLS-CID]. There are, however, several improvements:

- The use of the Post-Handshake message allows the client and the server to update their connection IDs and those values are exchanged with confidentiality protection.

- With multi-homing, an adversary is able to correlate the communication interaction over the two paths, which adds further privacy concerns. In order to prevent this, implementations SHOULD attempt to use fresh connection IDs whenever they change local addresses or ports (though this is not always possible to detect). The RequestConnectionId message can be used to ask for new IDs in order to ensure that you have a pool of suitable IDs.

- Switching connection ID based on certain events, or even regularly, helps against tracking by onpath adversaries but the sequence numbers can still allow linkability. For this reason

this specification defines an algorithm for encrypting sequence numbers, see Section 4.2.3.

- Since the DTLS 1.3 exchange encrypts handshake messages much earlier than in previous DTLS versions information identifying the DTLS client, such as the client certificate, less information is available to an on-path adversary.

12. Changes to DTLS 1.2

Since TLS 1.3 introduces a large number of changes to TLS 1.2, the list of changes from DTLS 1.2 to DTLS 1.3 is equally large. For this reason this section focuses on the most important changes only.

- New handshake pattern, which leads to a shorter message exchange

- Support for AEAD-only ciphers

- HelloRetryRequest of TLS 1.3 used instead of HelloVerifyRequest

- More flexible ciphersuite negotiation

- New session resumption mechanism

- PSK authentication redefined

- New key derivation hierarchy utilizing a new key derivation construct

- Removed support for weaker and older cryptographic algorithms

- Improved version negotiation

- Optimized record layer encoding and thereby its size

- Added connection ID functionality

- Sequence numbers are encrypted.

13. IANA Considerations

IANA is requested to allocate a new value in the "TLS ContentType" registry for the ACK message, defined in Section 7, with content type 25. IANA is requested to reserve the content type range 32-63 so that content types in this range are not allocated.

IANA is requested to allocate two values in the "TLS Handshake Type" registry, defined in [TLS13], for RequestConnectionId (TBD), and NewConnectionId (TBD), as defined in this document.

14. References

14.1. Normative References

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <https://www.rfc-editor.org/info/rfc768>.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <https://www.rfc-editor.org/info/rfc1191>.

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

[RFC4443] Conta, A., Deering, S., and M. Gupta, Ed., "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification", STD 89, RFC 4443, DOI 10.17487/RFC4443, March 2006, <https://www.rfc-editor.org/info/rfc4443>.

[RFC4821] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, DOI 10.17487/RFC4821, March 2007, <https://www.rfc-editor.org/info/rfc4821>.

[RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <https://www.rfc-editor.org/info/rfc6298>.

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

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

14.2. Informative References

[DTLS-CID]
Rescorla, E., Tschofenig, H., Fossati, T., and T. Gondrom, "Connection Identifiers for DTLS 1.2", draft-ietf-tls-dtls-connection-id-02 (work in progress), October 2018.

[I-D.rescorla-tls-dtls-connection-id]
Rescorla, E., Tschofenig, H., Fossati, T., and T. Gondrom, "The Datagram Transport Layer Security (DTLS) Connection Identifier", draft-rescorla-tls-dtls-connection-id-02 (work in progress), November 2017.

[RFC2522] Karn, P. and W. Simpson, "Photuris: Session-Key Management Protocol", RFC 2522, DOI 10.17487/RFC2522, March 1999, <https://www.rfc-editor.org/info/rfc2522>.

[RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <https://www.rfc-editor.org/info/rfc4303>.

[RFC4340] Kohler, E., Handley, M., and S. Floyd, "Datagram Congestion Control Protocol (DCCP)", RFC 4340, DOI 10.17487/RFC4340, March 2006, <https://www.rfc-editor.org/info/rfc4340>.

[RFC4346] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, DOI 10.17487/RFC4346, April 2006, <https://www.rfc-editor.org/info/rfc4346>.

[RFC4347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", RFC 4347, DOI 10.17487/RFC4347, April 2006, <https://www.rfc-editor.org/info/rfc4347>.

[RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007, <https://www.rfc-editor.org/info/rfc4960>.

[RFC5238] Phelan, T., "Datagram Transport Layer Security (DTLS) over the Datagram Congestion Control Protocol (DCCP)", RFC 5238, DOI 10.17487/RFC5238, May 2008, <https://www.rfc-editor.org/info/rfc5238>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <https://www.rfc-editor.org/info/rfc6347>.

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

[RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <https://www.rfc-editor.org/info/rfc7525>.

14.3. URIs

[1] mailto:tls@ietf.org

[2] https://www1.ietf.org/mailman/listinfo/tls

[3] https://www.ietf.org/mail-archive/web/tls/current/index.html

Appendix A. Protocol Data Structures and Constant Values

This section provides the normative protocol types and constants definitions.

A.1. Record Layer

```
struct {
    ContentType type;
    ProtocolVersion legacy_record_version;
    uint16 epoch = 0                                 // DTLS field
    uint48 sequence_number;                          // DTLS field
    uint16 length;
    opaque fragment[DTLSPlaintext.length];
} DTLSPlaintext;

struct {
     opaque content[DTLSPlaintext.length];
     ContentType type;
     uint8 zeros[length_of_padding];
} DTLSInnerPlaintext;

struct {
    opaque unified_hdr[variable];
    opaque encrypted_record[length];
} DTLSCiphertext;

0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|0|0|1|C|S|L|E E|
+-+-+-+-+-+-+-+-+
| Connection ID |   Legend:
| (if any,      |
/  length as    /   C   - CID present
|  negotiated)  |   S   - Sequence number length
+-+-+-+-+-+-+-+-+   L   - Length present
|  8 or 16 bit  |   E   - Epoch
|Sequence Number|
+-+-+-+-+-+-+-+-+
| 16 bit Length |
| (if present)  |
+-+-+-+-+-+-+-+-+
```

A.2. Handshake Protocol

```
enum {
    hello_request_RESERVED(0),
    client_hello(1),
```

```
       server_hello(2),
       hello_verify_request_RESERVED(3),
       new_session_ticket(4),
       end_of_early_data(5),
       hello_retry_request_RESERVED(6),
       encrypted_extensions(8),
       certificate(11),
       server_key_exchange_RESERVED(12),
       certificate_request(13),
       server_hello_done_RESERVED(14),
       certificate_verify(15),
       client_key_exchange_RESERVED(16),
       finished(20),
       key_update(24),
       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 (HandshakeType) {
           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;
       } body;
   } Handshake;

   uint16 ProtocolVersion;
   opaque Random[32];

   uint8 CipherSuite[2];    /* Cryptographic suite selector */

   struct {
       ProtocolVersion legacy_version = { 254,253 }; // DTLSv1.2
       Random random;
       opaque legacy_session_id<0..32>;
       opaque legacy_cookie<0..2^8-1>;                  // DTLS
```

```
       CipherSuite cipher_suites<2..2^16-2>;
       opaque legacy_compression_methods<1..2^8-1>;
       Extension extensions<8..2^16-1>;
   } ClientHello;
```

A.3. ACKs

```
   struct {
       uint64 record_numbers<0..2^16-1>;
   } ACK;
```

A.4. Connection ID Management

```
   enum {
       cid_immediate(0), cid_spare(1), (255)
   } ConnectionIdUsage;

   opaque ConnectionId<0..2^8-1>;

   struct {
       ConnectionIds cids<0..2^16-1>;
       ConnectionIdUsage usage;
   } NewConnectionId;

   struct {
     uint8 num_cids;
   } RequestConnectionId;
```

Appendix B. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

IETF Drafts draft-29: - Added support for sequence number encryption - Update to new record format - Emphasize that compatibility mode isn't used.

draft-28: - Version bump to align with TLS 1.3 pre-RFC version.

draft-27: - Incorporated unified header format. - Added support for connection IDs.

draft-04 - 26: - Submissions to align with TLS 1.3 draft versions

draft-03 - Only update keys after KeyUpdate is ACKed.

draft-02 - Shorten the protected record header and introduce an ultra-short version of the record header. - Reintroduce KeyUpdate, which works properly now that we have ACK. - Clarify the ACK rules.

draft-01 - Restructured the ACK to contain a list of packets and also be a record rather than a handshake message.

draft-00 - First IETF Draft

Personal Drafts draft-01 - Alignment with version -19 of the TLS 1.3 specification

draft-00

- Initial version using TLS 1.3 as a baseline.

- Use of epoch values instead of KeyUpdate message

- Use of cookie extension instead of cookie field in ClientHello and HelloVerifyRequest messages

- Added ACK message

- Text about sequence number handling

Appendix C. Working Group Information

The discussion list for the IETF TLS working group is located at the e-mail address tls@ietf.org [1]. Information on the group and information on how to subscribe to the list is at https://www1.ietf.org/mailman/listinfo/tls [2]

Archives of the list can be found at: https://www.ietf.org/mail-archive/web/tls/current/index.html [3]

Appendix D. Contributors

Many people have contributed to previous DTLS versions and they are acknowledged in prior versions of DTLS specifications or in the referenced specifications. The sequence number encryption concept is taken from the QUIC specification. We would like to thank the authors of the QUIC specification for their work.

In addition, we would like to thank:

* Ilari Liusvaara
  Independent
  ilariliusvaara@welho.com

* Martin Thomson
  Mozilla
  martin.thomson@gmail.com

* Yin Xinxing
  Huawei
  yinxinxing@huawei.com

* Thomas Fossati
  Nokia
  thomas.fossati@nokia.com

* Tobias Gondrom
  Huawei
  tobias.gondrom@gondrom.org

Authors' Addresses

Eric Rescorla
RTFM, Inc.

EMail: ekr@rtfm.com

Hannes Tschofenig
Arm Limited

EMail: hannes.tschofenig@arm.com

Nagendra Modadugu
Google, Inc.

EMail: nagendra@cs.stanford.edu