

TLS
Internet-Draft
Obsoletes: 6347 (if approved)
Intended status: Standards Track
Expires: 1 November 2021

E. Rescorla
RTFM, Inc.
H. Tschofenig
Arm Limited
N. Modadugu
Google, Inc.
30 April 2021

The Datagram Transport Layer Security (DTLS) Protocol Version 1.3
draft-ietf-tls-dtls13-43

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 with the exception of order protection/non-replayability. Datagram semantics of the underlying transport are preserved by the DTLS protocol.

This document obsoletes RFC 6347.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 1 November 2021.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

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

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Introduction	4
2. Conventions and Terminology	4
3. DTLS Design Rationale and Overview	6
3.1. Packet Loss	7
3.2. Reordering	8
3.3. Fragmentation	8
3.4. Replay Detection	8
4. The DTLS Record Layer	8
4.1. Demultiplexing DTLS Records	12
4.2. Sequence Number and Epoch	14
4.2.1. Processing Guidelines	14
4.2.2. Reconstructing the Sequence Number and Epoch	15
4.2.3. Record Number Encryption	15
4.3. Transport Layer Mapping	16
4.4. PMTU Issues	17
4.5. Record Payload Protection	19
4.5.1. Anti-Replay	19
4.5.2. Handling Invalid Records	20
4.5.3. AEAD Limits	20
5. The DTLS Handshake Protocol	22
5.1. Denial-of-Service Countermeasures	22
5.2. DTLS Handshake Message Format	25
5.3. ClientHello Message	27
5.4. ServerHello Message	28
5.5. Handshake Message Fragmentation and Reassembly	28

5.6.	End Of Early Data	29
5.7.	DTLS Handshake Flights	30
5.8.	Timeout and Retransmission	34
5.8.1.	State Machine	34
5.8.2.	Timer Values	37
5.8.3.	Large Flight Sizes	38
5.8.4.	State machine duplication for post-handshake messages	38
5.9.	CertificateVerify and Finished Messages	40
5.10.	Cryptographic Label Prefix	40
5.11.	Alert Messages	40
5.12.	Establishing New Associations with Existing Parameters	40
6.	Example of Handshake with Timeout and Retransmission	41
6.1.	Epoch Values and Rekeying	42
7.	ACK Message	45
7.1.	Sending ACKs	46
7.2.	Receiving ACKs	47
7.3.	Design Rationale	48
8.	Key Updates	48
9.	Connection ID Updates	50
9.1.	Connection ID Example	51
10.	Application Data Protocol	53
11.	Security Considerations	53
12.	Changes since DTLS 1.2	55
13.	Updates affecting DTLS 1.2	56
14.	IANA Considerations	56
15.	References	57
15.1.	Normative References	57
15.2.	Informative References	58
Appendix A.	Protocol Data Structures and Constant Values	61
A.1.	Record Layer	61
A.2.	Handshake Protocol	62
A.3.	ACKs	64
A.4.	Connection ID Management	64
Appendix B.	Analysis of Limits on CCM Usage	64
B.1.	Confidentiality Limits	65
B.2.	Integrity Limits	66
B.3.	Limits for AEAD_AES_128_CCM_8	66
Appendix C.	Implementation Pitfalls	67
Appendix D.	History	67
Appendix E.	Working Group Information	70
Appendix F.	Contributors	70
Appendix G.	Acknowledgements	71
Authors' Addresses	71

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 establish an authenticated, confidentiality and integrity protected channel 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 use UDP [RFC0768] as a transport and to offer communication security protection for those applications the Datagram Transport Layer Security (DTLS) protocol has been developed. 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 as a delta from TLS 1.3 [TLS13]. It obsoletes DTLS 1.2.

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] and [DEPRECATE].

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.
- * **association:** Shared state between two endpoints established with a DTLS handshake.
- * **connection:** Synonym for association.
- * **endpoint:** Either the client or server of the connection.
- * **epoch:** one set of cryptographic keys used for encryption and decryption.
- * **handshake:** An initial negotiation between client and server that establishes the parameters of the connection.
- * **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.
- * **server:** The endpoint which did not initiate the DTLS connection.
- * **CID:** Connection ID
- * **MSL:** Maximum Segment Lifetime

The reader is assumed to be familiar with [TLS13]. 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.

DTLS 1.3 uses network byte order (big-endian) format for encoding messages based on the encoding format defined in [TLS13] and earlier (D)TLS specifications.

The reader is also assumed to be familiar with [I-D.ietf-tls-dtls-connection-id] as this document applies the CID functionality to DTLS 1.3.

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 reordered data traffic. Note that while low-latency streaming and gaming use DTLS to protect data (e.g. for protection of a WebRTC data channel), telephony utilizes DTLS for key establishment, and Secure Real-time Transport Protocol (SRTP) for protection of data [RFC5763].

TLS cannot be used directly over datagram transports the following five reasons:

1. TLS relies on an implicit sequence number on records. If a record is not received, then the recipient will use the wrong sequence number when attempting to remove record protection from subsequent records. DTLS solves this problem by adding sequence numbers to records.
2. The TLS handshake is a lock-step cryptographic protocol. Messages must be transmitted and received in a defined order; any other order is an error. The DTLS handshake includes message sequence numbers to enable fragmented message reassembly and in-order delivery in case datagrams are lost or reordered.
3. During the handshake, messages are implicitly acknowledged by other handshake messages. Some handshake messages, such as the `NewSessionTicket` message, do not result in any direct response that would allow the sender to detect loss. DTLS adds an acknowledgment message to enable better loss recovery.

4. Handshake messages are potentially larger than can be contained in a single datagram. DTLS adds fields to handshake messages to support fragmentation and reassembly.
5. Datagram transport protocols, like UDP, are susceptible to abusive behavior effecting denial of service attacks against nonparticipants. DTLS adds a return-routability check and DTLS 1.3 uses the TLS 1.3 HelloRetryRequest message (see Section 5.1 for details).

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:

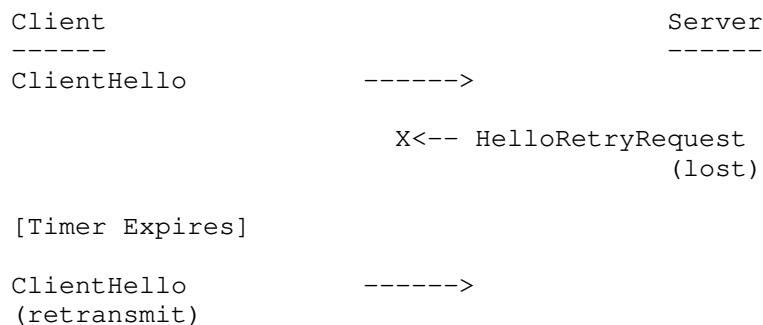


Figure 1: DTLS retransmission example

Once the client has transmitted the ClientHello message, it expects to see a HelloRetryRequest or a ServerHello from the server. However, if the timer expires, the client knows that either the ClientHello or the response from the server has been lost, which causes the the client to retransmit the ClientHello. When the server receives the retransmission, it knows to retransmit its HelloRetryRequest or ServerHello.

The server also maintains a retransmission timer for messages it sends (other than HelloRetryRequest) and retransmits when that timer expires. Not applying retransmissions to the HelloRetryRequest avoids the need to create 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.

For more detail on timeouts and retransmission, see Section 5.8.

3.2. Reordering

In DTLS, each handshake message is assigned a specific sequence number. 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.

3.3. Fragmentation

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 UDP datagram (see Section 4.4 for guidance). 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.

3.4. 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 1.3 record layer is different from the TLS 1.3 record layer and also different from the DTLS 1.2 record layer.

1. The DTLSCiphertext structure omits the superfluous version number and type fields.
2. DTLS adds an 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 from those in previous versions.
3. The DTLSCiphertext structure has a variable length header.

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;
    uint48 sequence_number;
    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 Formats

legacy_record_version This value MUST be set to {254, 253} for all records other than the initial ClientHello (i.e., one not generated after a HelloRetryRequest), where it may also be {254, 255} for compatibility purposes. It MUST be ignored for all purposes. See [TLS13]; Appendix D.1 for the rationale for this.

unified_hdr: The unified header (unified_hdr) is a structure of variable length, as shown in Figure 3.

encrypted_record: The AEAD-encrypted form of the serialized DTLSInnerPlaintext structure.

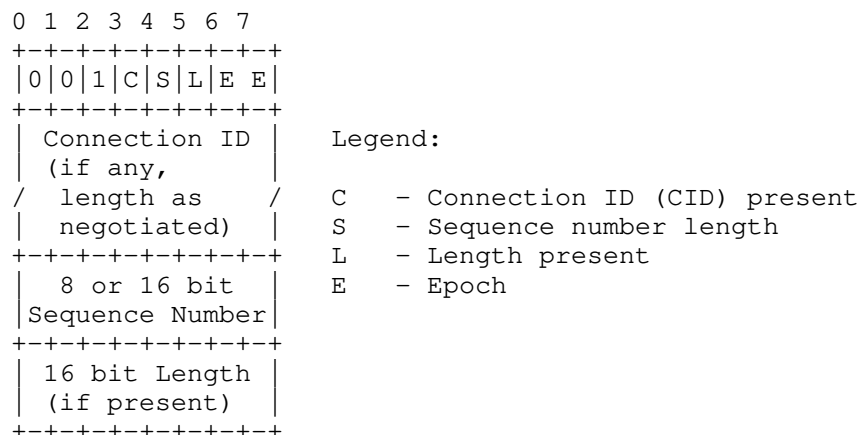


Figure 3: DTLS 1.3 Unified Header

Fixed Bits: The three high bits of the first byte of the unified header are set to 001. This ensures that the value will fit within the DTLS region when multiplexing is performed as described in [RFC7983]. It also ensures that distinguishing encrypted DTLS 1.3 records from encrypted DTLS 1.2 records is possible when they are carried on the same host/port quartet; such multiplexing is only possible when CIDs [I-D.ietf-tls-dtls-connection-id] are in use, in which case DTLS 1.2 records will have the content type `tls12_cid` (25).

C: The C bit (0x10) is set if the Connection ID is present.

S: The S bit (0x08) indicates the size of the sequence number. 0 means an 8-bit sequence number, 1 means 16-bit. Implementations MAY mix sequence numbers of different lengths on the same connection.

L: The L bit (0x04) is set if the length is present.

E: The two low bits (0x03) include the low order two bits of the epoch.

Connection ID: Variable length CID. The CID functionality is described in [I-D.ietf-tls-dtls-connection-id]. 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 headers.

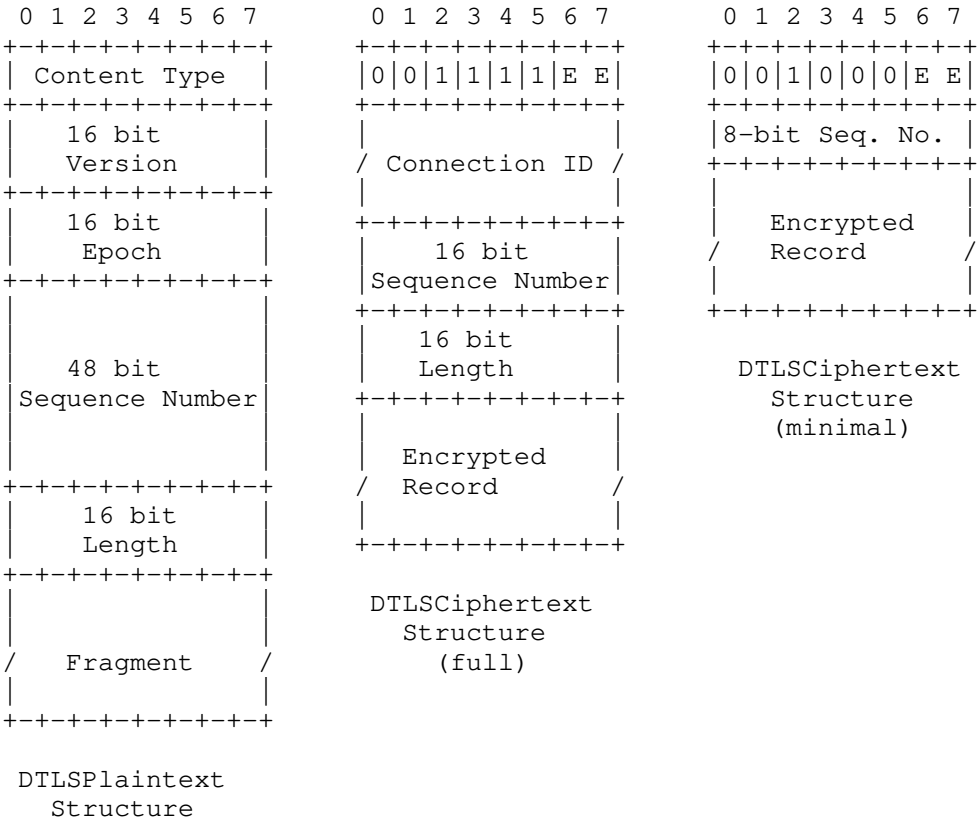


Figure 4: DTLS 1.3 Header Examples

The length field MAY be omitted by clearing the L bit, which means that 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 the last record in a datagram. Implementations MAY mix records with and without length fields on the same connection.

If a Connection ID is negotiated, then it MUST be contained in all datagrams. Sending implementations MUST NOT mix records from multiple DTLS associations in the same datagram. If the second or later record has a connection ID which does not correspond to the same association used for previous records, the rest of the datagram MUST be discarded.

When expanded, the epoch and sequence number can be combined into an unpacked RecordNumber structure, as shown below:

```
struct {  
    uint16 epoch;  
    uint48 sequence_number;  
} RecordNumber;
```

This 64-bit value is used in the ACK message as well as in the "record_sequence_number" input to the AEAD function.

The entire header value shown in Figure 4 (but prior to record number encryption, see Section 4.2.3) is used as the additional data value for the AEAD function. For instance, if the minimal variant is used, the AAD is 2 octets long. Note that this design is different from the additional data calculation for DTLS 1.2 and for DTLS 1.2 with Connection ID.

4.1. Demultiplexing DTLS Records

DTLS 1.3 uses a variable length record format and hence the demultiplexing process is more complex since more header formats need to be distinguished. Implementations can demultiplex DTLS 1.3 records by examining the first byte as follows:

- * If the first byte is alert(21), handshake(22), or ack(proposed, 26), the record MUST be interpreted as a DTLSPlaintext record.
- * If the first byte is any 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.

Figure 5 shows this demultiplexing procedure graphically taking DTLS 1.3 and earlier versions of DTLS into account.

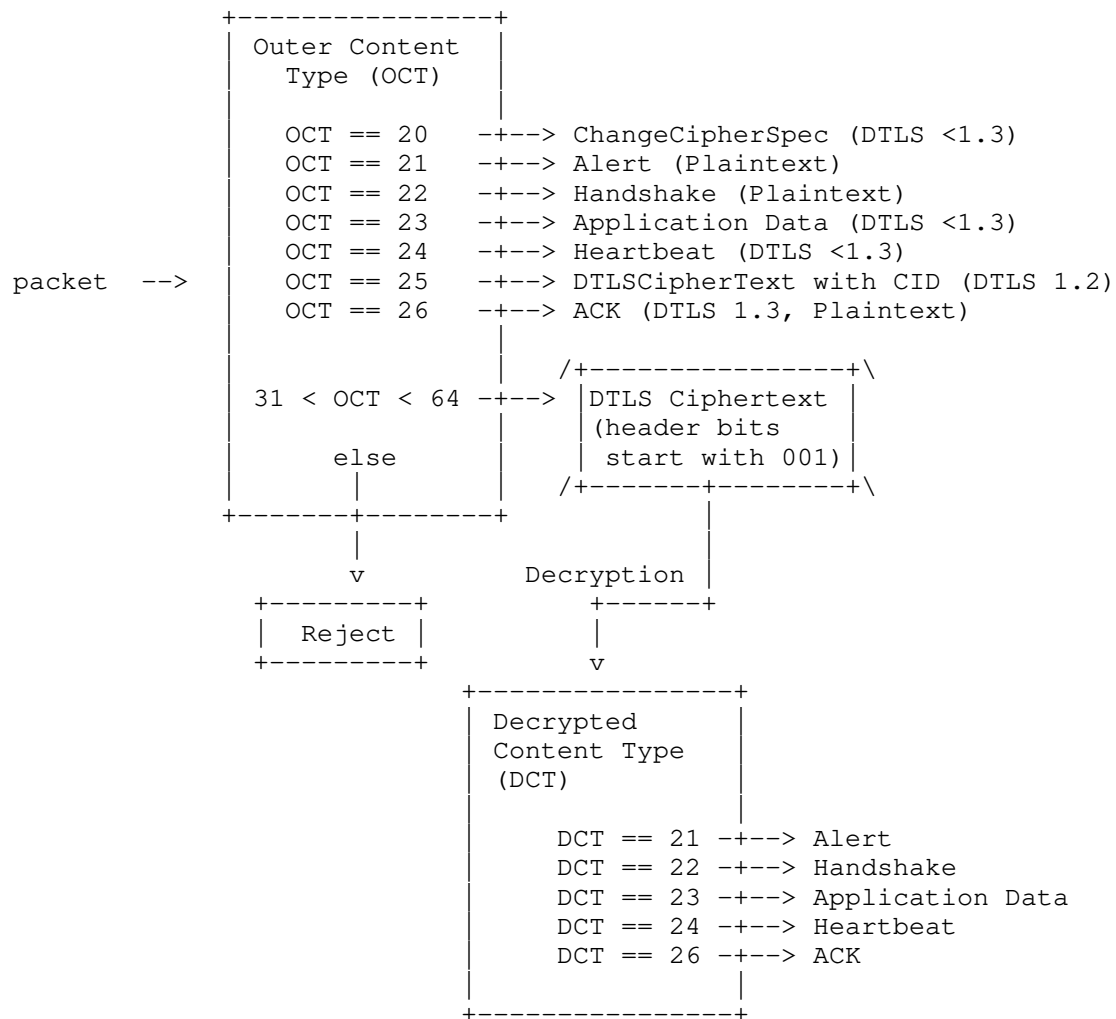


Figure 5: Demultiplexing DTLS 1.2 and DTLS 1.3 Records

Note: The optimized DTLS header format shown in Figure 3, which does not carry the Content Type in the Unified Header format, requires a different demultiplexing strategy compared to what was used in previous DTLS versions where the Content Type was conveyed in every record. As described in Figure 5, the first byte determines how an incoming DTLS record is demultiplexed. The first 3 bits of the first byte distinguish a DTLS 1.3 encrypted record from record types used in previous DTLS versions and plaintext DTLS 1.3 record types. Hence, the range 32 (0b0010 0000) to 63 (0b0011 1111) needs to be excluded from future allocations by IANA to avoid problems while demultiplexing; see Section 14.

4.2. Sequence Number and Epoch

DTLS uses an explicit or partly 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.

4.2.1. Processing Guidelines

Because DTLS records could be reordered, a record from epoch M may be received after epoch N (where $N > M$) has begun. Implementations SHOULD discard records from earlier epochs, but 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.)

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 records, 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 records may be sent still apply, and the receiver treats the records as if they were sent in the right order.

Implementations MUST send retransmissions of lost messages using the same epoch and keying material as the original transmission.

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, the recipient does not have a full epoch or sequence number value in the record 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 in the current epoch.

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 for that epoch as described above.

4.2.3. Record 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 4 bytes of the ciphertext as the block counter and the next 12 bytes as the nonce, passing them to the ChaCha20 block function (Section 2.3 of [CHACHA]):

```
Mask = ChaCha20(sn_key, Ciphertext[0..3], Ciphertext[4..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]. Note that a new key is used for each epoch: because the epoch is sent in the clear, this does not result in ambiguity.

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

This procedure requires the ciphertext length be at least 16 bytes. Receivers MUST reject shorter records as if they had failed deprotection, as described in Section 4.5.2. Senders MUST pad short plaintexts out (using the conventional record padding mechanism) in order to make a suitable-length ciphertext. Note most of the DTLS AEAD algorithms have a 16-byte authentication tag and need no padding. However, some algorithms such as TLS_AES_128_CCM_8_SHA256 have a shorter authentication tag and may require padding for short inputs.

Future cipher suites, which are not based on AES or ChaCha20, MUST define their own record sequence number encryption in order to be used with DTLS.

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 fragmented 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 Path MTU (PMTU) estimates obtained from the record layer. For more information about PMTU issues see Section 4.4.

Multiple DTLS records MAY be placed in a single datagram. Records are encoded consecutively. The length field from DTLS records containing that field can be used to determine the boundaries between records. The final record in a datagram can omit the length field. The first byte of the datagram payload MUST be the beginning of a record. Records MUST NOT span datagrams.

DTLS records without CIDs 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 without CIDs.

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 also allow the upper layer protocol to discover the amount of record expansion expected by the DTLS processing; alternately it MAY report PMTU estimates minus the estimated expansion from the transport layer and DTLS record framing.

Note that DTLS does not defend against spoofed ICMP messages; implementations SHOULD ignore any such messages that indicate PMTUs below the IPv4 and IPv6 minimums of 576 and 1280 bytes respectively.

If there is a transport protocol indication that the PMTU was exceeded (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] and [RFC4821] for IPv4 or via [RFC8201] for IPv6. 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, the handshake layer SHOULD immediately attempt to fragment the message, 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 specification 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]. Because each epoch resets the sequence number space, a separate sliding window is needed for each epoch.

The received record counter for an epoch MUST be initialized to zero when that epoch is first used. 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 in that epoch during the lifetime of the association. This check SHOULD happen after deprotecting the record; otherwise the record discard might itself serve as a timing channel for the record number. Note that computing the full record number from the partial is still a potential timing channel for the record number, though a less powerful one than whether the record was deprotected.

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.) The receiver SHOULD pick a window large enough to handle any plausible reordering, which depends on the data rate. (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 in the epoch. Records that contain sequence numbers lower than the "left" edge of the window are rejected. Records falling within the window are checked against a list of received records 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 record is to the right of the window, then the record is new.

The window **MUST NOT** be updated until the record 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 fatal 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, generating fatal alerts 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.

Note that because invalid records are rejected at a layer lower than the handshake state machine, they do not affect pending retransmission timers.

4.5.3. AEAD Limits

Section 5.5 of TLS [TLS13] defines limits on the number of records that can be protected using the same keys. These limits are specific to an AEAD algorithm, and apply equally to DTLS. Implementations **SHOULD NOT** protect more records than allowed by the limit specified for the negotiated AEAD. Implementations **SHOULD** initiate a key update before reaching this limit.

[TLS13] does not specify a limit for AEAD_AES_128_CCM, but the analysis in Appendix B shows that a limit of 2^{23} packets can be used to obtain the same confidentiality protection as the limits specified in TLS.

The usage limits defined in TLS 1.3 exist for protection against attacks on confidentiality and apply to successful applications of AEAD protection. The integrity protections in authenticated encryption also depend on limiting the number of attempts to forge packets. TLS achieves this by closing connections after any record fails an authentication check. In comparison, DTLS ignores any packet that cannot be authenticated, allowing multiple forgery attempts.

Implementations MUST count the number of received packets that fail authentication with each key. If the number of packets that fail authentication exceed a limit that is specific to the AEAD in use, an implementation SHOULD immediately close the connection. Implementations SHOULD initiate a key update with `update_requested` before reaching this limit. Once a key update has been initiated, the previous keys can be dropped when the limit is reached rather than closing the connection. Applying a limit reduces the probability that an attacker is able to successfully forge a packet; see [AEBounds] and [ROBUST].

For AEAD_AES_128_GCM, AEAD_AES_256_GCM, and AEAD_CHACHA20_POLY1305, the limit on the number of records that fail authentication is 2^{36} . Note that the analysis in [AEBounds] supports a higher limit for the AEAD_AES_128_GCM and AEAD_AES_256_GCM, but this specification recommends a lower limit. For AEAD_AES_128_CCM, the limit on the number of records that fail authentication is $2^{23.5}$; see Appendix B.

The AEAD_AES_128_CCM_8 AEAD, as used in TLS_AES_128_CCM_8_SHA256, does not have a limit on the number of records that fail authentication that both limits the probability of forgery by the same amount and does not expose implementations to the risk of denial of service; see Appendix B.3. Therefore, TLS_AES_128_CCM_8_SHA256 MUST NOT be used in DTLS without additional safeguards against forgery. Implementations MUST set usage limits for AEAD_AES_128_CCM_8 based on an understanding of any additional forgery protections that are used.

Any TLS cipher suite that is specified for use with DTLS MUST define limits on the use of the associated AEAD function that preserves margins for both confidentiality and integrity. That is, limits MUST be specified for the number of packets that can be authenticated and for the number of packets that can fail authentication before a key update is required. Providing a reference to any analysis upon which values are based – and any assumptions used in that analysis – allows limits to be adapted to varying usage conditions.

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 Section D.4 of [TLS13]. DTLS servers MUST NOT echo the "legacy_session_id" value from the client and endpoints MUST NOT send ChangeCipherSpec messages.

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 address that belongs to a victim. The server then sends its response to the victim machine, thus flooding it. Depending on the selected parameters this response message can be quite large, as 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 (see [TLS13]; Section 4.2.2). The client MUST send a new 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 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 reasons, 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 6. 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 6: 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 MUST be set to a zero-length vector (i.e., a zero-valued single byte length field).

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 content or hash of the initial ClientHello (and HelloRetryRequest) to be stored in the cookie. The initial ClientHello is included in the handshake transcript as a synthetic "message_hash" message, so only the hash value is needed for the handshake to complete, though the complete HelloRetryRequest contents are needed.

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 or, more generically, when the DTLS handshake uses a PSK-based key exchange and the IP address matches one associated with the PSK. Servers which process 0-RTT requests and send 0.5-RTT responses without a cookie exchange risk being used in an amplification attack if the size of outgoing messages greatly exceeds the size of those that are received. A server SHOULD limit the amount of data it sends toward a client address to three times the amount of data sent by the client before it verifies that the client is able to receive data at that address. A client address is valid after a cookie exchange or handshake completion. Clients MUST be prepared to do a cookie exchange with every handshake. Note that cookies are only valid for the existing handshake and cannot be stored for future handshakes.

If a server receives a ClientHello with an invalid cookie, it MUST terminate the handshake with an "illegal_parameter" alert. This allows the client to restart the connection from scratch without a cookie.

As described in Section 4.1.4 of [TLS13], clients MUST 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).

DTLS clients which do not want to receive a Connection ID SHOULD still offer the "connection_id" extension unless there is an application profile to the contrary. This permits a server which wants to receive a CID to negotiate one.

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 (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;
    } 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.

Note: In DTLS 1.2 the message_seq was reset to zero in case of a rehandshake (i.e., renegotiation). On the surface, a rehandshake in DTLS 1.2 shares similarities with a post-handshake message exchange in DTLS 1.3. However, in DTLS 1.3 the message_seq is not reset to allow distinguishing a retransmission from a previously sent post-handshake message from a newly sent post-handshake message.

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 implementations 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. The supported_versions entries for DTLS 1.0 and DTLS 1.2 are 0xfeff and 0xfefd (to match the wire versions). The value 0xfefc is used to indicate DTLS 1.3.

random: Same as for TLS 1.3, except that the downgrade sentinels described in Section 4.1.3 of [TLS13] when TLS 1.2 and TLS 1.1 and below are negotiated apply to DTLS 1.2 and DTLS 1.0 respectively.

legacy_session_id: Versions of TLS and DTLS before version 1.3 supported a "session resumption" feature which has been merged with pre-shared keys in version 1.3. A client which has a cached session ID set by a pre-DTLS 1.3 server SHOULD set this field to that value. Otherwise, it MUST be set as a zero-length vector (i.e., a zero-valued single byte length field).

legacy_cookie: A DTLS 1.3-only client MUST set the legacy_cookie field to zero length. If a DTLS 1.3 ClientHello is received with any other value in this field, the server MUST abort the handshake with an "illegal_parameter" alert.

cipher_suites: Same as for TLS 1.3; only suites with DTLS-OK=Y may be used.

legacy_compression_methods: Same as for TLS 1.3.

extensions: Same as for TLS 1.3.

5.4. ServerHello Message

The DTLS 1.3 ServerHello message is the same as the TLS 1.3 ServerHello message, except that the legacy_version field is set to 0xfefd, indicating DTLS 1.2.

5.5. Handshake Message Fragmentation and Reassembly

As described in Section 4.3 one or more handshake messages may be carried in a single datagram. However, handshake messages are potentially bigger than the size allowed by the underlying datagram transport. DTLS provides a mechanism for fragmenting a handshake message over a number of records, each of which can be transmitted in separate datagrams, thus avoiding IP fragmentation.

When transmitting the handshake message, the sender divides the message into a series of N contiguous data ranges. 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. Each handshake message fragment that is placed into a record MUST be delivered in a single UDP datagram.

When a DTLS implementation receives a handshake message fragment corresponding to the next expected handshake message sequence number, 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. Senders MUST NOT change handshake message bytes upon retransmission. Receivers MAY check that retransmitted bytes are identical and SHOULD abort the handshake with an "illegal_parameter" alert if the value of a byte 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 handshake messages into the same datagram: in the same record or in separate records.

5.6. 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 NOT accept records from epoch 1 indefinitely once they are able to process records from epoch 3. Though reordering of IP packets can result in records from epoch 1 arriving after records from epoch 3, this is not likely to persist for very long relative to the round trip time. Servers could discard epoch 1 keys after the first epoch 3 data arrives, or retain keys for processing epoch 1 data for a short period. (See Section 6.1 for the definitions of each epoch.)

5.7. DTLS Handshake Flights

DTLS handshake messages are grouped into a series of message flights. A flight starts with the handshake message transmission of one peer and ends with the expected response from the other peer. Table 1 contains a complete list of message combinations that constitute flights.

Note	Client	Server	Handshake Messages
	x		ClientHello
		x	HelloRetryRequest
		x	ServerHello, EncryptedExtensions, CertificateRequest, Certificate, CertificateVerify, Finished
1	x		Certificate, CertificateVerify, Finished
1		x	NewSessionTicket

Table 1: Flight Handshake Message Combinations.

Remarks:

- * Table 1 does not highlight any of the optional messages.
- * Regarding note (1): When a handshake flight is sent without any expected response, as it is the case with the client's final flight or with the NewSessionTicket message, the flight must be acknowledged with an ACK message.

Below are several example message exchange illustrating the flight concept. The notational conventions from [TLS13] are used.

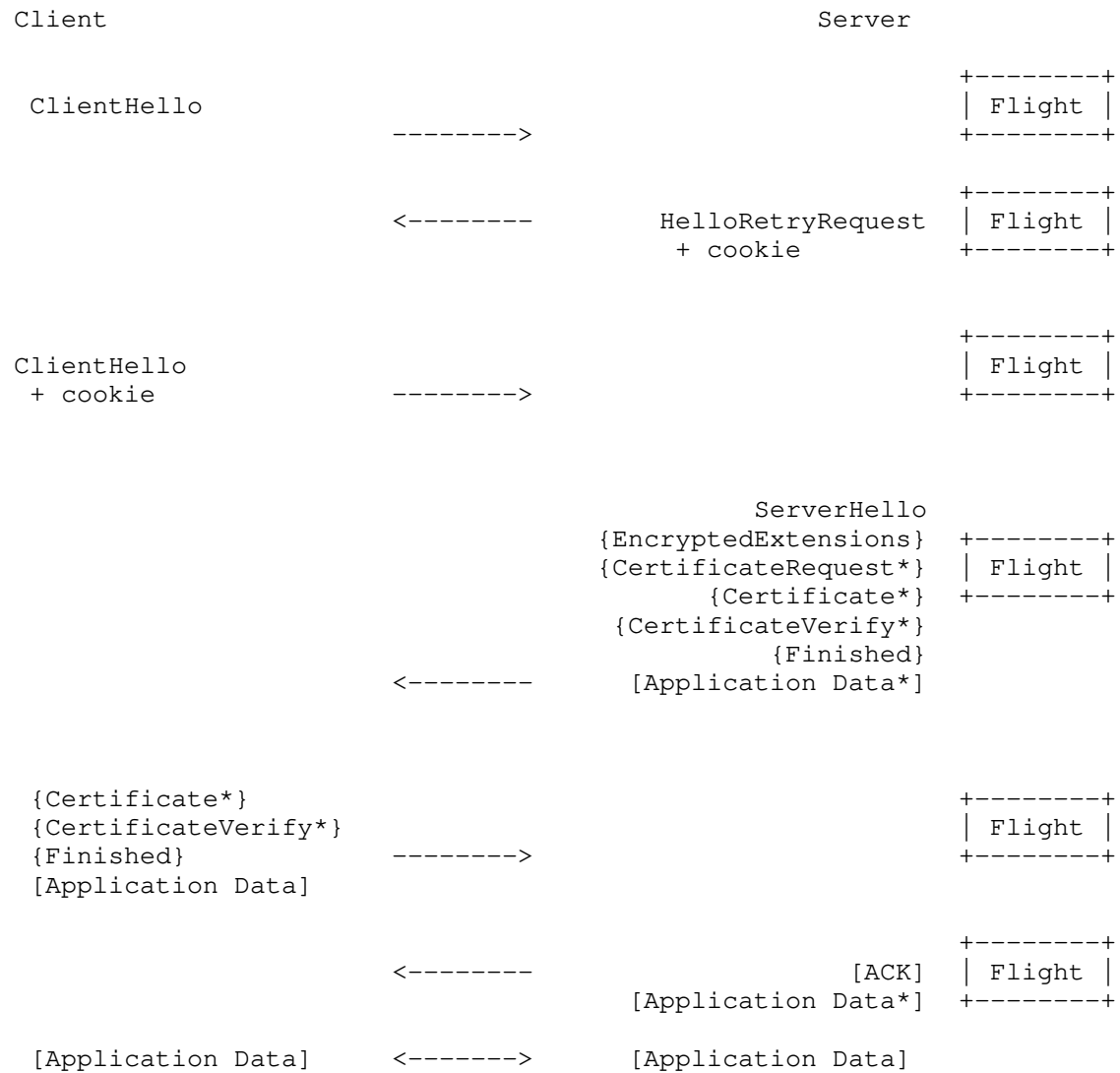


Figure 7: Message flights for a full DTLS Handshake (with cookie exchange)

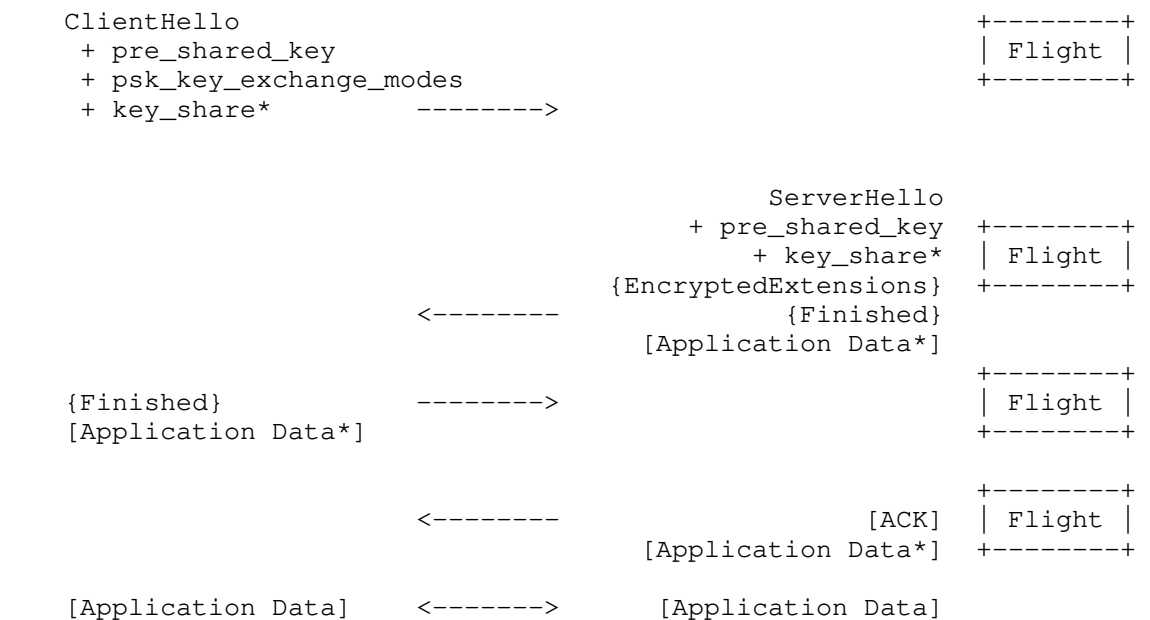


Figure 8: Message flights for resumption and PSK handshake (without cookie exchange)

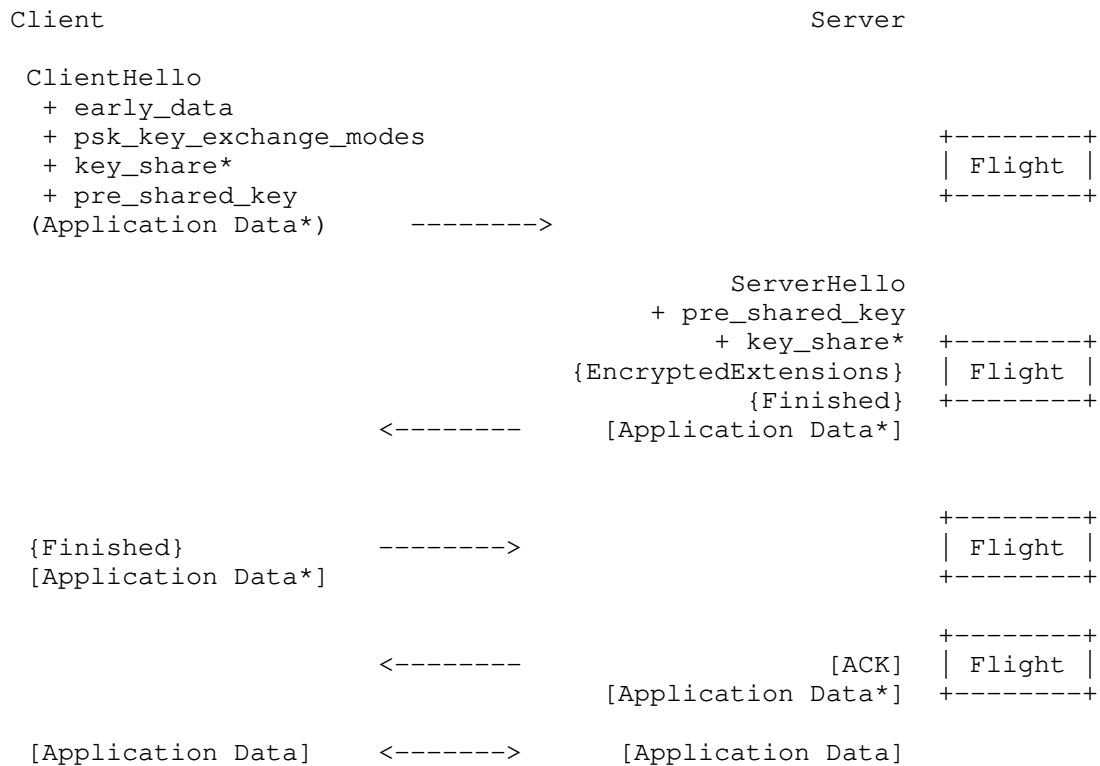
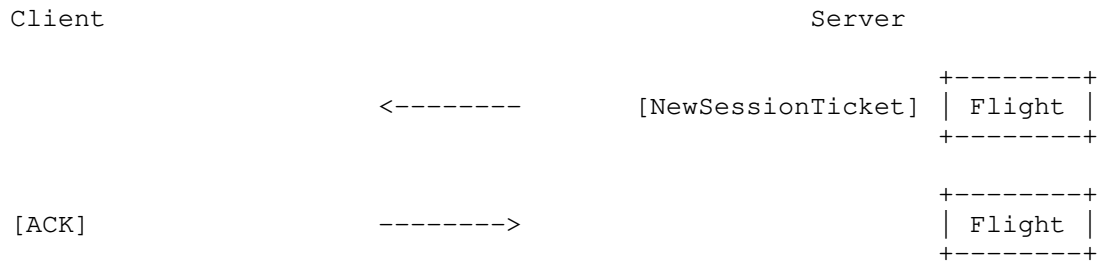


Figure 9: Message flights for the Zero-RTT handshake

Figure 10: Message flights for the `NewSessionTicket` message

`KeyUpdate`, `NewConnectionId` and `RequestConnectionId` follow a similar pattern to `NewSessionTicket`: a single message sent by one side followed by an `ACK` by the other.

5.8. Timeout and Retransmission

5.8.1. State Machine

DTLS uses a simple timeout and retransmission scheme with the state machine shown in Figure 11.

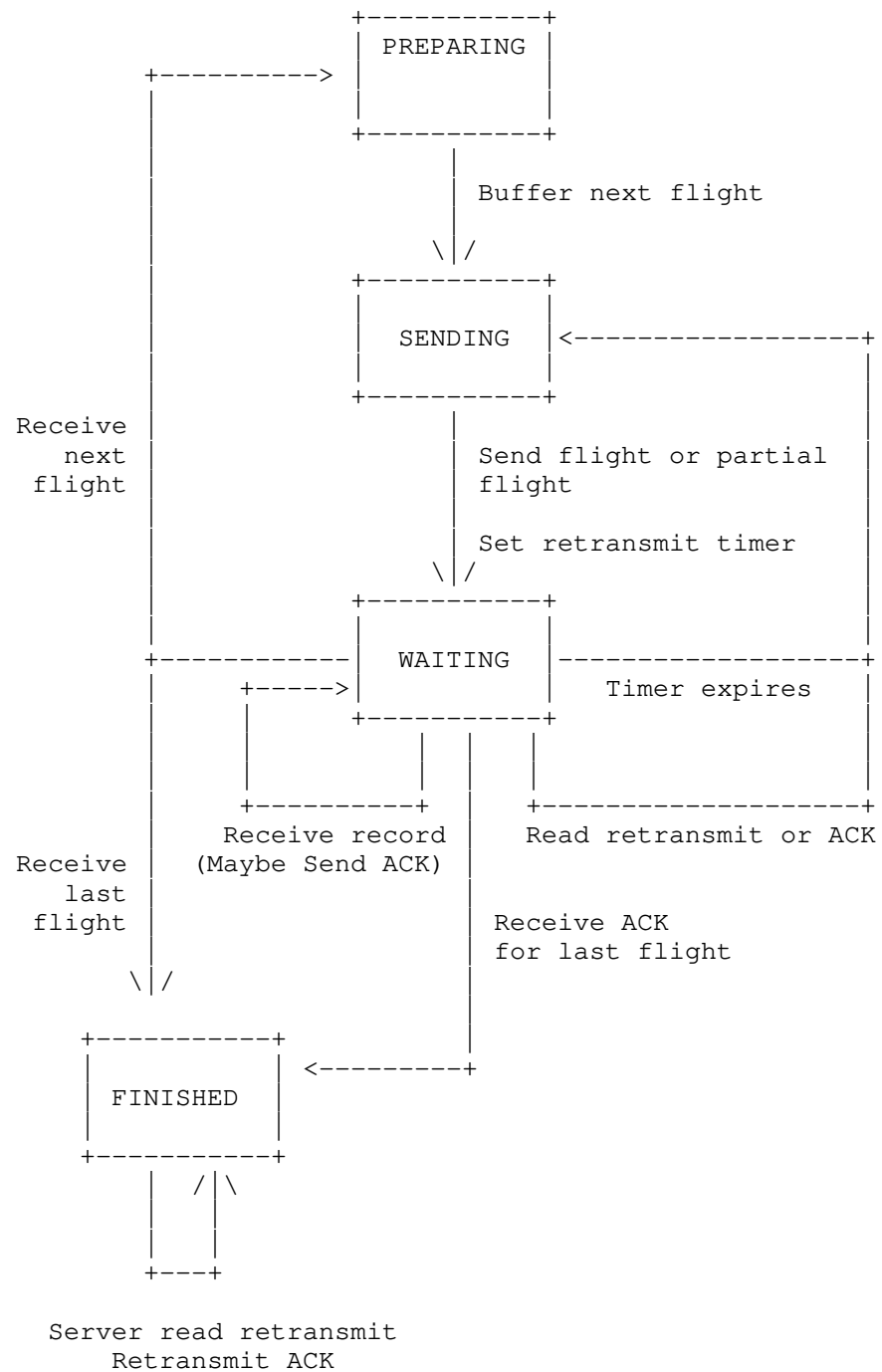


Figure 11: DTLS timeout and retransmission state machine

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 transmission 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 sets a retransmit timer and 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, adjusts and re-arms the retransmit timer (see Section 5.8.2), and returns to the WAITING state.
2. The implementation reads an 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, adjusts and re-arms 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, adjusts and re-arms 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 of the 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 MSL defined for [RFC0793], when in the FINISHED state, the server MUST respond to retransmission of the client's final 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 records for epoch 3 and above until they have received the Finished message from the peer. 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.

5.8.2. Timer Values

The configuration of timer settings varies with implementations, and certain deployment environments require timer value adjustments. 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.

Unless implementations have deployment-specific and/or external information about the round trip time, implementations SHOULD use an initial timer value of 1000 ms and double the value at each retransmission, up to no less than 60 seconds (the RFC 6298 [RFC6298] maximum). Application specific profiles MAY recommend shorter or longer timer values. For instance:

- * Profiles for specific deployment environments, such as in low-power, multi-hop mesh scenarios as used in some Internet of Things (IoT) networks, MAY specify longer timeouts. See [I-D.ietf-uta-tls13-iot-profile] for more information about one such DTLS 1.3 IoT profile.
- * Real-time protocols MAY specify shorter timeouts. It is RECOMMENDED that for DTLS-SRTP [RFC5764], a default timeout of 400ms be used; because customer experience degrades with one-way latencies of greater than 200ms, real-time deployments are less likely to have long latencies.

In settings where there is external information (for instance from an ICE [RFC8445] handshake, or from previous connections to the same server) about the RTT, implementations SHOULD use 1.5 times that RTT estimate as the retransmit timer.

Implementations SHOULD retain the current timer value until a message is transmitted and acknowledged without having to be retransmitted, at which time the value SHOULD be adjusted to 1.5 times the measured round trip time for that message. After a long period of idleness, no less than 10 times the current timer value, implementations MAY reset the timer to the initial value.

Note that because retransmission is for the handshake and not dataflow, the effect on congestion of shorter timeouts is smaller than in generic protocols such as TCP or QUIC. Experience with DTLS 1.2, which uses a simpler "retransmit everything on timeout" approach, has not shown serious congestion problems in practice.

5.8.3. Large Flight Sizes

DTLS does not have any built-in congestion control or rate control; in general this is not an issue because messages tend to be small. However, in principle, some messages - especially Certificate - can be quite large. If all the messages in a large flight are sent at once, this can result in network congestion. A better strategy is to send out only part of the flight, sending more when messages are acknowledged. Several extensions have been standardized to reduce the size of the certificate message, for example the cached information extension [RFC7924], certificate compression [RFC8879] and [RFC6066], which defines the "client_certificate_url" extension allowing DTLS clients to send a sequence of Uniform Resource Locators (URLs) instead of the client certificate.

DTLS stacks SHOULD NOT send more than 10 records in a single transmission.

5.8.4. State machine duplication for post-handshake messages

DTLS 1.3 makes use of the following categories of post-handshake messages:

1. NewSessionTicket
2. KeyUpdate
3. NewConnectionId
4. RequestConnectionId

5. Post-handshake client authentication

Messages of each category can be sent independently, and reliability is established via independent state machines each of which behaves as described in Section 5.8.1. For example, if a server sends a `NewSessionTicket` and a `CertificateRequest` message, two independent state machines will be created.

As explained in the corresponding sections, sending multiple instances of messages of a given category without having completed earlier transmissions is allowed for some categories, but not for others. Specifically, a server MAY send multiple `NewSessionTicket` messages at once without awaiting ACKs for earlier `NewSessionTicket` first. Likewise, a server MAY send multiple `CertificateRequest` messages at once without having completed earlier client authentication requests before. In contrast, implementations MUST NOT send `KeyUpdate`, `NewConnectionId` or `RequestConnectionId` messages if an earlier message of the same type has not yet been acknowledged.

Note: Except for post-handshake client authentication, which involves handshake messages in both directions, post-handshake messages are single-flight, and their respective state machines on the sender side reduce to waiting for an ACK and retransmitting the original message. In particular, note that a `RequestConnectionId` message does not force the receiver to send a `NewConnectionId` message in reply, and both messages are therefore treated independently.

Creating and correctly updating multiple state machines requires feedback from the handshake logic to the state machine layer, indicating which message belongs to which state machine. For example, if a server sends multiple `CertificateRequest` messages and receives a `Certificate` message in response, the corresponding state machine can only be determined after inspecting the `certificate_request_context` field. Similarly, a server sending a single `CertificateRequest` and receiving a `NewConnectionId` message in response can only decide that the `NewConnectionId` message should be treated through an independent state machine after inspecting the handshake message type.

5.9. 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.

5.10. Cryptographic Label Prefix

Section 7.1 of [TLS13] specifies that HKDF-Expand-Label uses a label prefix of "tls13 ". For DTLS 1.3, that label SHALL be "dtls13". This ensures key separation between DTLS 1.3 and TLS 1.3. Note that there is no trailing space; this is necessary in order to keep the overall label size inside of one hash iteration because "DTLS" is one letter longer than "TLS".

5.11. 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. Note that alerts are not reliably transmitted; implementation SHOULD NOT depend on receiving alerts in order to signal errors or connection closure.

5.12. 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 record 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 use CIDs to avoid the 5-tuple-based ambiguity.

6. Example of Handshake with Timeout and Retransmission

The following is an example of a handshake with lost packets and retransmissions. Note that the client sends an empty ACK message because it can only acknowledge Record 2 sent by the server once it has processed messages in Record 0 needed to establish epoch 2 keys, which are needed to encrypt or decrypt messages found in Record 2. Section 7 provides the necessary background details for this interaction. Note: for simplicity we are not re-setting record numbers in this diagram, so "Record 1" is really "Epoch 2, Record 0, etc.".

Client -----		Server -----
Record 0 ClientHello (message_seq=0)	----->	
	X<----- (lost)	Record 0 ServerHello (message_seq=0) Record 1 EncryptedExtensions (message_seq=1) Certificate (message_seq=2)
	<-----	Record 2 CertificateVerify (message_seq=3) Finished (message_seq=4)
Record 1 ACK []	----->	

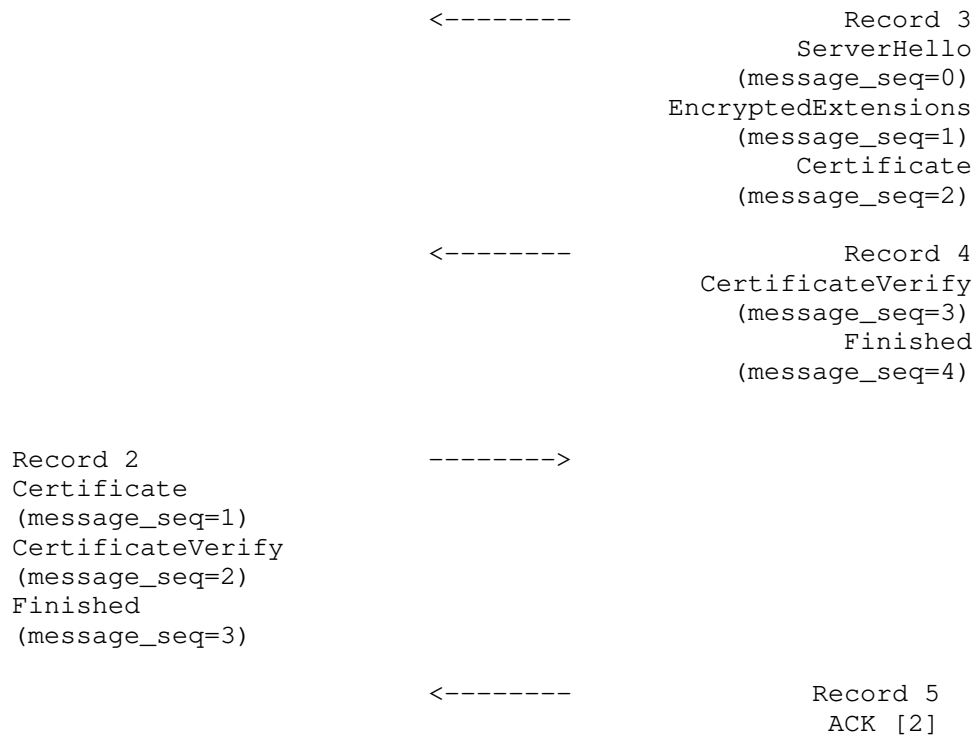


Figure 12: Example DTLS exchange illustrating message loss

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-ordering, 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 TLS 1.3-defined key derivation steps, see Section 7 of [TLS13], a sender may want to rekey at any time during the lifetime of the connection. It therefore needs 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.

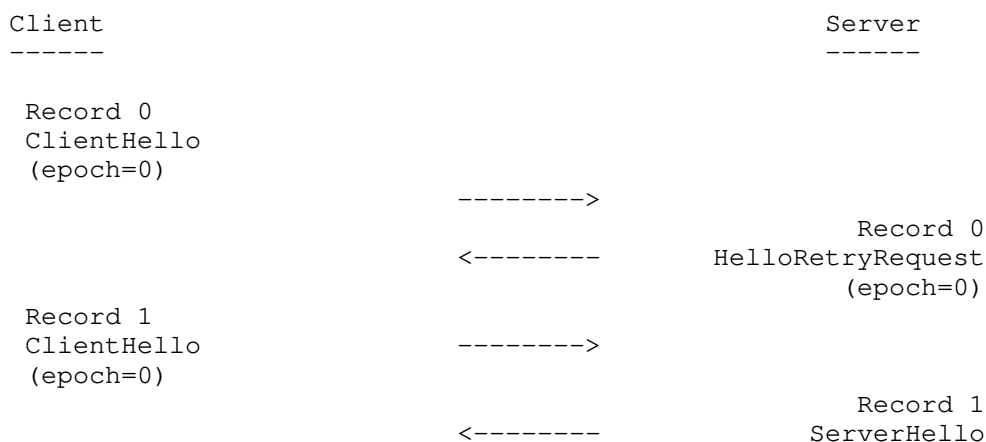
- * epoch value (1) is used for messages protected using keys derived from `client_early_traffic_secret`. Note this epoch is skipped if the client does not offer early data.
- * epoch value (2) is used for messages protected using keys derived from `[sender]_handshake_traffic_secret`. Messages transmitted 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 `[sender]_application_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 `[sender]_application_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 record with an epoch value for which no corresponding cipher state can be determined SHOULD handle it as a record which fails deprotection.

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 13 illustrates the epoch values in an example DTLS handshake.



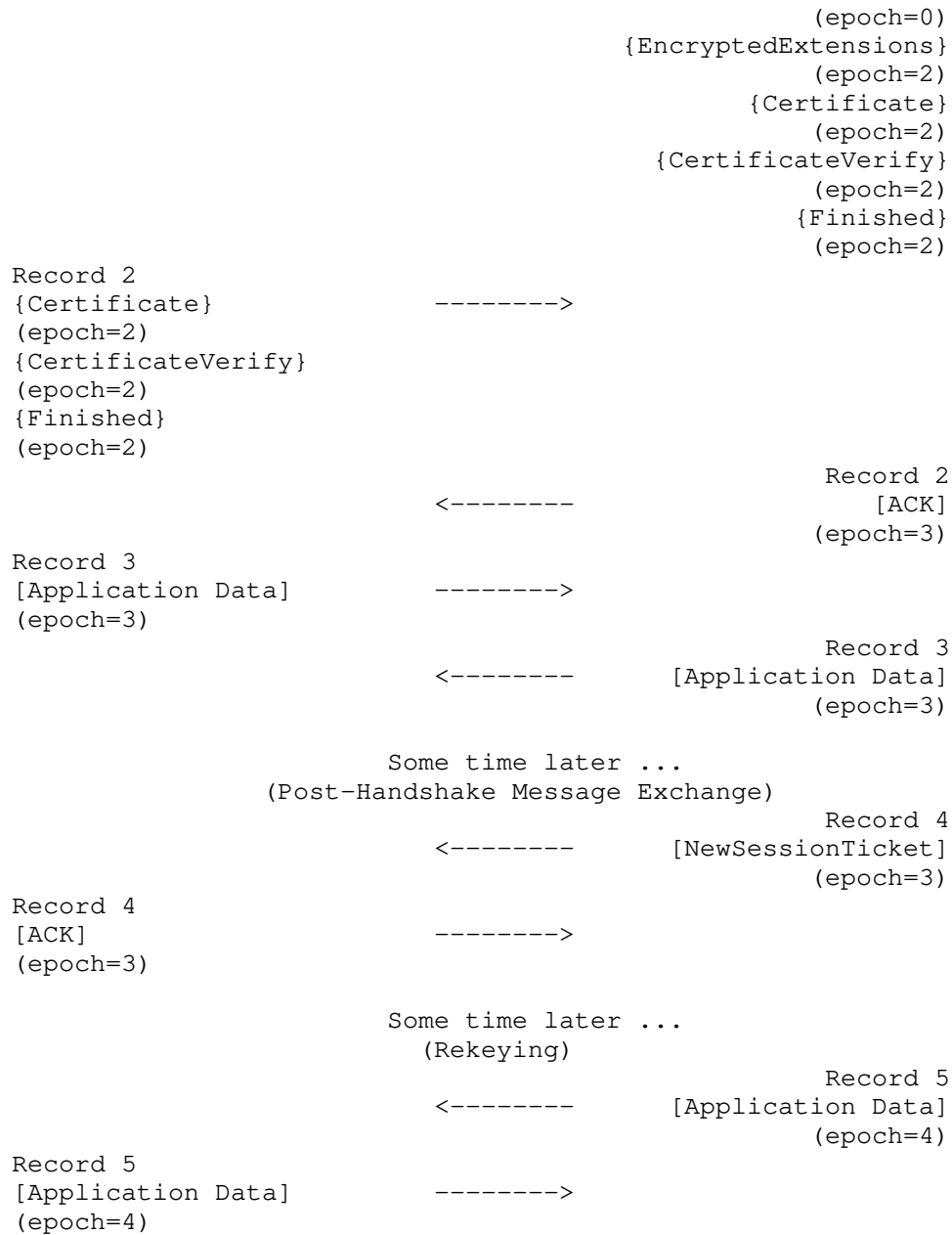


Figure 13: Example DTLS exchange with epoch information

7. ACK Message

The ACK message is used by an endpoint to indicate which handshake records it has received and processed 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 having ACK being added to the handshake transcript. Note that ACKs can still be sent in the same UDP datagram as handshake records.

```
struct {  
    RecordNumber 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 and either processed or buffered, in numerically increasing order.

Implementations **MUST NOT** acknowledge records containing handshake messages or fragments which have not been processed or buffered. Otherwise, deadlock can ensue. As an example, implementations **MUST NOT** send ACKs for handshake messages which they discard because they are not the next expected message.

During the handshake, ACKs only cover the current outstanding flight (this is possible because DTLS is generally a lockstep protocol). In particular, receiving a message from a handshake flight implicitly acknowledges all messages from the previous flight(s). Accordingly, an ACK from the server would not cover both the ClientHello and the client's Certificate, because the ClientHello and client Certificate are in different flights. Implementations can accomplish this by clearing their ACK list upon receiving the start of the next flight.

After the handshake, ACKs **SHOULD** be sent once for each received and processed handshake record (potentially subject to some delay) and **MAY** cover more than one flight. This includes records containing messages which are discarded because a previous copy has been received.

During the handshake, ACK records **MUST** be sent with an epoch that is equal to or higher than the record which is being acknowledged. Note that some care is required when processing flights spanning multiple epochs. For instance, if the client receives only the Server Hello and Certificate and wishes to ACK them in a single record, it must do so in epoch 2, as it is required to use an epoch greater than or equal to 2 and cannot yet send with any greater epoch. Implementations **SHOULD** simply use the highest current sending epoch, which will generally be the highest available. After the handshake, implementations **MUST** use the highest available sending epoch.

7.1. Sending ACKs

When an implementation detects a disruption in the receipt of the current incoming flight, it SHOULD generate an ACK that covers the messages from that flight which it has received and processed so far. Implementations have some discretion about which events to treat as signs of disruption, but it is RECOMMENDED that they generate ACKs 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.
- * 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). "Immediately" is hard to define. One approach 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. Note: the 1/4 value here is somewhat arbitrary. Given that the round trip estimates in the DTLS handshake are generally very rough (or the default), any value will be an approximation, and there is an inherent compromise due to competition between retransmission due to over-aggressive ACKing and over-aggressive timeout-based retransmission. As a comparison point, QUIC's loss-based recovery algorithms ([I-D.ietf-quic-recovery]; Section 6.1.2) work out to a delay of about 1/3 of the retransmit timer.

In general, flights MUST be ACKed unless they are implicitly acknowledged. In the present specification the following flights are implicitly acknowledged by the receipt of the next flight, which generally immediately follows the flight,

1. Handshake flights other than the client's final flight of the main handshake.
2. The server's post-handshake CertificateRequest.

ACKs SHOULD NOT be sent for these flights unless the responding flight cannot be generated immediately. In this case, implementations MAY send explicit ACKs for the complete received flight even though it will eventually also be implicitly acknowledged through the responding flight. A notable example for this is the case of client authentication in constrained environments, where generating the CertificateVerify message can take considerable time on the client. All other flights MUST be ACKed. Implementations MAY acknowledge the records corresponding to each transmission of each flight or simply acknowledge the most recent one. In general,

implementations SHOULD ACK as many received packets as can fit into the ACK record, as this provides the most complete information and thus reduces the chance of spurious retransmission; if space is limited, implementations SHOULD favor including records which have not yet been acknowledged.

Note: While some post-handshake messages follow a request/response pattern, this does not necessarily imply receipt. For example, a KeyUpdate sent in response to a KeyUpdate with request_update set to 'update_requested' does not implicitly acknowledge the earlier KeyUpdate message because the two KeyUpdate messages might have crossed in flight.

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 acknowledge 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 that leaves it with only some messages from a flight having been acknowledged an implementation SHOULD retransmit the unacknowledged 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. Implementations MUST treat a record as having been acknowledged if it appears in any ACK; this prevents spurious retransmission in cases where a flight is very large and the receiver is forced to elide acknowledgements for records which have already been ACKed. As noted above, the receipt of any record responding to a given flight MUST be taken as an implicit acknowledgement for the entire flight to which it is responding.

7.3. Design Rationale

ACK messages are used in two circumstances, namely :

- * on sign of disruption, or lack of progress, and
- * to indicate complete receipt of the last flight in a handshake.

In the first case the use of the ACK message is optional because the peer will retransmit in any case and therefore the ACK just allows for selective or early retransmission, as opposed to the timeout-based whole flight retransmission in previous versions of DTLS. When DTLS 1.3 is used in deployments with lossy networks, such as low-power, long range radio networks as well as low-power mesh networks, the use of ACKs is recommended.

The use of the ACK for the second case is mandatory for the proper functioning of the protocol. For instance, the ACK message sent by the client in Figure 13, acknowledges receipt and processing of record 4 (containing the NewSessionTicket message) and if it is not sent the server will continue retransmission of the NewSessionTicket indefinitely until its maximum retransmission count is reached.

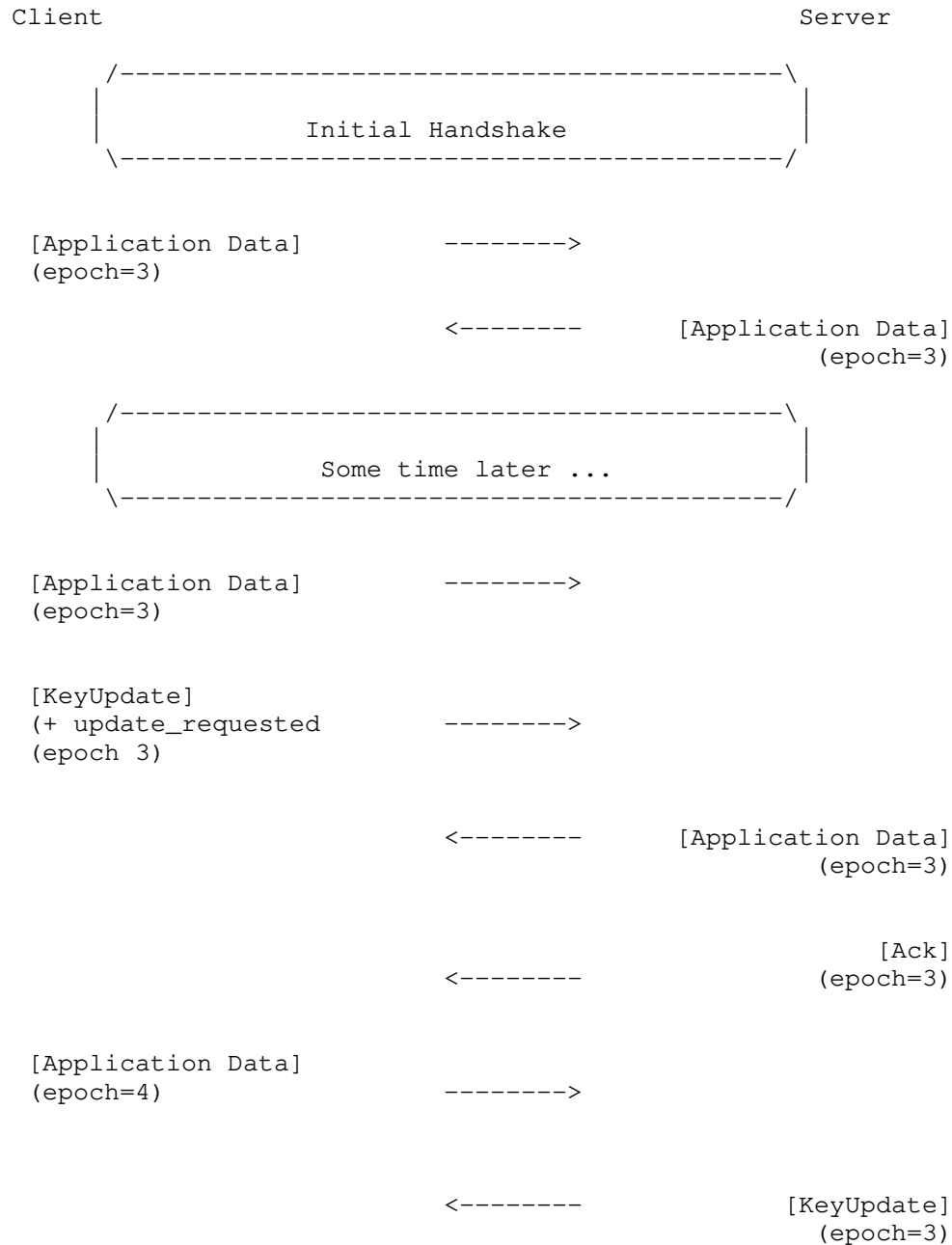
8. Key Updates

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 records 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.

Due to the possibility of an ACK message for a KeyUpdate being lost and thereby preventing the sender of the KeyUpdate 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.

Figure 14 shows an example exchange illustrating that a successful ACK processing updates the keys of the KeyUpdate message sender, which is reflected in the change of epoch values.



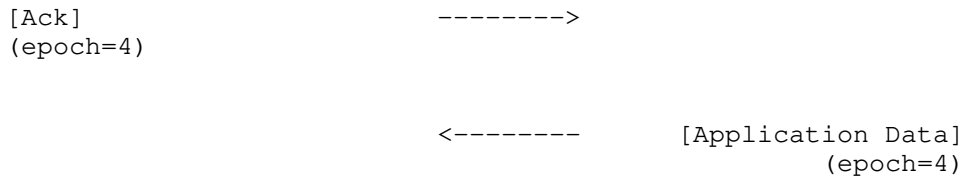


Figure 14: Example DTLS Key Update

9. Connection ID Updates

If the client and server have negotiated the "connection_id" extension [I-D.ietf-tls-dtls-connection-id], either side can send a new CID 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. Implementations which receive more spare CIDs than they wish to maintain MAY simply discard any extra CIDs. Endpoints MUST NOT have more than one NewConnectionId message outstanding.

Implementations which either did not negotiate the "connection_id" extension or which have negotiated receiving an empty CID MUST NOT send NewConnectionId. Implementations MUST NOT send RequestConnectionId when sending an empty Connection ID. Implementations which detect a violation of these rules MUST terminate the connection with an "unexpected_message" alert.

Implementations SHOULD use a new CID whenever sending on a new path, and SHOULD request new CIDs for this purpose if path changes are anticipated.

```
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 handle requests, which it considers excessive, by responding with a NewConnectionId message containing fewer than num_cid CIDs, including no CIDs at all. Endpoints MAY handle an excessive number of RequestConnectionId messages by terminating the connection using a "too_many_cids_requested" (alert number 52) alert.

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

9.1. Connection ID Example

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

Note: The connection_id extension is defined in [I-D.ietf-tls-dtls-connection-id], which is used in ClientHello and ServerHello messages.

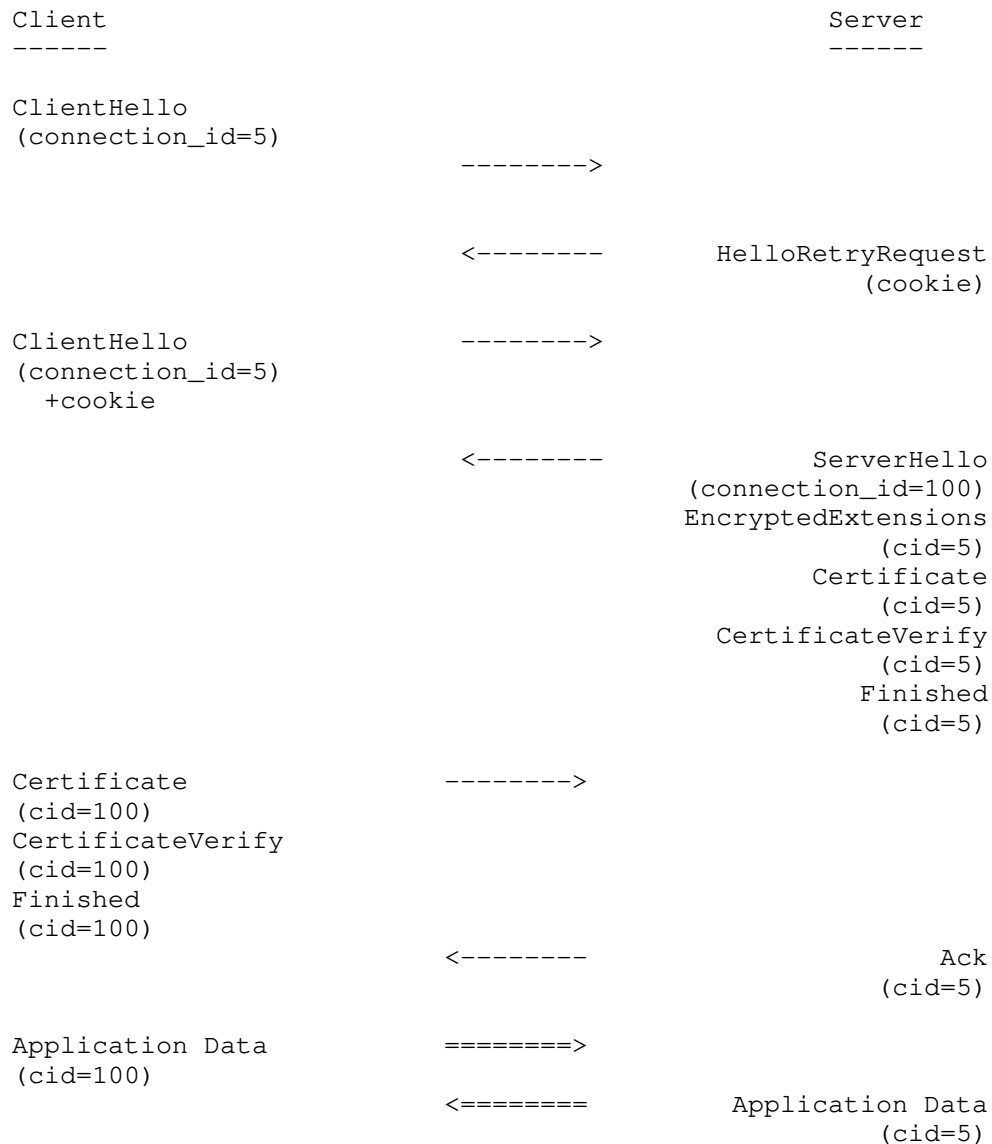


Figure 15: Example DTLS 1.3 Exchange with CIDs

If no CID is negotiated, then the receiver MUST reject any records it receives that contain a CID.

10. Application Data Protocol

Application data messages are carried by the record layer and are split into records 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 by excessive resource consumption. 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.

Some key properties required of the cookie for the cookie-exchange mechanism to be functional are described in Section 3.3 of [RFC2522]:

- * the cookie **MUST** depend on the client's address.
- * it **MUST NOT** be possible for anyone other than the issuing entity to generate cookies that are accepted as valid by that entity. This typically entails an integrity check based on a secret key.
- * cookie generation and verification are triggered by unauthenticated parties, and as such their resource consumption needs to be restrained in order to avoid having the cookie-exchange mechanism itself serve as a DoS vector.

Although the cookie must allow the server to produce the right handshake transcript, it **SHOULD** be constructed so that knowledge of the cookie is insufficient to reproduce the ClientHello contents. Otherwise, this may create problems with future extensions such as [I-D.ietf-tls-esni].

When cookies are generated using a keyed authentication mechanism it should be possible to rotate the associated secret key, so that temporary compromise of the key does not permanently compromise the integrity of the cookie-exchange mechanism. Though this secret is not as high-value as, e.g., a session-ticket-encryption key, rotating

the cookie-generation key on a similar timescale would ensure that the key-rotation functionality is exercised regularly and thus in working order.

The cookie exchange provides address validation during the initial handshake. DTLS with Connection IDs allows for endpoint addresses to change during the association; any such updated addresses are not covered by the cookie exchange during the handshake. DTLS implementations **MUST NOT** update the address they send to in response to packets from a different address unless they first perform some reachability test; no such test is defined in this specification. Even with such a test, an active on-path adversary can also black-hole traffic or create a reflection attack against third parties because a DTLS peer has no means to distinguish a genuine address update event (for example, due to a NAT rebinding) from one that is malicious. This attack is of concern when there is a large asymmetry of request/response message sizes.

With the exception of order protection and non-replayability, the security guarantees for DTLS 1.3 are the same as TLS 1.3. While TLS always provides order protection and non-replayability, DTLS does not provide order protection and may not provide replay protection.

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

TLS 1.3 requires replay protection for 0-RTT data (or rather, for connections that use 0-RTT data; see Section 8 of [TLS13]). DTLS provides an optional per-record replay-protection mechanism, since datagram protocols are inherently subject to message reordering and replay. These two replay-protection mechanisms are orthogonal, and neither mechanism meets the requirements for the other.

The security and privacy properties of the CID for DTLS 1.3 builds on top of what is described for DTLS 1.2 in [I-D.ietf-tls-dtls-connection-id]. There are, however, several differences:

- * In both versions of DTLS extension negotiation is used to agree on the use of the CID feature and the CID values. In both versions the CID is carried in the DTLS record header (if negotiated). However, the way the CID is included in the record header differs between the two versions.
- * The use of the Post-Handshake message allows the client and the server to update their CIDs and those values are exchanged with confidentiality protection.

- * The ability to use multiple CIDs allows for improved privacy properties in multi-homed scenarios. When only a single CID is in use on multiple paths from such a host, an adversary can correlate the communication interaction across paths, which adds further privacy concerns. In order to prevent this, implementations SHOULD attempt to use fresh CIDs whenever they change local addresses or ports (though this is not always possible to detect). The RequestConnectionId message can be used by a peer to ask for new CIDs to ensure that a pool of suitable CIDs is available.
- * The mechanism for encrypting sequence numbers (Section 4.2.3) prevents trivial tracking by on-path adversaries that attempt to correlate the pattern of sequence numbers received on different paths; such tracking could occur even when different CIDs are used on each path, in the absence of sequence number encryption. Switching CIDs based on certain events, or even regularly, helps against tracking by on-path adversaries. Note that sequence number encryption is used for all encrypted DTLS 1.3 records irrespective of whether a CID is used or not. Unlike the sequence number, the epoch is not encrypted because it acts as a key identifier, which may improve correlation of packets from a single connection across different network paths.
- * DTLS 1.3 encrypts handshake messages much earlier than in previous DTLS versions. Therefore, less information identifying the DTLS client, such as the client certificate, is available to an on-path adversary.

12. Changes since DTLS 1.2

Since TLS 1.3 introduces a large number of changes with respect 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
- * Only AEAD ciphers are supported. Additional data calculation has been simplified.
- * Removed support for weaker and older cryptographic algorithms
- * 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
- * Improved version negotiation
- * Optimized record layer encoding and thereby its size
- * Added CID functionality
- * Sequence numbers are encrypted.

13. Updates affecting DTLS 1.2

This document defines several changes that optionally affect implementations of DTLS 1.2, including those which do not also support DTLS 1.3.

- * A version downgrade protection mechanism as described in [TLS13]; Section 4.1.3 and applying to DTLS as described in Section 5.3.
- * The updates described in [TLS13]; Section 3.
- * The new compliance requirements described in [TLS13]; Section 9.3.

14. 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 26. The value for the "DTLS-OK" column is "Y". 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 "the too_many_cids_requested" alert in the "TLS Alerts" registry with value 52.

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. The value for the "DTLS-OK" columns are "Y".

IANA is requested to add this RFC as a reference to the TLS Cipher Suite Registry along with the following Note:

Any TLS cipher suite that is specified for use with DTLS MUST define limits on the use of the associated AEAD function that preserves margins for both confidentiality and integrity, as specified in [THIS RFC; Section TODO]

15. References

15.1. Normative References

- [CHACHA] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <<https://www.rfc-editor.org/info/rfc8439>>.
- [I-D.ietf-tls-dtls-connection-id] Rescorla, E., Tschofenig, H., Fossati, T., and A. Kraus, "Connection Identifiers for DTLS 1.2", Work in Progress, Internet-Draft, draft-ietf-tls-dtls-connection-id-11, 14 April 2021, <<https://www.ietf.org/internet-drafts/draft-ietf-tls-dtls-connection-id-11.txt>>.
- [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>>.

15.2. Informative References

- [AEBounds] Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", 8 March 2016, <<http://www.isg.rhul.ac.uk/~kp/TLS-AEBounds.pdf>>.
- [CCM-ANALYSIS] Jonsson, J., "On the Security of CTR + CBC-MAC", Selected Areas in Cryptography pp. 76-93, DOI 10.1007/3-540-36492-7_7, 2003, <https://doi.org/10.1007/3-540-36492-7_7>.
- [DEPRECATE] Moriarty, K. and S. Farrell, "Deprecating TLSv1.0 and TLSv1.1", Work in Progress, Internet-Draft, draft-ietf-tls-oldversions-deprecate-12, 21 January 2021, <<http://www.ietf.org/internet-drafts/draft-ietf-tls-oldversions-deprecate-12.txt>>.
- [I-D.ietf-quic-recovery] Iyengar, J. and I. Swett, "QUIC Loss Detection and Congestion Control", Work in Progress, Internet-Draft, draft-ietf-quic-recovery-34, 14 January 2021, <<https://www.ietf.org/internet-drafts/draft-ietf-quic-recovery-34.txt>>.
- [I-D.ietf-tls-esni] Rescorla, E., Oku, K., Sullivan, N., and C. Wood, "TLS Encrypted Client Hello", Work in Progress, Internet-Draft, draft-ietf-tls-esni-10, 8 March 2021, <<https://www.ietf.org/internet-drafts/draft-ietf-tls-esni-10.txt>>.

- [I-D.ietf-uta-tls13-iot-profile]
Tschofenig, H. and T. Fossati, "TLS/DTLS 1.3 Profiles for the Internet of Things", Work in Progress, Internet-Draft, draft-ietf-uta-tls13-iot-profile-01, 22 February 2021, <<https://www.ietf.org/internet-drafts/draft-ietf-uta-tls13-iot-profile-01.txt>>.
- [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>>.
- [RFC5763] Fischl, J., Tschofenig, H., and E. Rescorla, "Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS)", RFC 5763, DOI 10.17487/RFC5763, May 2010, <<https://www.rfc-editor.org/info/rfc5763>>.

- [RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, DOI 10.17487/RFC5764, May 2010, <<https://www.rfc-editor.org/info/rfc5764>>.
- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <<https://www.rfc-editor.org/info/rfc6066>>.
- [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>>.
- [RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <<https://www.rfc-editor.org/info/rfc7924>>.
- [RFC7983] Petit-Huguenin, M. and G. Salgueiro, "Multiplexing Scheme Updates for Secure Real-time Transport Protocol (SRTP) Extension for Datagram Transport Layer Security (DTLS)", RFC 7983, DOI 10.17487/RFC7983, September 2016, <<https://www.rfc-editor.org/info/rfc7983>>.
- [RFC8201] McCann, J., Deering, S., Mogul, J., and R. Hinden, Ed., "Path MTU Discovery for IP version 6", STD 87, RFC 8201, DOI 10.17487/RFC8201, July 2017, <<https://www.rfc-editor.org/info/rfc8201>>.
- [RFC8445] Keranen, A., Holmberg, C., and J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal", RFC 8445, DOI 10.17487/RFC8445, July 2018, <<https://www.rfc-editor.org/info/rfc8445>>.

- [RFC8879] Ghedini, A. and V. Vasiliev, "TLS Certificate Compression", RFC 8879, DOI 10.17487/RFC8879, December 2020, <<https://www.rfc-editor.org/info/rfc8879>>.
- [ROBUST] Fischlin, M., Günther, F., and C. Janson, "Robust Channels: Handling Unreliable Networks in the Record Layers of QUIC and DTLS 1.3", 15 June 2020, <<https://eprint.iacr.org/2020/718>>.

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;
    uint48 sequence_number;
    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 |
| (if any,      |
| / length as   /
| negotiated)   |
+---+---+---+---+---+---+
| 8 or 16 bit   |
| Sequence Number|
+---+---+---+---+---+---+
| 16 bit Length |
| (if present)  |
+---+---+---+---+---+---+

```

Legend:

C - Connection ID (CID) present
 S - Sequence number length
 L - Length present
 E - Epoch

```

struct {
    uint16 epoch;
    uint48 sequence_number;
} RecordNumber;

```

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),
    certificate_url_RESERVED(21),
    certificate_status_RESERVED(22),
    supplemental_data_RESERVED(23),
    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 (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;
    } 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 {
    RecordNumber 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. Analysis of Limits on CCM Usage

TLS [TLS13] and [AEBounds] do not specify limits on key usage for AEAD_AES_128_CCM. However, any AEAD that is used with DTLS requires limits on use that ensure that both confidentiality and integrity are preserved. This section documents that analysis for AEAD_AES_128_CCM.

[CCM-ANALYSIS] is used as the basis of this analysis. The results of that analysis are used to derive usage limits that are based on those chosen in [TLS13].

This analysis uses symbols for multiplication (*), division (/), and exponentiation (^), plus parentheses for establishing precedence. The following symbols are also used:

- t: The size of the authentication tag in bits. For this cipher, t is 128.
- n: The size of the block function in bits. For this cipher, n is 128.
- l: The number of blocks in each packet (see below).
- q: The number of genuine packets created and protected by endpoints. This value is the bound on the number of packets that can be protected before updating keys.
- v: The number of forged packets that endpoints will accept. This value is the bound on the number of forged packets that an endpoint can reject before updating keys.

The analysis of AEAD_AES_128_CCM relies on a count of the number of block operations involved in producing each message. For simplicity, and to match the analysis of other AEAD functions in [AEBounds], this analysis assumes a packet length of 2^{10} blocks and a packet size limit of 2^{14} bytes.

For AEAD_AES_128_CCM, the total number of block cipher operations is the sum of: the length of the associated data in blocks, the length of the ciphertext in blocks, and the length of the plaintext in blocks, plus 1. In this analysis, this is simplified to a value of twice the maximum length of a record in blocks (that is, " $2l = 2^{11}$ "). This simplification is based on the associated data being limited to one block.

B.1. Confidentiality Limits

For confidentiality, Theorem 2 in [CCM-ANALYSIS] establishes that an attacker gains a distinguishing advantage over an ideal pseudorandom permutation (PRP) of no more than:

$$(2l * q)^2 / 2^n$$

For a target advantage of 2^{-60} , which matches that used by [TLS13], this results in the relation:

$$q \leq 2^{23}$$

That is, endpoints cannot protect more than 2^{23} packets with the same set of keys without causing an attacker to gain an larger advantage than the target of 2^{-60} .

B.2. Integrity Limits

For integrity, Theorem 1 in [CCM-ANALYSIS] establishes that an attacker gains an advantage over an ideal PRP of no more than:

$$v / 2^t + (2l * (v + q))^2 / 2^n$$

The goal is to limit this advantage to 2^{-57} , to match the target in [TLS13]. As "t" and "n" are both 128, the first term is negligible relative to the second, so that term can be removed without a significant effect on the result. This produces the relation:

$$v + q \leq 2^{24.5}$$

Using the previously-established value of 2^{23} for "q" and rounding, this leads to an upper limit on "v" of $2^{23.5}$. That is, endpoints cannot attempt to authenticate more than $2^{23.5}$ packets with the same set of keys without causing an attacker to gain an larger advantage than the target of 2^{-57} .

B.3. Limits for AEAD_AES_128_CCM_8

The TLS_AES_128_CCM_8_SHA256 cipher suite uses the AEAD_AES_128_CCM_8 function, which uses a short authentication tag (that is, $t=64$).

The confidentiality limits of AEAD_AES_128_CCM_8 are the same as those for AEAD_AES_128_CCM, as this does not depend on the tag length; see Appendix B.1.

The shorter tag length of 64 bits means that the simplification used in Appendix B.2 does not apply to AEAD_AES_128_CCM_8. If the goal is to preserve the same margins as other cipher suites, then the limit on forgeries is largely dictated by the first term of the advantage formula:

$$v \leq 2^7$$

As this represents attempts to fail authentication, applying this limit might be feasible in some environments. However, applying this limit in an implementation intended for general use exposes connections to an inexpensive denial of service attack.

This analysis supports the view that TLS_AES_128_CCM_8_SHA256 is not suitable for general use. Specifically, TLS_AES_128_CCM_8_SHA256 cannot be used without additional measures to prevent forgery of records, or to mitigate the effect of forgeries. This might require understanding the constraints that exist in a particular deployment or application. For instance, it might be possible to set a different target for the advantage an attacker gains based on an understanding of the constraints imposed on a specific usage of DTLS.

Appendix C. Implementation Pitfalls

In addition to the aspects of TLS that have been a source of interoperability and security problems (Section C.3 of [TLS13]), DTLS presents a few new potential sources of issues, noted here.

- * Do you correctly handle messages received from multiple epochs during a key transition? This includes locating the correct key as well as performing replay detection, if enabled.
- * Do you retransmit handshake messages that are not (implicitly or explicitly) acknowledged (Section 5.8)?
- * Do you correctly handle handshake message fragments received, including when they are out of order?
- * Do you correctly handle handshake messages received out of order? This may include either buffering or discarding them.
- * Do you limit how much data you send to a peer before its address is validated?
- * Do you verify that the explicit record length is contained within the datagram in which it is contained?

Appendix D. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

(*) indicates a change that may affect interoperability.

IETF Drafts draft-42

- * SHOULD level requirement for the client to offer CID extension.
- * Change the default retransmission timer to 1s and allow people to do otherwise if they have side knowledge.
- * Cap any given flight to 10 records

- * Don't re-set the timer to the initial value but to 1.5 times the measured RTT.
- * A bunch more clarity about the reliability algorithms and timers (including changing reset to re-arm)
- * Update IANA considerations

draft-40

- Clarified encrypted_record structure in DTLS 1.3 record layer
- Added description of the demultiplexing process
- Added text about the DTLS 1.2 and DTLS 1.3 CID mechanism
- Forbid going from an empty CID to a non-empty CID (*)
- Add warning about certificates and congestion
- Use DTLS style version values, even for DTLS 1.3 (*)
- Describe how to distinguish DTLS 1.2 and DTLS 1.3 connections
- Updated examples
- Included editorial improvements from Ben Kaduk
- Removed stale text about out-of-epoch records
- Added clarifications around when ACKs are sent
- Noted that alerts are unreliable
- Clarify when you can reset the timer
- Indicated that records with bogus epochs should be discarded
- Relax age out text
- Updates to cookie text
- Require that cipher suites define a record number encryption algorithm
- Clean up use of connection and association
- Reference tls-old-versions-deprecate

draft-39 - Updated Figure 4 due to misalignment with Figure 3 content

draft-38 - Ban implicit Connection IDs (*) - ACKs are processed as the union.

draft-37: - Fix the other place where we have ACK.

draft-36: - Some editorial changes. - Changed the content type to not conflict with existing allocations (*)

draft-35: - I-D.ietf-tls-dtls-connection-id became a normative reference - Removed duplicate reference to I-D.ietf-tls-dtls-connection-id. - Fix figure 11 to have the right numbers and no cookie in message 1. - Clarify when you can ACK. - Clarify additional data computation.

draft-33: - Key separation between TLS and DTLS. Issue #72.

draft-32: - Editorial improvements and clarifications.

draft-31: - Editorial improvements in text and figures. - Added normative reference to ChaCha20 and Poly1305.

draft-30: - Changed record format - Added text about end of early data - Changed format of the Connection ID Update message - Added Appendix A "Protocol Data Structures and Constant Values"

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 CIDs.

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 records 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 E. Working Group Information

RFC EDITOR: PLEASE REMOVE THIS SECTION.

The discussion list for the IETF TLS working group is located at the e-mail address tls@ietf.org (<mailto:tls@ietf.org>). Information on the group and information on how to subscribe to the list is at <https://www1.ietf.org/mailman/listinfo/tls> (<https://www1.ietf.org/mailman/listinfo/tls>)

Archives of the list can be found at: <https://www.ietf.org/mail-archive/web/tls/current/index.html> (<https://www.ietf.org/mail-archive/web/tls/current/index.html>)

Appendix F. 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. Felix Guenther and Martin Thomson contributed the analysis in Appendix B.

In addition, we would like to thank:

- * David Benjamin
Google
davidben@google.com
- * Thomas Fossati
Arm Limited
Thomas.Fossati@arm.com
- * Tobias Gondrom
Huawei
tobias.gondrom@gondrom.org
- * Felix Günther
ETH Zurich
mail@felixguenther.info
- * Benjamin Kaduk
Akamai Technologies
kaduk@mit.edu
- * Ilari Liusvaara
Independent
ilariliusvaara@welho.com

- * Martin Thomson
Mozilla
martin.thomson@gmail.com
- * Christopher A. Wood
Apple Inc.
cawood@apple.com
- * Yin Xinxing
Huawei
yinxinxing@huawei.com
- * Hanno Becker
Arm Limited
Hanno.Becker@arm.com

Appendix G. Acknowledgements

We would like to thank Jonathan Hammell, Bernard Aboba and Andy Cunningham for their review comments.

Additionally, we would like to thank the IESG members for their review comments: Martin Duke, Erik Kline, Francesca Palombini, Lars Eggert, Zaheduzzaman Sarker, John Scudder, Eric Vyncke, Robert Wilton, Roman Danyliw, Benjamin Kaduk, Murray Kucherawy, Martin Vigoureaux, and Alvaro Retana

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

tls
Internet-Draft
Intended status: Standards Track
Expires: 17 August 2022

E. Rescorla
RTFM, Inc.
K. Oku
Fastly
N. Sullivan
C.A. Wood
Cloudflare
13 February 2022

TLS Encrypted Client Hello
draft-ietf-tls-esni-14

Abstract

This document describes a mechanism in Transport Layer Security (TLS) for encrypting a ClientHello message under a server public key.

Discussion Venues

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

Source for this draft and an issue tracker can be found at
<https://github.com/tlswg/draft-ietf-tls-esni>
(<https://github.com/tlswg/draft-ietf-tls-esni>).

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 17 August 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
2. Conventions and Definitions	4
3. Overview	4
3.1. Topologies	4
3.2. Encrypted ClientHello (ECH)	6
4. Encrypted ClientHello Configuration	6
4.1. Configuration Identifiers	9
4.2. Configuration Extensions	9
5. The "encrypted_client_hello" Extension	10
5.1. Encoding the ClientHelloInner	11
5.2. Authenticating the ClientHelloOuter	13
6. Client Behavior	14
6.1. Offering ECH	14
6.1.1. Encrypting the ClientHello	16
6.1.2. GREASE PSK	17
6.1.3. Recommended Padding Scheme	17
6.1.4. Determining ECH Acceptance	18
6.1.5. Handshaking with ClientHelloInner	19
6.1.6. Handshaking with ClientHelloOuter	20
6.1.7. Authenticating for the Public Name	21
6.2. GREASE ECH	22
7. Server Behavior	23
7.1. Client-Facing Server	23
7.1.1. Sending HelloRetryRequest	25
7.2. Backend Server	26
7.2.1. Sending HelloRetryRequest	27
8. Compatibility Issues	27
8.1. Misconfiguration and Deployment Concerns	28
8.2. Middleboxes	28
9. Compliance Requirements	28
10. Security Considerations	29
10.1. Security and Privacy Goals	29
10.2. Unauthenticated and Plaintext DNS	30
10.3. Client Tracking	30
10.4. Ignored Configuration Identifiers and Trial Decryption	31
10.5. Outer ClientHello	31

10.6.	Related Privacy Leaks	32
10.7.	Cookies	32
10.8.	Attacks Exploiting Acceptance Confirmation	33
10.9.	Comparison Against Criteria	33
10.9.1.	Mitigate Cut-and-Paste Attacks	34
10.9.2.	Avoid Widely Shared Secrets	34
10.9.3.	Prevent SNI-Based Denial-of-Service Attacks	34
10.9.4.	Do Not Stick Out	34
10.9.5.	Maintain Forward Secrecy	35
10.9.6.	Enable Multi-party Security Contexts	36
10.9.7.	Support Multiple Protocols	36
10.10.	Padding Policy	36
10.11.	Active Attack Mitigations	36
10.11.1.	Client Reaction Attack Mitigation	37
10.11.2.	HelloRetryRequest Hijack Mitigation	38
10.11.3.	ClientHello Malleability Mitigation	39
10.11.4.	ClientHelloInner Packet Amplification Mitigation	40
11.	IANA Considerations	41
11.1.	Update of the TLS ExtensionType Registry	41
11.2.	Update of the TLS Alert Registry	41
12.	ECHConfig Extension Guidance	41
13.	References	42
13.1.	Normative References	42
13.2.	Informative References	43
Appendix A.	Alternative SNI Protection Designs	44
A.1.	TLS-layer	44
A.1.1.	TLS in Early Data	44
A.1.2.	Combined Tickets	44
A.2.	Application-layer	45
A.2.1.	HTTP/2 CERTIFICATE Frames	45
Appendix B.	Linear-time Outer Extension Processing	45
Appendix C.	Acknowledgements	46
Appendix D.	Change Log	46
D.1.	Since draft-ietf-tls-esni-12	46
D.2.	Since draft-ietf-tls-esni-11	46
D.3.	Since draft-ietf-tls-esni-10	46
D.4.	Since draft-ietf-tls-esni-09	47
Authors'	Addresses	47

1. Introduction

DISCLAIMER: This draft is work-in-progress and has not yet seen significant (or really any) security analysis. It should not be used as a basis for building production systems. This published version of the draft has been designated an "implementation draft" for testing and interop purposes.

Although TLS 1.3 [RFC8446] encrypts most of the handshake, including the server certificate, there are several ways in which an on-path attacker can learn private information about the connection. The plaintext Server Name Indication (SNI) extension in ClientHello messages, which leaks the target domain for a given connection, is perhaps the most sensitive, unencrypted information in TLS 1.3.

The target domain may also be visible through other channels, such as plaintext client DNS queries or visible server IP addresses. However, DoH [RFC8484] and DPRIVE [RFC7858] [RFC8094] provide mechanisms for clients to conceal DNS lookups from network inspection, and many TLS servers host multiple domains on the same IP address. Private origins may also be deployed behind a common provider, such as a reverse proxy. In such environments, the SNI remains the primary explicit signal used to determine the server's identity.

This document specifies a new TLS extension, called Encrypted Client Hello (ECH), that allows clients to encrypt their ClientHello to such a deployment. This protects the SNI and other potentially sensitive fields, such as the ALPN list [RFC7301]. Co-located servers with consistent externally visible TLS configurations, including supported versions and cipher suites, form an anonymity set. Usage of this mechanism reveals that a client is connecting to a particular service provider, but does not reveal which server from the anonymity set terminates the connection.

ECH is only supported with (D)TLS 1.3 [RFC8446] and newer versions of the protocol.

2. Conventions and Definitions

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. All TLS notation comes from [RFC8446], Section 3.

3. Overview

This protocol is designed to operate in one of two topologies illustrated below, which we call "Shared Mode" and "Split Mode".

3.1. Topologies

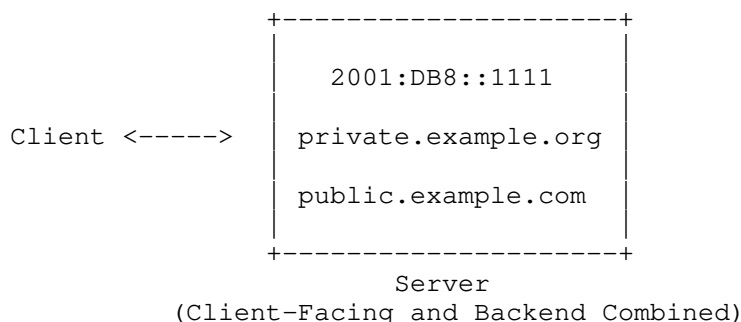


Figure 1: Shared Mode Topology

In Shared Mode, the provider is the origin server for all the domains whose DNS records point to it. In this mode, the TLS connection is terminated by the provider.

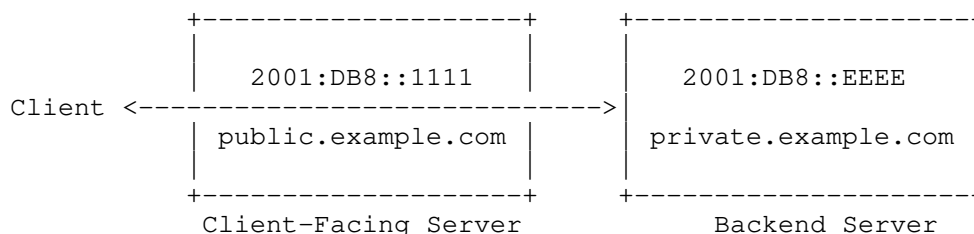


Figure 2: Split Mode Topology

In Split Mode, the provider is not the origin server for private domains. Rather, the DNS records for private domains point to the provider, and the provider's server relays the connection back to the origin server, who terminates the TLS connection with the client. Importantly, the service provider does not have access to the plaintext of the connection beyond the unencrypted portions of the handshake.

In the remainder of this document, we will refer to the ECH-service provider as the "client-facing server" and to the TLS terminator as the "backend server". These are the same entity in Shared Mode, but in Split Mode, the client-facing and backend servers are physically separated.

3.2. Encrypted ClientHello (ECH)

A client-facing server enables ECH by publishing an ECH configuration, which is an encryption public key and associated metadata. The server must publish this for all the domains it serves via Shared or Split Mode. This document defines the ECH configuration's format, but delegates DNS publication details to [HTTPS-RR]. Other delivery mechanisms are also possible. For example, the client may have the ECH configuration preconfigured.

When a client wants to establish a TLS session with some backend server, it constructs a private ClientHello, referred to as the ClientHelloInner. The client then constructs a public ClientHello, referred to as the ClientHelloOuter. The ClientHelloOuter contains innocuous values for sensitive extensions and an "encrypted_client_hello" extension (Section 5), which carries the encrypted ClientHelloInner. Finally, the client sends ClientHelloOuter to the server.

The server takes one of the following actions:

1. If it does not support ECH or cannot decrypt the extension, it completes the handshake with ClientHelloOuter. This is referred to as rejecting ECH.
2. If it successfully decrypts the extension, it forwards the ClientHelloInner to the backend server, which completes the handshake. This is referred to as accepting ECH.

Upon receiving the server's response, the client determines whether or not ECH was accepted (Section 6.1.4) and proceeds with the handshake accordingly. When ECH is rejected, the resulting connection is not usable by the client for application data. Instead, ECH rejection allows the client to retry with up-to-date configuration (Section 6.1.6).

The primary goal of ECH is to ensure that connections to servers in the same anonymity set are indistinguishable from one another. Moreover, it should achieve this goal without affecting any existing security properties of TLS 1.3. See Section 10.1 for more details about the ECH security and privacy goals.

4. Encrypted ClientHello Configuration

ECH uses HPKE for public key encryption [I-D.irtf-cfrg-hpke]. The ECH configuration is defined by the following ECHConfig structure.

```
opaque HpkePublicKey<1..2^16-1>;
uint16 HpkeKemId; // Defined in I-D.irtf-cfrg-hpke
uint16 HpkeKdfId; // Defined in I-D.irtf-cfrg-hpke
uint16 HpkeAeadId; // Defined in I-D.irtf-cfrg-hpke

struct {
    HpkeKdfId kdf_id;
    HpkeAeadId aead_id;
} HpkeSymmetricCipherSuite;

struct {
    uint8 config_id;
    HpkeKemId kem_id;
    HpkePublicKey public_key;
    HpkeSymmetricCipherSuite cipher_suites<4..2^16-4>;
} HpkeKeyConfig;

struct {
    HpkeKeyConfig key_config;
    uint8 maximum_name_length;
    opaque public_name<1..255>;
    Extension extensions<0..2^16-1>;
} ECHConfigContents;

struct {
    uint16 version;
    uint16 length;
    select (ECHConfig.version) {
        case 0xfe0d: ECHConfigContents contents;
    }
} ECHConfig;
```

The structure contains the following fields:

version The version of ECH for which this configuration is used. Beginning with draft-08, the version is the same as the code point for the "encrypted_client_hello" extension. Clients MUST ignore any ECHConfig structure with a version they do not support.

length The length, in bytes, of the next field. This length field allows implementations to skip over the elements in such a list where they cannot parse the specific version of ECHConfig.

contents An opaque byte string whose contents depend on the version. For this specification, the contents are an ECHConfigContents structure.

The ECHConfigContents structure contains the following fields:

key_config A HpkeKeyConfig structure carrying the configuration information associated with the HPKE public key. Note that this structure contains the config_id field, which applies to the entire ECHConfigContents.

maximum_name_length The longest name of a backend server, if known. If not known, this value can be set to zero. It is used to compute padding (Section 6.1.3) and does not constrain server name lengths. Names may exceed this length if, e.g., the server uses wildcard names or added new names to the anonymity set.

public_name The DNS name of the client-facing server, i.e., the entity trusted to update the ECH configuration. This is used to correct misconfigured clients, as described in Section 6.1.6.

Clients MUST ignore any ECHConfig structure whose public_name is not parsable as a dot-separated sequence of LDH labels, as defined in [RFC5890], Section 2.3.1 or which begins or end with an ASCII dot.

Clients SHOULD ignore the ECHConfig if it contains an encoded IPv4 address. To determine if a public_name value is an IPv4 address, clients can invoke the IPv4 parser algorithm in [WHATWG-IPV4]. It returns a value when the input is an IPv4 address.

See Section 6.1.7 for how the client interprets and validates the public_name.

extensions A list of extensions that the client must take into consideration when generating a ClientHello message. These are described below (Section 4.2).

[[OPEN ISSUE: determine if clients should enforce a 63-octet label limit for public_name]] [[OPEN ISSUE: fix reference to WHATWG-IPV4]]

The HpkeKeyConfig structure contains the following fields:

config_id A one-byte identifier for the given HPKE key configuration. This is used by clients to indicate the key used for ClientHello encryption. Section 4.1 describes how client-facing servers allocate this value.

kem_id The HPKE KEM identifier corresponding to public_key. Clients MUST ignore any ECHConfig structure with a key using a KEM they do not support.

public_key The HPKE public key used by the client to encrypt ClientHelloInner.

`cipher_suites` The list of HPKE KDF and AEAD identifier pairs clients can use for encrypting `ClientHelloInner`. See Section 6.1 for how clients choose from this list.

The client-facing server advertises a sequence of ECH configurations to clients, serialized as follows.

```
ECHConfig ECHConfigList<1..2^16-1>;
```

The `ECHConfigList` structure contains one or more `ECHConfig` structures in decreasing order of preference. This allows a server to support multiple versions of ECH and multiple sets of ECH parameters.

4.1. Configuration Identifiers

A client-facing server has a set of known `ECHConfig` values, with corresponding private keys. This set SHOULD contain the currently published values, as well as previous values that may still be in use, since clients may cache DNS records up to a TTL or longer.

Section 7.1 describes a trial decryption process for decrypting the `ClientHello`. This can impact performance when the client-facing server maintains many known `ECHConfig` values. To avoid this, the client-facing server SHOULD allocate distinct `config_id` values for each `ECHConfig` in its known set. The RECOMMENDED strategy is via rejection sampling, i.e., to randomly select `config_id` repeatedly until it does not match any known `ECHConfig`.

It is not necessary for `config_id` values across different client-facing servers to be distinct. A backend server may be hosted behind two different client-facing servers with colliding `config_id` values without any performance impact. Values may also be reused if the previous `ECHConfig` is no longer in the known set.

4.2. Configuration Extensions

ECH configuration extensions are used to provide room for additional functionality as needed. See Section 12 for guidance on which types of extensions are appropriate for this structure.

The format is as defined in [RFC8446], Section 4.2. The same interpretation rules apply: extensions MAY appear in any order, but there MUST NOT be more than one extension of the same type in the extensions block. An extension can be tagged as mandatory by using an extension type codepoint with the high order bit set to 1.

Clients MUST parse the extension list and check for unsupported mandatory extensions. If an unsupported mandatory extension is present, clients MUST ignore the ECHConfig.

5. The "encrypted_client_hello" Extension

To offer ECH, the client sends an "encrypted_client_hello" extension in the ClientHelloOuter. When it does, it MUST also send the extension in ClientHelloInner.

```
enum {  
    encrypted_client_hello(0xfe0d), (65535)  
} ExtensionType;
```

The payload of the extension has the following structure:

```
enum { outer(0), inner(1) } ECHClientHelloType;  
  
struct {  
    ECHClientHelloType type;  
    select (ECHClientHello.type) {  
        case outer:  
            HpkeSymmetricCipherSuite cipher_suite;  
            uint8 config_id;  
            opaque enc<0..2^16-1>;  
            opaque payload<1..2^16-1>;  
        case inner:  
            Empty;  
    };  
} ECHClientHello;
```

The outer extension uses the outer variant and the inner extension uses the inner variant. The inner extension has an empty payload. The outer extension has the following fields:

config_id The ECHConfigContents.key_config.config_id for the chosen ECHConfig.

cipher_suite The cipher suite used to encrypt ClientHelloInner. This MUST match a value provided in the corresponding ECHConfigContents.cipher_suites list.

enc The HPKE encapsulated key, used by servers to decrypt the corresponding payload field. This field is empty in a ClientHelloOuter sent in response to HelloRetryRequest.

payload The serialized and encrypted ClientHelloInner structure, encrypted using HPKE as described in Section 6.1.

When a client offers the outer version of an "encrypted_client_hello" extension, the server MAY include an "encrypted_client_hello" extension in its EncryptedExtensions message, as described in Section 7.1, with the following payload:

```
struct {
    ECHConfigList retry_configs;
} ECHEncryptedExtensions;
```

The response is valid only when the server used the ClientHelloOuter. If the server sent this extension in response to the inner variant, then the client MUST abort with an "unsupported_extension" alert.

`retry_configs` An ECHConfigList structure containing one or more ECHConfig structures, in decreasing order of preference, to be used by the client as described in Section 6.1.6. These are known as the server's "retry configurations".

Finally, when the client offers the "encrypted_client_hello", if the payload is the inner variant and the server responds with HelloRetryRequest, it MUST include an "encrypted_client_hello" extension with the following payload:

```
struct {
    opaque confirmation[8];
} ECHHelloRetryRequest;
```

The value of ECHHelloRetryRequest.confirmation is set to `hrr_accept_confirmation` as described in Section 7.2.1.

This document also defines the "ech_required" alert, which the client MUST send when it offered an "encrypted_client_hello" extension that was not accepted by the server. (See Section 11.2.)

5.1. Encoding the ClientHelloInner

Before encrypting, the client pads and optionally compresses ClientHelloInner into a EncodedClientHelloInner structure, defined below:

```
struct {
    ClientHello client_hello;
    uint8 zeros[length_of_padding];
} EncodedClientHelloInner;
```

The `client_hello` field is computed by first making a copy of `ClientHelloInner` and setting the `legacy_session_id` field to the empty string. Note this field uses the `ClientHello` structure, defined in Section 4.1.2 of [RFC8446] which does not include the Handshake structure's four byte header. The `zeros` field MUST be all zeroes.

Repeating large extensions, such as "key_share" with post-quantum algorithms, between `ClientHelloInner` and `ClientHelloOuter` can lead to excessive size. To reduce the size impact, the client MAY substitute extensions which it knows will be duplicated in `ClientHelloOuter`. It does so by removing and replacing extensions from `EncodedClientHelloInner` with a single "ech_outer_extensions" extension, defined as follows:

```
enum {  
    ech_outer_extensions(0xfd00), (65535)  
} ExtensionType;
```

```
ExtensionType OuterExtensions<2..254>;
```

`OuterExtensions` contains the removed `ExtensionType` values. Each value references the matching extension in `ClientHelloOuter`. The values MUST be ordered contiguously in `ClientHelloInner`, and the "ech_outer_extensions" extension MUST be inserted in the corresponding position in `EncodedClientHelloInner`. Additionally, the extensions MUST appear in `ClientHelloOuter` in the same relative order. However, there is no requirement that they be contiguous. For example, `OuterExtensions` may contain extensions A, B, C, while `ClientHelloOuter` contains extensions A, D, B, C, E, F.

The "ech_outer_extensions" extension can only be included in `EncodedClientHelloInner`, and MUST NOT appear in either `ClientHelloOuter` or `ClientHelloInner`.

Finally, the client pads the message by setting the `zeros` field to a byte string whose contents are all zeros and whose length is the amount of padding to add. Section 6.1.3 describes a recommended padding scheme.

The client-facing server computes `ClientHelloInner` by reversing this process. First it parses `EncodedClientHelloInner`, interpreting all bytes after `client_hello` as padding. If any padding byte is non-zero, the server MUST abort the connection with an "illegal_parameter" alert.

Next it makes a copy of the `client_hello` field and copies the `legacy_session_id` field from `ClientHelloOuter`. It then looks for an "ech_outer_extensions" extension. If found, it replaces the

extension with the corresponding sequence of extensions in the ClientHelloOuter. The server MUST abort the connection with an "illegal_parameter" alert if any of the following are true:

- * Any referenced extension is missing in ClientHelloOuter.
- * Any extension is referenced in OuterExtensions more than once.
- * "encrypted_client_hello" is referenced in OuterExtensions.
- * The extensions in ClientHelloOuter corresponding to those in OuterExtensions do not occur in the same order.

These requirements prevent an attacker from performing a packet amplification attack, by crafting a ClientHelloOuter which decompresses to a much larger ClientHelloInner. This is discussed further in Section 10.11.4.

Implementations SHOULD bound the time to compute a ClientHelloInner proportionally to the ClientHelloOuter size. If the cost is disproportionately large, a malicious client could exploit this in a denial of service attack. Appendix B describes a linear-time procedure that may be used for this purpose.

5.2. Authenticating the ClientHelloOuter

To prevent a network attacker from modifying the reconstructed ClientHelloInner (see Section 10.11.3), ECH authenticates ClientHelloOuter by passing ClientHelloOuterAAD as the associated data for HPKE sealing and opening operations. The ClientHelloOuterAAD is a serialized ClientHello structure, defined in Section 4.1.2 of [RFC8446], which matches the ClientHelloOuter except the payload field of the "encrypted_client_hello" is replaced with a byte string of the same length but whose contents are zeros. This value does not include the four-byte header from the Handshake structure.

The client follows the procedure in Section 6.1.1 to first construct ClientHelloOuterAAD with a placeholder payload field, then replace the field with the encrypted value to compute ClientHelloOuter.

The server then receives ClientHelloOuter and computes ClientHelloOuterAAD by making a copy and replacing the portion corresponding to the payload field with zeros.

The payload and the placeholder strings have the same length, so it is not necessary for either side to recompute length prefixes when applying the above transformations.

The decompression process in Section 5.1 forbids "encrypted_client_hello" in OuterExtensions. This ensures the unauthenticated portion of ClientHelloOuter is not incorporated into ClientHelloInner.

6. Client Behavior

Clients that implement the ECH extension behave in one of two ways: either they offer a real ECH extension, as described in Section 6.1; or they send a GREASE ECH extension, as described in Section 6.2. Clients of the latter type do not negotiate ECH. Instead, they generate a dummy ECH extension that is ignored by the server. (See Section 10.9.4 for an explanation.) The client offers ECH if it is in possession of a compatible ECH configuration and sends GREASE ECH otherwise.

6.1. Offering ECH

To offer ECH, the client first chooses a suitable ECHConfig from the server's ECHConfigList. To determine if a given ECHConfig is suitable, it checks that it supports the KEM algorithm identified by ECHConfig.contents.kem_id, at least one KDF/AEAD algorithm identified by ECHConfig.contents.cipher_suites, and the version of ECH indicated by ECHConfig.contents.version. Once a suitable configuration is found, the client selects the cipher suite it will use for encryption. It MUST NOT choose a cipher suite or version not advertised by the configuration. If no compatible configuration is found, then the client SHOULD proceed as described in Section 6.2.

Next, the client constructs the ClientHelloInner message just as it does a standard ClientHello, with the exception of the following rules:

1. It MUST NOT offer to negotiate TLS 1.2 or below. This is necessary to ensure the backend server does not negotiate a TLS version that is incompatible with ECH.
2. It MUST NOT offer to resume any session for TLS 1.2 and below.
3. If it intends to compress any extensions (see Section 5.1), it MUST order those extensions consecutively.
4. It MUST include the "encrypted_client_hello" extension of type inner as described in Section 5. (This requirement is not applicable when the "encrypted_client_hello" extension is generated as described in Section 6.2.)

The client then constructs `EncodedClientHelloInner` as described in Section 5.1. It also computes an HPKE encryption context and enc value as:

```
pkR = DeserializePublicKey(ECHConfig.contents.public_key)
enc, context = SetupBaseS(pkR,
                          "tls ech" || 0x00 || ECHConfig)
```

Next, it constructs a partial `ClientHelloOuterAAD` as it does a standard `ClientHello`, with the exception of the following rules:

1. It MUST offer to negotiate TLS 1.3 or above.
2. If it compressed any extensions in `EncodedClientHelloInner`, it MUST copy the corresponding extensions from `ClientHelloInner`. The copied extensions additionally MUST be in the same relative order as in `ClientHelloInner`.
3. It MUST copy the `legacy_session_id` field from `ClientHelloInner`. This allows the server to echo the correct session ID for TLS 1.3's compatibility mode (see Appendix D.4 of [RFC8446]) when ECH is negotiated.
4. It MAY copy any other field from the `ClientHelloInner` except `ClientHelloInner.random`. Instead, It MUST generate a fresh `ClientHelloOuter.random` using a secure random number generator. (See Section 10.11.1.)
5. The value of `ECHConfig.contents.public_name` MUST be placed in the "server_name" extension.
6. When the client offers the "pre_shared_key" extension in `ClientHelloInner`, it SHOULD also include a GREASE "pre_shared_key" extension in `ClientHelloOuter`, generated in the manner described in Section 6.1.2. The client MUST NOT use this extension to advertise a PSK to the client-facing server. (See Section 10.11.3.) When the client includes a GREASE "pre_shared_key" extension, it MUST also copy the "psk_key_exchange_modes" from the `ClientHelloInner` into the `ClientHelloOuter`.
7. When the client offers the "early_data" extension in `ClientHelloInner`, it MUST also include the "early_data" extension in `ClientHelloOuter`. This allows servers that reject ECH and use `ClientHelloOuter` to safely ignore any early data sent by the client per [RFC8446], Section 4.2.10.

Note that these rules may change in the presence of an application profile specifying otherwise.

The client might duplicate non-sensitive extensions in both messages. However, implementations need to take care to ensure that sensitive extensions are not offered in the ClientHelloOuter. See Section 10.5 for additional guidance.

Finally, the client encrypts the EncodedClientHelloInner with the above values, as described in Section 6.1.1, to construct a ClientHelloOuter. It sends this to the server, and processes the response as described in Section 6.1.4.

6.1.1. Encrypting the ClientHello

Given an EncodedClientHelloInner, an HPKE encryption context and enc value, and a partial ClientHelloOuterAAD, the client constructs a ClientHelloOuter as follows.

First, the client determines the length *L* of encrypting EncodedClientHelloInner with the selected HPKE AEAD. This is typically the sum of the plaintext length and the AEAD tag length. The client then completes the ClientHelloOuterAAD with an "encrypted_client_hello" extension. This extension value contains the outer variant of ECHClientHello with the following fields:

- * *config_id*, the identifier corresponding to the chosen ECHConfig structure;
- * *cipher_suite*, the client's chosen cipher suite;
- * *enc*, as given above; and
- * *payload*, a placeholder byte string containing *L* zeros.

If configuration identifiers (see Section 10.4) are to be ignored, *config_id* SHOULD be set to a randomly generated byte in the first ClientHelloOuter and, in the event of HRR, MUST be left unchanged for the second ClientHelloOuter.

The client serializes this structure to construct the ClientHelloOuterAAD. It then computes the final payload as:

```
final_payload = context.Seal(ClientHelloOuterAAD,  
                             EncodedClientHelloInner)
```

Finally, the client replaces payload with final_payload to obtain ClientHelloOuter. The two values have the same length, so it is not necessary to recompute length prefixes in the serialized structure.

Note this construction requires the "encrypted_client_hello" be computed after all other extensions. This is possible because the ClientHelloOuter's "pre_shared_key" extension is either omitted, or uses a random binder (Section 6.1.2).

6.1.2. GREASE PSK

When offering ECH, the client is not permitted to advertise PSK identities in the ClientHelloOuter. However, the client can send a "pre_shared_key" extension in the ClientHelloInner. In this case, when resuming a session with the client, the backend server sends a "pre_shared_key" extension in its ServerHello. This would appear to a network observer as if the the server were sending this extension without solicitation, which would violate the extension rules described in [RFC8446]. Sending a GREASE "pre_shared_key" extension in the ClientHelloOuter makes it appear to the network as if the extension were negotiated properly.

The client generates the extension payload by constructing an OfferedPsks structure (see [RFC8446], Section 4.2.11) as follows. For each PSK identity advertised in the ClientHelloInner, the client generates a random PSK identity with the same length. It also generates a random, 32-bit, unsigned integer to use as the obfuscated_ticket_age. Likewise, for each inner PSK binder, the client generates a random string of the same length.

Per the rules of Section 6.1, the server is not permitted to resume a connection in the outer handshake. If ECH is rejected and the client-facing server replies with a "pre_shared_key" extension in its ServerHello, then the client MUST abort the handshake with an "illegal_parameter" alert.

6.1.3. Recommended Padding Scheme

This section describes a deterministic padding mechanism based on the following observation: individual extensions can reveal sensitive information through their length. Thus, each extension in the inner ClientHello may require different amounts of padding. This padding may be fully determined by the client's configuration or may require server input.

By way of example, clients typically support a small number of application profiles. For instance, a browser might support HTTP with ALPN values ["http/1.1", "h2"] and WebRTC media with ALPNs

["webrtc", "c-webrtc"]. Clients SHOULD pad this extension by rounding up to the total size of the longest ALPN extension across all application profiles. The target padding length of most ClientHello extensions can be computed in this way.

In contrast, clients do not know the longest SNI value in the client-facing server's anonymity set without server input. Clients SHOULD use the ECHConfig's maximum_name_length field as follows, where L is the maximum_name_length value.

1. If the ClientHelloInner contained a "server_name" extension with a name of length D, add $\max(0, L - D)$ bytes of padding.
2. If the ClientHelloInner did not contain a "server_name" extension (e.g., if the client is connecting to an IP address), add $L + 9$ bytes of padding. This is the length of a "server_name" extension with an L-byte name.

Finally, the client SHOULD pad the entire message as follows:

1. Let L be the length of the EncodedClientHelloInner with all the padding computed so far.
2. Let $N = 31 - ((L - 1) \% 32)$ and add N bytes of padding.

This rounds the length of EncodedClientHelloInner up to a multiple of 32 bytes, reducing the set of possible lengths across all clients.

In addition to padding ClientHelloInner, clients and servers will also need to pad all other handshake messages that have sensitive-length fields. For example, if a client proposes ALPN values in ClientHelloInner, the server-selected value will be returned in an EncryptedExtension, so that handshake message also needs to be padded using TLS record layer padding.

6.1.4. Determining ECH Acceptance

As described in Section 7, the server may either accept ECH and use ClientHelloInner or reject it and use ClientHelloOuter. This is determined by the server's initial message.

If the message does not negotiate TLS 1.3 or higher, the server has rejected ECH. Otherwise, it is either a ServerHello or HelloRetryRequest.

If the message is a `ServerHello`, the client computes `accept_confirmation` as described in Section 7.2. If this value matches the last 8 bytes of `ServerHello.random`, the server has accepted ECH. Otherwise, it has rejected ECH.

If the message is a `HelloRetryRequest`, the client checks for the `"encrypted_client_hello"` extension. If none is found, the server has rejected ECH. Otherwise, if it has a length other than 8, the client aborts the handshake with a `"decode_error"` alert. Otherwise, the client computes `hrr_accept_confirmation` as described in Section 7.2.1. If this value matches the extension payload, the server has accepted ECH. Otherwise, it has rejected ECH.

[[OPEN ISSUE: Depending on what we do for issue#450, it may be appropriate to change the client behavior if the HRR extension is present but with the wrong value.]]

If the server accepts ECH, the client handshakes with `ClientHelloInner` as described in Section 6.1.5. Otherwise, the client handshakes with `ClientHelloOuter` as described in Section 6.1.6.

6.1.5. Handshaking with `ClientHelloInner`

If the server accepts ECH, the client proceeds with the connection as in [RFC8446], with the following modifications:

The client behaves as if it had sent `ClientHelloInner` as the `ClientHello`. That is, it evaluates the handshake using the `ClientHelloInner`'s preferences, and, when computing the transcript hash (Section 4.4.1 of [RFC8446]), it uses `ClientHelloInner` as the first `ClientHello`.

If the server responds with a `HelloRetryRequest`, the client computes the updated `ClientHello` message as follows:

1. It computes a second `ClientHelloInner` based on the first `ClientHelloInner`, as in Section 4.1.4 of [RFC8446]. The `ClientHelloInner`'s `"encrypted_client_hello"` extension is left unmodified.
2. It constructs `EncodedClientHelloInner` as described in Section 5.1.

3. It constructs a second partial ClientHelloOuterAAD message. This message MUST be syntactically valid. The extensions MAY be copied from the original ClientHelloOuter unmodified, or omitted. If not sensitive, the client MAY copy updated extensions from the second ClientHelloInner for compression.
4. It encrypts EncodedClientHelloInner as described in Section 6.1.1, using the second partial ClientHelloOuterAAD, to obtain a second ClientHelloOuter. It reuses the original HPKE encryption context computed in Section 6.1 and uses the empty string for enc.

The HPKE context maintains a sequence number, so this operation internally uses a fresh nonce for each AEAD operation. Reusing the HPKE context avoids an attack described in Section 10.11.2.

The client then sends the second ClientHelloOuter to the server. However, as above, it uses the second ClientHelloInner for preferences, and both the ClientHelloInner messages for the transcript hash. Additionally, it checks the resulting ServerHello for ECH acceptance as in Section 6.1.4. If the ServerHello does not also indicate ECH acceptance, the client MUST terminate the connection with an "illegal_parameter" alert.

6.1.6. Handshaking with ClientHelloOuter

If the server rejects ECH, the client proceeds with the handshake, authenticating for ECHConfig.contents.public_name as described in Section 6.1.7. If authentication or the handshake fails, the client MUST return a failure to the calling application. It MUST NOT use the retry configurations. It MUST NOT treat this as a secure signal to disable ECH.

If the server supplied an "encrypted_client_hello" extension in its EncryptedExtensions message, the client MUST check that it is syntactically valid and the client MUST abort the connection with a "decode_error" alert otherwise. If an earlier TLS version was negotiated, the client MUST NOT enable the False Start optimization [RFC7918] for this handshake. If both authentication and the handshake complete successfully, the client MUST perform the processing described below then abort the connection with an "ech_required" alert before sending any application data to the server.

If the server provided "retry_configs" and if at least one of the values contains a version supported by the client, the client can regard the ECH keys as securely replaced by the server. It SHOULD retry the handshake with a new transport connection, using the retry

configurations supplied by the server. The retry configurations may only be applied to the retry connection. The client MUST NOT use retry configurations for connections beyond the retry. This avoids introducing pinning concerns or a tracking vector, should a malicious server present client-specific retry configurations in order to identify the client in a subsequent ECH handshake.

If none of the values provided in "retry_configs" contains a supported version, or an earlier TLS version was negotiated, the client can regard ECH as securely disabled by the server, and it SHOULD retry the handshake with a new transport connection and ECH disabled.

Clients SHOULD implement a limit on retries caused by receipt of "retry_configs" or servers which do not acknowledge the "encrypted_client_hello" extension. If the client does not retry in either scenario, it MUST report an error to the calling application.

6.1.7. Authenticating for the Public Name

When the server rejects ECH, it continues with the handshake using the plaintext "server_name" extension instead (see Section 7). Clients that offer ECH then authenticate the connection with the public name, as follows:

- * The client MUST verify that the certificate is valid for ECHConfig.contents.public_name. If invalid, it MUST abort the connection with the appropriate alert.
- * If the server requests a client certificate, the client MUST respond with an empty Certificate message, denoting no client certificate.

In verifying the client-facing server certificate, the client MUST interpret the public name as a DNS-based reference identity. Clients that incorporate DNS names and IP addresses into the same syntax (e.g. [RFC3986], Section 7.4 and [WHATWG-IPV4]) MUST reject names that would be interpreted as IPv4 addresses. Clients that enforce this by checking and rejecting encoded IPv4 addresses in ECHConfig.contents.public_name do not need to repeat the check at this layer.

Note that authenticating a connection for the public name does not authenticate it for the origin. The TLS implementation MUST NOT report such connections as successful to the application. It additionally MUST ignore all session tickets and session IDs presented by the server. These connections are only used to trigger retries, as described in Section 6.1.6. This may be implemented, for instance, by reporting a failed connection with a dedicated error code.

6.2. GREASE ECH

If the client attempts to connect to a server and does not have an ECHConfig structure available for the server, it SHOULD send a GREASE [RFC8701] "encrypted_client_hello" extension in the first ClientHello as follows:

- * Set the config_id field to a random byte.
- * Set the cipher_suite field to a supported HpkeSymmetricCipherSuite. The selection SHOULD vary to exercise all supported configurations, but MAY be held constant for successive connections to the same server in the same session.
- * Set the enc field to a randomly-generated valid encapsulated public key output by the HPKE KEM.
- * Set the payload field to a randomly-generated string of L+C bytes, where C is the ciphertext expansion of the selected AEAD scheme and L is the size of the EncodedClientHelloInner the client would compute when offering ECH, padded according to Section 6.1.3.

If sending a second ClientHello in response to a HelloRetryRequest, the client copies the entire "encrypted_client_hello" extension from the first ClientHello. The identical value will reveal to an observer that the value of "encrypted_client_hello" was fake, but this only occurs if there is a HelloRetryRequest.

If the server sends an "encrypted_client_hello" extension in either HelloRetryRequest or EncryptedExtensions, the client MUST check the extension syntactically and abort the connection with a "decode_error" alert if it is invalid. It otherwise ignores the extension. It MUST NOT save the "retry_config" value in EncryptedExtensions.

Offering a GREASE extension is not considered offering an encrypted ClientHello for purposes of requirements in Section 6.1. In particular, the client MAY offer to resume sessions established without ECH.

7. Server Behavior

Servers that support ECH play one of two roles, depending on the payload of the "encrypted_client_hello" extension in the initial ClientHello:

- * If ECHClientHello.type is outer, then the server acts as a client-facing server and proceeds as described in Section 7.1 to extract a ClientHelloInner, if available.
- * If ECHClientHello.type is inner, then the server acts as a backend server and proceeds as described in Section 7.2.
- * Otherwise, if ECHClientHello.type is not a valid ECHClientHelloType, then the server MUST abort with an "illegal_parameter" alert.

If the "encrypted_client_hello" is not present, then the server completes the handshake normally, as described in [RFC8446].

7.1. Client-Facing Server

Upon receiving an "encrypted_client_hello" extension in an initial ClientHello, the client-facing server determines if it will accept ECH, prior to negotiating any other TLS parameters. Note that successfully decrypting the extension will result in a new ClientHello to process, so even the client's TLS version preferences may have changed.

First, the server collects a set of candidate ECHConfig values. This list is determined by one of the two following methods:

1. Compare ECHClientHello.config_id against identifiers of each known ECHConfig and select the ones that match, if any, as candidates.
2. Collect all known ECHConfig values as candidates, with trial decryption below determining the final selection.

Some uses of ECH, such as local discovery mode, may randomize the ECHClientHello.config_id since it can be used as a tracking vector. In such cases, the second method should be used for matching the ECHClientHello to a known ECHConfig. See Section 10.4. Unless specified by the application profile or otherwise externally configured, implementations MUST use the first method.

The server then iterates over the candidate ECHConfig values, attempting to decrypt the "encrypted_client_hello" extension:

The server verifies that the ECHConfig supports the cipher suite indicated by the ECHClientHello.cipher_suite and that the version of ECH indicated by the client matches the ECHConfig.version. If not, the server continues to the next candidate ECHConfig.

Next, the server decrypts ECHClientHello.payload, using the private key skR corresponding to ECHConfig, as follows:

```
context = SetupBaseR(ECHClientHello.enc, skR,  
                    "tls ech" || 0x00 || ECHConfig)  
EncodedClientHelloInner = context.Open(ClientHelloOuterAAD,  
                                       ECHClientHello.payload)
```

ClientHelloOuterAAD is computed from ClientHelloOuter as described in Section 5.2. The info parameter to SetupBaseR is the concatenation "tls ech", a zero byte, and the serialized ECHConfig. If decryption fails, the server continues to the next candidate ECHConfig. Otherwise, the server reconstructs ClientHelloInner from EncodedClientHelloInner, as described in Section 5.1. It then stops iterating over the candidate ECHConfig values.

Upon determining the ClientHelloInner, the client-facing server checks that the message includes a well-formed "encrypted_client_hello" extension of type inner and that it does not offer TLS 1.2 or below. If either of these checks fails, the client-facing server MUST abort with an "illegal_parameter" alert.

If these checks succeed, the client-facing server then forwards the ClientHelloInner to the appropriate backend server, which proceeds as in Section 7.2. If the backend server responds with a HelloRetryRequest, the client-facing server forwards it, decrypts the client's second ClientHelloOuter using the procedure in Section 7.1.1, and forwards the resulting second ClientHelloInner. The client-facing server forwards all other TLS messages between the client and backend server unmodified.

Otherwise, if all candidate ECHConfig values fail to decrypt the extension, the client-facing server MUST ignore the extension and proceed with the connection using ClientHelloOuter, with the following modifications:

- * If sending a HelloRetryRequest, the server MAY include an "encrypted_client_hello" extension with a payload of 8 random bytes; see Section 10.9.4 for details.
- * If the server is configured with any ECHConfigs, it MUST include the "encrypted_client_hello" extension in its EncryptedExtensions with the "retry_configs" field set to one or more ECHConfig

structures with up-to-date keys. Servers MAY supply multiple ECHConfig values of different versions. This allows a server to support multiple versions at once.

Note that decryption failure could indicate a GREASE ECH extension (see Section 6.2), so it is necessary for servers to proceed with the connection and rely on the client to abort if ECH was required. In particular, the unrecognized value alone does not indicate a misconfigured ECH advertisement (Section 8.1). Instead, servers can measure occurrences of the "ech_required" alert to detect this case.

7.1.1. Sending HelloRetryRequest

After sending or forwarding a HelloRetryRequest, the client-facing server does not repeat the steps in Section 7.1 with the second ClientHelloOuter. Instead, it continues with the ECHConfig selection from the first ClientHelloOuter as follows:

If the client-facing server accepted ECH, it checks the second ClientHelloOuter also contains the "encrypted_client_hello" extension. If not, it MUST abort the handshake with a "missing_extension" alert. Otherwise, it checks that ECHClientHello.cipher_suite and ECHClientHello.config_id are unchanged, and that ECHClientHello.enc is empty. If not, it MUST abort the handshake with an "illegal_parameter" alert.

Finally, it decrypts the new ECHClientHello.payload as a second message with the previous HPKE context:

```
EncodedClientHelloInner = context.Open(ClientHelloOuterAAD,
                                       ECHClientHello.payload)
```

ClientHelloOuterAAD is computed as described in Section 5.2, but using the second ClientHelloOuter. If decryption fails, the client-facing server MUST abort the handshake with a "decrypt_error" alert. Otherwise, it reconstructs the second ClientHelloInner from the new EncodedClientHelloInner as described in Section 5.1, using the second ClientHelloOuter for any referenced extensions.

The client-facing server then forwards the resulting ClientHelloInner to the backend server. It forwards all subsequent TLS messages between the client and backend server unmodified.

If the client-facing server rejected ECH, or if the first ClientHello did not include an "encrypted_client_hello" extension, the client-facing server proceeds with the connection as usual. The server does not decrypt the second ClientHello's ECHClientHello.payload value, if there is one. Moreover, if the server is configured with any

ECHConfigs, it MUST include the "encrypted_client_hello" extension in its EncryptedExtensions with the "retry_configs" field set to one or more ECHConfig structures with up-to-date keys, as described in Section 7.1.

Note that a client-facing server that forwards the first ClientHello cannot include its own "cookie" extension if the backend server sends a HelloRetryRequest. This means that the client-facing server either needs to maintain state for such a connection or it needs to coordinate with the backend server to include any information it requires to process the second ClientHello.

7.2. Backend Server

Upon receipt of an "encrypted_client_hello" extension of type inner in a ClientHello, if the backend server negotiates TLS 1.3 or higher, then it MUST confirm ECH acceptance to the client by computing its ServerHello as described here.

The backend server embeds in ServerHello.random a string derived from the inner handshake. It begins by computing its ServerHello as usual, except the last 8 bytes of ServerHello.random are set to zero. It then computes the transcript hash for ClientHelloInner up to and including the modified ServerHello, as described in [RFC8446], Section 4.4.1. Let transcript_ech_conf denote the output. Finally, the backend server overwrites the last 8 bytes of the ServerHello.random with the following string:

```
accept_confirmation = HKDF-Expand-Label(  
    HKDF-Extract(0, ClientHelloInner.random),  
    "ech accept confirmation",  
    transcript_ech_conf,  
    8)
```

where HKDF-Expand-Label is defined in [RFC8446], Section 7.1, "0" indicates a string of Hash.length bytes set to zero, and Hash is the hash function used to compute the transcript hash.

The backend server MUST NOT perform this operation if it negotiated TLS 1.2 or below. Note that doing so would overwrite the downgrade signal for TLS 1.3 (see [RFC8446], Section 4.1.3).

7.2.1. Sending HelloRetryRequest

When the backend server sends HelloRetryRequest in response to the ClientHello, it similarly confirms ECH acceptance by adding a confirmation signal to its HelloRetryRequest. But instead of embedding the signal in the HelloRetryRequest.random (the value of which is specified by [RFC8446]), it sends the signal in an extension.

The backend server begins by computing HelloRetryRequest as usual, except that it also contains an "encrypted_client_hello" extension with a payload of 8 zero bytes. It then computes the transcript hash for the first ClientHelloInner, denoted ClientHelloInner1, up to and including the modified HelloRetryRequest. Let transcript_hrr_ech_conf denote the output. Finally, the backend server overwrites the payload of the "encrypted_client_hello" extension with the following string:

```
hrr_accept_confirmation = HKDF-Expand-Label(  
    HKDF-Extract(0, ClientHelloInner1.random),  
    "hrr ech accept confirmation",  
    transcript_hrr_ech_conf,  
    8)
```

In the subsequent ServerHello message, the backend server sends the accept_confirmation value as described in Section 7.2.

8. Compatibility Issues

Unlike most TLS extensions, placing the SNI value in an ECH extension is not interoperable with existing servers, which expect the value in the existing plaintext extension. Thus server operators SHOULD ensure servers understand a given set of ECH keys before advertising them. Additionally, servers SHOULD retain support for any previously-advertised keys for the duration of their validity.

However, in more complex deployment scenarios, this may be difficult to fully guarantee. Thus this protocol was designed to be robust in case of inconsistencies between systems that advertise ECH keys and servers, at the cost of extra round-trips due to a retry. Two specific scenarios are detailed below.

8.1. Misconfiguration and Deployment Concerns

It is possible for ECH advertisements and servers to become inconsistent. This may occur, for instance, from DNS misconfiguration, caching issues, or an incomplete rollout in a multi-server deployment. This may also occur if a server loses its ECH keys, or if a deployment of ECH must be rolled back on the server.

The retry mechanism repairs inconsistencies, provided the server is authoritative for the public name. If server and advertised keys mismatch, the server will reject ECH and respond with "retry_configs". If the server does not understand the "encrypted_client_hello" extension at all, it will ignore it as required by Section 4.1.2 of [RFC8446]. Provided the server can present a certificate valid for the public name, the client can safely retry with updated settings, as described in Section 6.1.6.

Unless ECH is disabled as a result of successfully establishing a connection to the public name, the client **MUST NOT** fall back to using unencrypted ClientHellos, as this allows a network attacker to disclose the contents of this ClientHello, including the SNI. It **MAY** attempt to use another server from the DNS results, if one is provided.

8.2. Middleboxes

When connecting through a TLS-terminating proxy that does not support this extension, [RFC8446], Section 9.3 requires the proxy still act as a conforming TLS client and server. The proxy must ignore unknown parameters, and generate its own ClientHello containing only parameters it understands. Thus, when presenting a certificate to the client or sending a ClientHello to the server, the proxy will act as if connecting to the public name, without echoing the "encrypted_client_hello" extension.

Depending on whether the client is configured to accept the proxy's certificate as authoritative for the public name, this may trigger the retry logic described in Section 6.1.6 or result in a connection failure. A proxy which is not authoritative for the public name cannot forge a signal to disable ECH.

9. Compliance Requirements

In the absence of an application profile standard specifying otherwise, a compliant ECH application **MUST** implement the following HPKE cipher suite:

- * KEM: DHKEM(X25519, HKDF-SHA256) (see [I-D.irtf-cfrg-hpke], Section 7.1)
- * KDF: HKDF-SHA256 (see [I-D.irtf-cfrg-hpke], Section 7.2)
- * AEAD: AES-128-GCM (see [I-D.irtf-cfrg-hpke], Section 7.3)

10. Security Considerations

10.1. Security and Privacy Goals

ECH considers two types of attackers: passive and active. Passive attackers can read packets from the network, but they cannot perform any sort of active behavior such as probing servers or querying DNS. A middlebox that filters based on plaintext packet contents is one example of a passive attacker. In contrast, active attackers can also write packets into the network for malicious purposes, such as interfering with existing connections, probing servers, and querying DNS. In short, an active attacker corresponds to the conventional threat model for TLS 1.3 [RFC8446].

Given these types of attackers, the primary goals of ECH are as follows.

1. Use of ECH does not weaken the security properties of TLS without ECH.
2. TLS connection establishment to a host with a specific ECHConfig and TLS configuration is indistinguishable from a connection to any other host with the same ECHConfig and TLS configuration. (The set of hosts which share the same ECHConfig and TLS configuration is referred to as the anonymity set.)

Client-facing server configuration determines the size of the anonymity set. For example, if a client-facing server uses distinct ECHConfig values for each host, then each anonymity set has size $k = 1$. Client-facing servers SHOULD deploy ECH in such a way so as to maximize the size of the anonymity set where possible. This means client-facing servers should use the same ECHConfig for as many hosts as possible. An attacker can distinguish two hosts that have different ECHConfig values based on the ECHClientHello.config_id value. This also means public information in a TLS handshake should be consistent across hosts. For example, if a client-facing server services many backend origin hosts, only one of which supports some cipher suite, it may be possible to identify that host based on the contents of unencrypted handshake messages.

Beyond these primary security and privacy goals, ECH also aims to hide, to some extent, the fact that it is being used at all. Specifically, the GREASE ECH extension described in Section 6.2 does not change the security properties of the TLS handshake at all. Its goal is to provide "cover" for the real ECH protocol (Section 6.1), as a means of addressing the "do not stick out" requirements of [RFC8744]. See Section 10.9.4 for details.

10.2. Unauthenticated and Plaintext DNS

In comparison to [I-D.kazuho-protected-sni], wherein DNS Resource Records are signed via a server private key, ECH records have no authenticity or provenance information. This means that any attacker which can inject DNS responses or poison DNS caches, which is a common scenario in client access networks, can supply clients with fake ECH records (so that the client encrypts data to them) or strip the ECH record from the response. However, in the face of an attacker that controls DNS, no encryption scheme can work because the attacker can replace the IP address, thus blocking client connections, or substitute a unique IP address which is 1:1 with the DNS name that was looked up (modulo DNS wildcards). Thus, allowing the ECH records in the clear does not make the situation significantly worse.

Clearly, DNSSEC (if the client validates and hard fails) is a defense against this form of attack, but DoH/DPRIVE are also defenses against DNS attacks by attackers on the local network, which is a common case where ClientHello and SNI encryption are desired. Moreover, as noted in the introduction, SNI encryption is less useful without encryption of DNS queries in transit via DoH or DPRIVE mechanisms.

10.3. Client Tracking

A malicious client-facing server could distribute unique, per-client ECHConfig structures as a way of tracking clients across subsequent connections. On-path adversaries which know about these unique keys could also track clients in this way by observing TLS connection attempts.

The cost of this type of attack scales linearly with the desired number of target clients. Moreover, DNS caching behavior makes targeting individual users for extended periods of time, e.g., using per-client ECHConfig structures delivered via HTTPS RRs with high TTLs, challenging. Clients can help mitigate this problem by flushing any DNS or ECHConfig state upon changing networks.

10.4. Ignored Configuration Identifiers and Trial Decryption

Ignoring configuration identifiers may be useful in scenarios where clients and client-facing servers do not want to reveal information about the client-facing server in the "encrypted_client_hello" extension. In such settings, clients send a randomly generated config_id in the ECHClientHello. Servers in these settings must perform trial decryption since they cannot identify the client's chosen ECH key using the config_id value. As a result, ignoring configuration identifiers may exacerbate DoS attacks. Specifically, an adversary may send malicious ClientHello messages, i.e., those which will not decrypt with any known ECH key, in order to force wasteful decryption. Servers that support this feature should, for example, implement some form of rate limiting mechanism to limit the potential damage caused by such attacks.

Unless specified by the application using (D)TLS or externally configured, implementations MUST NOT use this mode.

10.5. Outer ClientHello

Any information that the client includes in the ClientHelloOuter is visible to passive observers. The client SHOULD NOT send values in the ClientHelloOuter which would reveal a sensitive ClientHelloInner property, such as the true server name. It MAY send values associated with the public name in the ClientHelloOuter.

In particular, some extensions require the client send a server-name-specific value in the ClientHello. These values may reveal information about the true server name. For example, the "cached_info" ClientHello extension [RFC7924] can contain the hash of a previously observed server certificate. The client SHOULD NOT send values associated with the true server name in the ClientHelloOuter. It MAY send such values in the ClientHelloInner.

A client may also use different preferences in different contexts. For example, it may send a different ALPN lists to different servers or in different application contexts. A client that treats this context as sensitive SHOULD NOT send context-specific values in ClientHelloOuter.

Values which are independent of the true server name, or other information the client wishes to protect, MAY be included in ClientHelloOuter. If they match the corresponding ClientHelloInner, they MAY be compressed as described in Section 5.1. However, note the payload length reveals information about which extensions are compressed, so inner extensions which only sometimes match the corresponding outer extension SHOULD NOT be compressed.

Clients MAY include additional extensions in ClientHelloOuter to avoid signaling unusual behavior to passive observers, provided the choice of value and value itself are not sensitive. See Section 10.9.4.

10.6. Related Privacy Leaks

ECH requires encrypted DNS to be an effective privacy protection mechanism. However, verifying the server's identity from the Certificate message, particularly when using the X509 CertificateType, may result in additional network traffic that may reveal the server identity. Examples of this traffic may include requests for revocation information, such as OCSP or CRL traffic, or requests for repository information, such as authorityInformationAccess. It may also include implementation-specific traffic for additional information sources as part of verification.

Implementations SHOULD avoid leaking information that may identify the server. Even when sent over an encrypted transport, such requests may result in indirect exposure of the server's identity, such as indicating a specific CA or service being used. To mitigate this risk, servers SHOULD deliver such information in-band when possible, such as through the use of OCSP stapling, and clients SHOULD take steps to minimize or protect such requests during certificate validation.

Attacks that rely on non-ECH traffic to infer server identity in an ECH connection are out of scope for this document. For example, a client that connects to a particular host prior to ECH deployment may later resume a connection to that same host after ECH deployment. An adversary that observes this can deduce that the ECH-enabled connection was made to a host that the client previously connected to and which is within the same anonymity set.

10.7. Cookies

Section 4.2.2 of [RFC8446] defines a cookie value that servers may send in HelloRetryRequest for clients to echo in the second ClientHello. While ECH encrypts the cookie in the second ClientHelloInner, the backend server's HelloRetryRequest is unencrypted. This means differences in cookies between backend servers, such as lengths or cleartext components, may leak information about the server identity.

Backend servers in an anonymity set SHOULD NOT reveal information in the cookie which identifies the server. This may be done by handling HelloRetryRequest statefully, thus not sending cookies, or by using the same cookie construction for all backend servers.

Note that, if the cookie includes a key name, analogous to Section 4 of [RFC5077], this may leak information if different backend servers issue cookies with different key names at the time of the connection. In particular, if the deployment operates in Split Mode, the backend servers may not share cookie encryption keys. Backend servers may mitigate this by either handling key rotation with trial decryption, or coordinating to match key names.

10.8. Attacks Exploiting Acceptance Confirmation

To signal acceptance, the backend server overwrites 8 bytes of its ServerHello.random with a value derived from the ClientHelloInner.random. (See Section 7.2 for details.) This behavior increases the likelihood of the ServerHello.random colliding with the ServerHello.random of a previous session, potentially reducing the overall security of the protocol. However, the remaining 24 bytes provide enough entropy to ensure this is not a practical avenue of attack.

On the other hand, the probability that two 8-byte strings are the same is non-negligible. This poses a modest operational risk. Suppose the client-facing server terminates the connection (i.e., ECH is rejected or bypassed): if the last 8 bytes of its ServerHello.random coincide with the confirmation signal, then the client will incorrectly presume acceptance and proceed as if the backend server terminated the connection. However, the probability of a false positive occurring for a given connection is only 1 in 2^{64} . This value is smaller than the probability of network connection failures in practice.

Note that the same bytes of the ServerHello.random are used to implement downgrade protection for TLS 1.3 (see [RFC8446], Section 4.1.3). These mechanisms do not interfere because the backend server only signals ECH acceptance in TLS 1.3 or higher.

10.9. Comparison Against Criteria

[RFC8744] lists several requirements for SNI encryption. In this section, we re-iterate these requirements and assess the ECH design against them.

10.9.1. Mitigate Cut-and-Paste Attacks

Since servers process either ClientHelloInner or ClientHelloOuter, and because ClientHelloInner.random is encrypted, it is not possible for an attacker to "cut and paste" the ECH value in a different Client Hello and learn information from ClientHelloInner.

10.9.2. Avoid Widely Shared Secrets

This design depends upon DNS as a vehicle for semi-static public key distribution. Server operators may partition their private keys however they see fit provided each server behind an IP address has the corresponding private key to decrypt a key. Thus, when one ECH key is provided, sharing is optimally bound by the number of hosts that share an IP address. Server operators may further limit sharing by publishing different DNS records containing ECHConfig values with different keys using a short TTL.

10.9.3. Prevent SNI-Based Denial-of-Service Attacks

This design requires servers to decrypt ClientHello messages with ECHClientHello extensions carrying valid digests. Thus, it is possible for an attacker to force decryption operations on the server. This attack is bound by the number of valid TCP connections an attacker can open.

10.9.4. Do Not Stick Out

As a means of reducing the impact of network ossification, [RFC8744] recommends SNI-protection mechanisms be designed in such a way that network operators do not differentiate connections using the mechanism from connections not using the mechanism. To that end, ECH is designed to resemble a standard TLS handshake as much as possible. The most obvious difference is the extension itself: as long as middleboxes ignore it, as required by [RFC8446], the rest of the handshake is designed to look very much as usual.

The GREASE ECH protocol described in Section 6.2 provides a low-risk way to evaluate the deployability of ECH. It is designed to mimic the real ECH protocol (Section 6.1) without changing the security properties of the handshake. The underlying theory is that if GREASE ECH is deployable without triggering middlebox misbehavior, and real ECH looks enough like GREASE ECH, then ECH should be deployable as well. Thus, our strategy for mitigating network ossification is to deploy GREASE ECH widely enough to disincentivize differential treatment of the real ECH protocol by the network.

Ensuring that networks do not differentiate between real ECH and GREASE ECH may not be feasible for all implementations. While most middleboxes will not treat them differently, some operators may wish to block real ECH usage but allow GREASE ECH. This specification aims to provide a baseline security level that most deployments can achieve easily, while providing implementations enough flexibility to achieve stronger security where possible. Minimally, real ECH is designed to be indifferentiable from GREASE ECH for passive adversaries with following capabilities:

1. The attacker does not know the ECHConfigList used by the server.
2. The attacker keeps per-connection state only. In particular, it does not track endpoints across connections.
3. ECH and GREASE ECH are designed so that the following features do not vary: the code points of extensions negotiated in the clear; the length of messages; and the values of plaintext alert messages.

This leaves a variety of practical differentiators out-of-scope. including, though not limited to, the following:

1. the value of the configuration identifier;
2. the value of the outer SNI;
3. the TLS version negotiated, which may depend on ECH acceptance;
4. client authentication, which may depend on ECH acceptance; and
5. HRR issuance, which may depend on ECH acceptance.

These can be addressed with more sophisticated implementations, but some mitigations require coordination between the client and server. These mitigations are out-of-scope for this specification.

10.9.5. Maintain Forward Secrecy

This design is not forward secret because the server's ECH key is static. However, the window of exposure is bound by the key lifetime. It is RECOMMENDED that servers rotate keys frequently.

10.9.6. Enable Multi-party Security Contexts

This design permits servers operating in Split Mode to forward connections directly to backend origin servers. The client authenticates the identity of the backend origin server, thereby avoiding unnecessary MiTM attacks.

Conversely, assuming ECH records retrieved from DNS are authenticated, e.g., via DNSSEC or fetched from a trusted Recursive Resolver, spoofing a client-facing server operating in Split Mode is not possible. See Section 10.2 for more details regarding plaintext DNS.

Authenticating the ECHConfig structure naturally authenticates the included public name. This also authenticates any retry signals from the client-facing server because the client validates the server certificate against the public name before retrying.

10.9.7. Support Multiple Protocols

This design has no impact on application layer protocol negotiation. It may affect connection routing, server certificate selection, and client certificate verification. Thus, it is compatible with multiple application and transport protocols. By encrypting the entire ClientHello, this design additionally supports encrypting the ALPN extension.

10.10. Padding Policy

Variations in the length of the ClientHelloInner ciphertext could leak information about the corresponding plaintext. Section 6.1.3 describes a RECOMMENDED padding mechanism for clients aimed at reducing potential information leakage.

10.11. Active Attack Mitigations

This section describes the rationale for ECH properties and mechanics as defenses against active attacks. In all the attacks below, the attacker is on-path between the target client and server. The goal of the attacker is to learn private information about the inner ClientHello, such as the true SNI value.

10.11.1. Client Reaction Attack Mitigation

This attack uses the client's reaction to an incorrect certificate as an oracle. The attacker intercepts a legitimate ClientHello and replies with a ServerHello, Certificate, CertificateVerify, and Finished messages, wherein the Certificate message contains a "test" certificate for the domain name it wishes to query. If the client decrypted the Certificate and failed verification (or leaked information about its verification process by a timing side channel), the attacker learns that its test certificate name was incorrect. As an example, suppose the client's SNI value in its inner ClientHello is "example.com," and the attacker replied with a Certificate for "test.com". If the client produces a verification failure alert because of the mismatch faster than it would due to the Certificate signature validation, information about the name leaks. Note that the attacker can also withhold the CertificateVerify message. In that scenario, a client which first verifies the Certificate would then respond similarly and leak the same information.



Figure 3: Client reaction attack

ClientHelloInner.random prevents this attack. In particular, since the attacker does not have access to this value, it cannot produce the right transcript and handshake keys needed for encrypting the Certificate message. Thus, the client will fail to decrypt the Certificate and abort the connection.

10.11.2. HelloRetryRequest Hijack Mitigation

This attack aims to exploit server HRR state management to recover information about a legitimate ClientHello using its own attacker-controlled ClientHello. To begin, the attacker intercepts and forwards a legitimate ClientHello with an "encrypted_client_hello" (ech) extension to the server, which triggers a legitimate HelloRetryRequest in return. Rather than forward the retry to the client, the attacker attempts to generate its own ClientHello in response based on the contents of the first ClientHello and HelloRetryRequest exchange with the result that the server encrypts the Certificate to the attacker. If the server used the SNI from the first ClientHello and the key share from the second (attacker-controlled) ClientHello, the Certificate produced would leak the client's chosen SNI to the attacker.

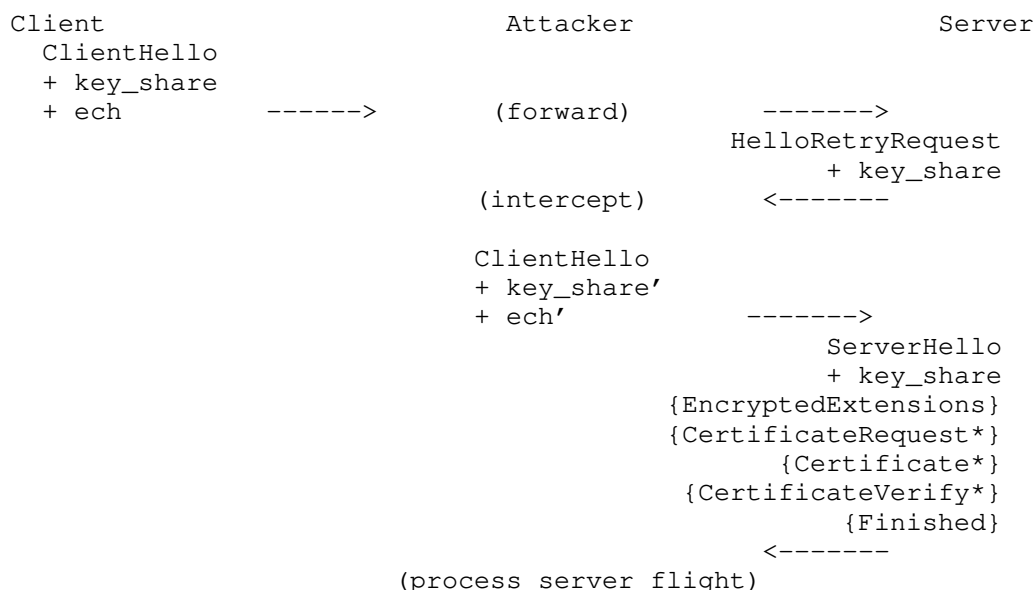


Figure 4: HelloRetryRequest hijack attack

This attack is mitigated by using the same HPKE context for both ClientHello messages. The attacker does not possess the context's keys, so it cannot generate a valid encryption of the second inner ClientHello.

If the attacker could manipulate the second ClientHello, it might be possible for the server to act as an oracle if it required parameters from the first ClientHello to match that of the second ClientHello. For example, imagine the client's original SNI value in the inner

ClientHello is "example.com", and the attacker's hijacked SNI value in its inner ClientHello is "test.com". A server which checks these for equality and changes behavior based on the result can be used as an oracle to learn the client's SNI.

10.11.3. ClientHello Malleability Mitigation

This attack aims to leak information about secret parts of the encrypted ClientHello by adding attacker-controlled parameters and observing the server's response. In particular, the compression mechanism described in Section 5.1 references parts of a potentially attacker-controlled ClientHelloOuter to construct ClientHelloInner, or a buggy server may incorrectly apply parameters from ClientHelloOuter to the handshake.

To begin, the attacker first interacts with a server to obtain a resumption ticket for a given test domain, such as "example.com". Later, upon receipt of a ClientHelloOuter, it modifies it such that the server will process the resumption ticket with ClientHelloInner. If the server only accepts resumption PSKs that match the server name, it will fail the PSK binder check with an alert when ClientHelloInner is for "example.com" but silently ignore the PSK and continue when ClientHelloInner is for any other name. This introduces an oracle for testing encrypted SNI values.

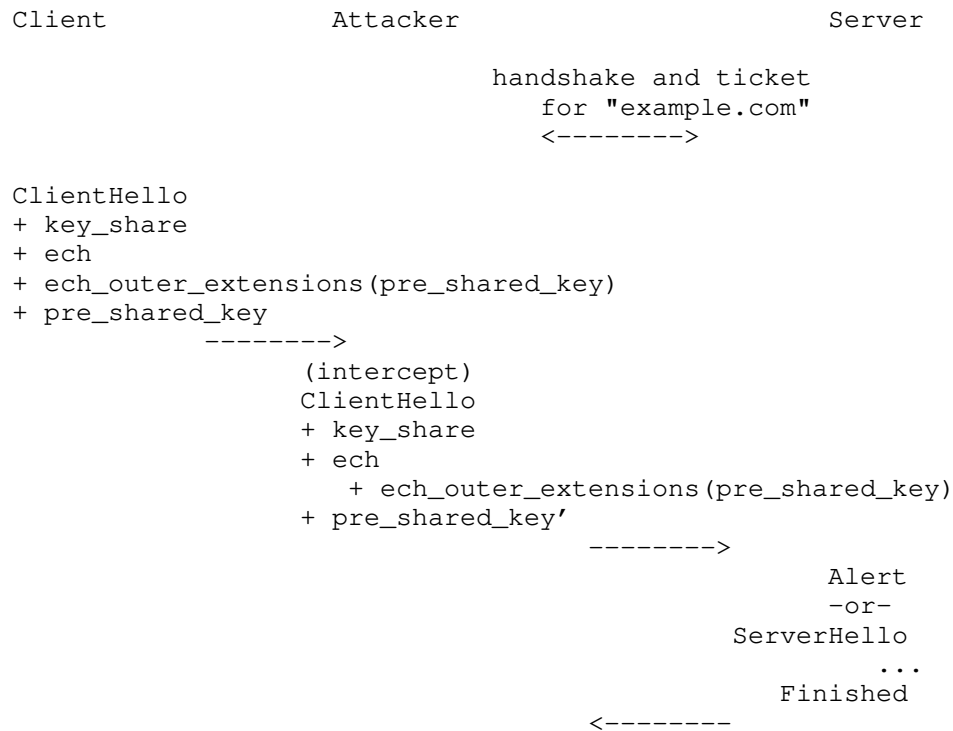


Figure 5: Message flow for malleable ClientHello

This attack may be generalized to any parameter which the server varies by server name, such as ALPN preferences.

ECH mitigates this attack by only negotiating TLS parameters from ClientHelloInner and authenticating all inputs to the ClientHelloInner (EncodedClientHelloInner and ClientHelloOuter) with the HPKE AEAD. See Section 5.2. An earlier iteration of this specification only encrypted and authenticated the "server_name" extension, which left the overall ClientHello vulnerable to an analogue of this attack.

10.11.4. ClientHelloInner Packet Amplification Mitigation

Client-facing servers must decompress EncodedClientHelloInners. A malicious attacker may craft a packet which takes excessive resources to decompress or may be much larger than the incoming packet:

- * If looking up a ClientHelloOuter extension takes time linear in the number of extensions, the overall decoding process would take $O(M*N)$ time, where M is the number of extensions in ClientHelloOuter and N is the size of OuterExtensions.
- * If the same ClientHelloOuter extension can be copied multiple times, an attacker could cause the client-facing server to construct a large ClientHelloInner by including a large extension in ClientHelloOuter, of length L , and an OuterExtensions list referencing N copies of that extension. The client-facing server would then use $O(N*L)$ memory in response to $O(N+L)$ bandwidth from the client. In split-mode, an $O(N*L)$ sized packet would then be transmitted to the backend server.

ECH mitigates this attack by requiring that OuterExtensions be referenced in order, that duplicate references be rejected, and by recommending that client-facing servers use a linear scan to perform decompression. These requirements are detailed in Section 5.1.

11. IANA Considerations

11.1. Update of the TLS ExtensionType Registry

IANA is requested to create the following entries in the existing registry for ExtensionType (defined in [RFC8446]):

1. encrypted_client_hello(0xfe0d), with "TLS 1.3" column values set to "CH, HRR, EE", and "Recommended" column set to "Yes".
2. ech_outer_extensions(0xfd00), with the "TLS 1.3" column values set to "", and "Recommended" column set to "Yes".

11.2. Update of the TLS Alert Registry

IANA is requested to create an entry, ech_required(121) in the existing registry for Alerts (defined in [RFC8446]), with the "DTLS-OK" column set to "Y".

12. ECHConfig Extension Guidance

Any future information or hints that influence ClientHelloOuter SHOULD be specified as ECHConfig extensions. This is primarily because the outer ClientHello exists only in support of ECH. Namely, it is both an envelope for the encrypted inner ClientHello and enabler for authenticated key mismatch signals (see Section 7). In contrast, the inner ClientHello is the true ClientHello used upon ECH negotiation.

13. References

13.1. Normative References

- [HTTPS-RR] Schwartz, B., Bishop, M., and E. Nygren, "Service binding and parameter specification via the DNS (DNS SVCB and HTTPS RRs)", Work in Progress, Internet-Draft, draft-ietf-dnsop-svcb-https-08, 12 October 2021, <<https://www.ietf.org/archive/id/draft-ietf-dnsop-svcb-https-08.txt>>.
- [I-D.ietf-tls-exported-authenticator] Sullivan, N., "Exported Authenticators in TLS", Work in Progress, Internet-Draft, draft-ietf-tls-exported-authenticator-14, 25 January 2021, <<https://www.ietf.org/archive/id/draft-ietf-tls-exported-authenticator-14.txt>>.
- [I-D.irtf-cfrg-hpke] Barnes, R. L., Bhargavan, K., Lipp, B., and C. A. Wood, "Hybrid Public Key Encryption", Work in Progress, Internet-Draft, draft-irtf-cfrg-hpke-12, 2 September 2021, <<https://www.ietf.org/archive/id/draft-irtf-cfrg-hpke-12.txt>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <<https://www.rfc-editor.org/info/rfc5890>>.
- [RFC7918] Langley, A., Modadugu, N., and B. Moeller, "Transport Layer Security (TLS) False Start", RFC 7918, DOI 10.17487/RFC7918, August 2016, <<https://www.rfc-editor.org/info/rfc7918>>.
- [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>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

13.2. Informative References

- [I-D.kazuho-protected-sni]
Oku, K., "TLS Extensions for Protecting SNI", Work in Progress, Internet-Draft, draft-kazuho-protected-sni-00, 18 July 2017, <<https://www.ietf.org/archive/id/draft-kazuho-protected-sni-00.txt>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.
- [RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 5077, DOI 10.17487/RFC5077, January 2008, <<https://www.rfc-editor.org/info/rfc5077>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<https://www.rfc-editor.org/info/rfc7301>>.
- [RFC7858] Hu, Z., Zhu, L., Heidemann, J., Mankin, A., Wessels, D., and P. Hoffman, "Specification for DNS over Transport Layer Security (TLS)", RFC 7858, DOI 10.17487/RFC7858, May 2016, <<https://www.rfc-editor.org/info/rfc7858>>.
- [RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <<https://www.rfc-editor.org/info/rfc7924>>.
- [RFC8094] Reddy, T., Wing, D., and P. Patil, "DNS over Datagram Transport Layer Security (DTLS)", RFC 8094, DOI 10.17487/RFC8094, February 2017, <<https://www.rfc-editor.org/info/rfc8094>>.
- [RFC8484] Hoffman, P. and P. McManus, "DNS Queries over HTTPS (DoH)", RFC 8484, DOI 10.17487/RFC8484, October 2018, <<https://www.rfc-editor.org/info/rfc8484>>.
- [RFC8701] Benjamin, D., "Applying Generate Random Extensions And Sustain Extensibility (GREASE) to TLS Extensibility", RFC 8701, DOI 10.17487/RFC8701, January 2020, <<https://www.rfc-editor.org/info/rfc8701>>.

[RFC8744] Huitema, C., "Issues and Requirements for Server Name Identification (SNI) Encryption in TLS", RFC 8744, DOI 10.17487/RFC8744, July 2020, <<https://www.rfc-editor.org/info/rfc8744>>.

[WHATWG-IPV4] "URL Living Standard - IPv4 Parser", May 2021, <<https://url.spec.whatwg.org/#concept-ipv4-parser>>.

Appendix A. Alternative SNI Protection Designs

Alternative approaches to encrypted SNI may be implemented at the TLS or application layer. In this section we describe several alternatives and discuss drawbacks in comparison to the design in this document.

A.1. TLS-layer

A.1.1. TLS in Early Data

In this variant, TLS Client Hellos are tunneled within early data payloads belonging to outer TLS connections established with the client-facing server. This requires clients to have established a previous session --- and obtained PSKs --- with the server. The client-facing server decrypts early data payloads to uncover Client Hellos destined for the backend server, and forwards them onwards as necessary. Afterwards, all records to and from backend servers are forwarded by the client-facing server -- unmodified. This avoids double encryption of TLS records.

Problems with this approach are: (1) servers may not always be able to distinguish inner Client Hellos from legitimate application data, (2) nested 0-RTT data may not function correctly, (3) 0-RTT data may not be supported -- especially under DoS -- leading to availability concerns, and (4) clients must bootstrap tunnels (sessions), costing an additional round trip and potentially revealing the SNI during the initial connection. In contrast, encrypted SNI protects the SNI in a distinct Client Hello extension and neither abuses early data nor requires a bootstrapping connection.

A.1.2. Combined Tickets

In this variant, client-facing and backend servers coordinate to produce "combined tickets" that are consumable by both. Clients offer combined tickets to client-facing servers. The latter parse them to determine the correct backend server to which the Client Hello should be forwarded. This approach is problematic due to non-trivial coordination between client-facing and backend servers for

ticket construction and consumption. Moreover, it requires a bootstrapping step similar to that of the previous variant. In contrast, encrypted SNI requires no such coordination.

A.2. Application-layer

A.2.1. HTTP/2 CERTIFICATE Frames

In this variant, clients request secondary certificates with CERTIFICATE_REQUEST HTTP/2 frames after TLS connection completion. In response, servers supply certificates via TLS exported authenticators [I-D.ietf-tls-exported-authenticator] in CERTIFICATE frames. Clients use a generic SNI for the underlying client-facing server TLS connection. Problems with this approach include: (1) one additional round trip before peer authentication, (2) non-trivial application-layer dependencies and interaction, and (3) obtaining the generic SNI to bootstrap the connection. In contrast, encrypted SNI induces no additional round trip and operates below the application layer.

Appendix B. Linear-time Outer Extension Processing

The following procedure processes the "ech_outer_extensions" extension (see Section 5.1) in linear time, ensuring that each referenced extension in the ClientHelloOuter is included at most once:

1. Let I be zero and N be the number of extensions in ClientHelloOuter.
2. For each extension type, E, in OuterExtensions:
 - * If E is "encrypted_client_hello", abort the connection with an "illegal_parameter" alert and terminate this procedure.
 - * While I is less than N and the I-th extension of ClientHelloOuter does not have type E, increment I.
 - * If I is equal to N, abort the connection with an "illegal_parameter" alert and terminate this procedure.
 - * Otherwise, the I-th extension of ClientHelloOuter has type E. Copy it to the EncodedClientHelloInner and increment I.

Appendix C. Acknowledgements

This document draws extensively from ideas in [I-D.kazuho-protected-sni], but is a much more limited mechanism because it depends on the DNS for the protection of the ECH key. Richard Barnes, Christian Huitema, Patrick McManus, Matthew Prince, Nick Sullivan, Martin Thomson, and David Benjamin also provided important ideas and contributions.

Appendix D. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

Issue and pull request numbers are listed with a leading octothorp.

D.1. Since draft-ietf-tls-esni-12

- * Abort on duplicate OuterExtensions (#514)
- * Improve EncodedClientHelloInner definition (#503)
- * Clarify retry configuration usage (#498)
- * Expand on config_id generation implications (#491)
- * Server-side acceptance signal extension GREASE (#481)
- * Refactor overview, client implementation, and middlebox sections (#480, #478, #475, #508)
- * Editorial improvements (#485, #488, #490, #495, #496, #499, #500, #501, #504, #505, #507, #510, #511)

D.2. Since draft-ietf-tls-esni-11

- * Move ClientHello padding to the encoding (#443)
- * Align codepoints (#464)
- * Relax OuterExtensions checks for alignment with RFC8446 (#467)
- * Clarify HRR acceptance and rejection logic (#470)
- * Editorial improvements (#468, #465, #462, #461)

D.3. Since draft-ietf-tls-esni-10

- * Make HRR confirmation and ECH acceptance explicit (#422, #423)
- * Relax computation of the acceptance signal (#420, #449)
- * Simplify ClientHelloOuterAAD generation (#438, #442)
- * Allow empty enc in ECHClientHello (#444)
- * Authenticate ECHClientHello extensions position in ClientHelloOuterAAD (#410)
- * Allow clients to send a dummy PSK and early_data in ClientHelloOuter when applicable (#414, #415)
- * Compress ECHConfigContents (#409)
- * Validate ECHConfig.contents.public_name (#413, #456)
- * Validate ClientHelloInner contents (#411)
- * Note split-mode challenges for HRR (#418)
- * Editorial improvements (#428, #432, #439, #445, #458, #455)

D.4. Since draft-ietf-tls-esni-09

- * Finalize HPKE dependency (#390)
- * Move from client-computed to server-chosen, one-byte config identifier (#376, #381)
- * Rename ECHConfigs to ECHConfigList (#391)
- * Clarify some security and privacy properties (#385, #383)

Authors' Addresses

Eric Rescorla
RTFM, Inc.

Email: ekr@rtfm.com

Kazuho Oku
Fastly

Email: kazuhooku@gmail.com

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 10 November 2022

R. Barnes
Cisco
S. Iyengar
Facebook
N. Sullivan
Cloudflare
E. Rescorla
Mozilla
9 May 2022

Delegated Credentials for (D)TLS
draft-ietf-tls-subcerts-13

Abstract

The organizational separation between operators of TLS and DTLS endpoints and the certification authority can create limitations. For example, the lifetime of certificates, how they may be used, and the algorithms they support are ultimately determined by the certification authority. This document describes a mechanism to to overcome some of these limitations by enabling operators to delegate their own credentials for use in TLS and DTLS without breaking compatibility with peers that do not support this specification.

Discussion Venues

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

Source for this draft and an issue tracker can be found at <https://github.com/tlswg/tls-subcerts> (<https://github.com/tlswg/tls-subcerts>).

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 10 November 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document.

Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

- 1. Introduction
- 2. Conventions and Terminology
 - 2.1. Change Log
- 3. Solution Overview
 - 3.1. Rationale
 - 3.2. Related Work
- 4. Delegated Credentials
 - 4.1. Client and Server Behavior
 - 4.1.1. Server Authentication
 - 4.1.2. Client Authentication
 - 4.1.3. Validating a Delegated Credential
 - 4.2. Certificate Requirements
- 5. Operational Considerations
 - 5.1. Client Clock Skew
- 6. IANA Considerations
- 7. Security Considerations
 - 7.1. Security of Delegated Credential's Private Key
 - 7.2. Re-use of Delegated Credentials in Multiple Contexts
 - 7.3. Revocation of Delegated Credentials
 - 7.4. Interactions with Session Resumption
 - 7.5. Privacy Considerations
 - 7.6. The Impact of Signature Forgery Attacks
- 8. Acknowledgements
- 9. References
 - 9.1. Normative References
 - 9.2. Informative References
- Appendix A. ASN.1 Module
- Appendix B. Example Certificate
- Authors' Addresses

1. Introduction

Server operators often deploy (D)TLS termination services in locations such as remote data centers or Content Delivery Networks (CDNs) where it may be difficult to detect compromises of private key material corresponding to TLS certificates. Short-lived certificates may be used to limit the exposure of keys in these cases.

However, short-lived certificates need to be renewed more frequently than long-lived certificates. If an external Certification Authority (CA) is unable to issue a certificate in time to replace a deployed certificate, the server would no longer be able to present a valid certificate to clients. With short-lived certificates, there is a smaller window of time to renew a certificates and therefore a higher risk that an outage at a CA will negatively affect the uptime of the TLS-fronted service.

Typically, a (D)TLS server uses a certificate provided by some entity other than the operator of the server (a CA) [RFC8446] [RFC5280]. This organizational separation makes the (D)TLS server operator dependent on the CA for some aspects of its operations, for example:

- * Whenever the server operator wants to deploy a new certificate, it has to interact with the CA.

- * The CA might only issue credentials containing certain types of public key, which can limit the set of (D)TLS signature schemes usable by the server operator.

To reduce the dependency on external CAs, this document specifies a limited delegation mechanism that allows a (D)TLS peer to issue its own credentials within the scope of a certificate issued by an external CA. These credentials only enable the recipient of the delegation to speak for names that the CA has authorized. Furthermore, this mechanism allows the server to use modern signature algorithms such as Ed25519 [RFC8032] even if their CA does not support them.

This document refers to the certificate issued by the CA as a "certificate", or "delegation certificate", and the one issued by the operator as a "delegated credential" or "DC".

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.

2.1. Change Log

RFC EDITOR PLEASE DELETE THIS SECTION.

(*) indicates changes to the wire protocol.

draft-11

- * Editorial changes based on AD comments
- * Add support for DTLs
- * Address address ambiguity in cert expiry

draft-10

- * Address superficial comments
- * Add example certificate

draft-09

- * Address case nits
- * Fix section bullets in 4.1.3.
- * Add operational considerations section for clock skew
- * Add text around using an oracle to forge DCs in the future and past
- * Add text about certificate extension vs ECU

draft-08

- * Include details about the impact of signature forgery attacks

- * Copy edits
- * Fix section about DC reuse
- * Incorporate feedback from Jonathan Hammell and Kevin Jacobs on the list

draft-07

- * Minor text improvements

draft-06

- * Modified IANA section, fixed nits

draft-05

- * Removed support for PKCS 1.5 RSA signature algorithms.
- * Additional security considerations.

draft-04

- * Add support for client certificates.

draft-03

- * Remove protocol version from the Credential structure. (*)

draft-02

- * Change public key type. (*)
- * Change DelegationUsage extension to be NULL and define its object identifier.
- * Drop support for TLS 1.2.
- * Add the protocol version and credential signature algorithm to the Credential structure. (*)
- * Specify undefined behavior in a few cases: when the client receives a DC without indicated support; when the client indicates the extension in an invalid protocol version; and when DCs are sent as extensions to certificates other than the end-entity certificate.

3. Solution Overview

A delegated credential (DC) is a digitally signed data structure with two semantic fields: a validity interval and a public key (along with its associated signature algorithm). The signature on the delegated credential indicates a delegation from the certificate that is issued to the peer. The private key used to sign a credential corresponds to the public key of the peer's X.509 end-entity certificate [RFC5280].

A (D)TLS handshake that uses delegated credentials differs from a standard handshake in a few important ways:

- * The initiating peer provides an extension in its ClientHello or

CertificateRequest that indicates support for this mechanism.

- * The peer sending the Certificate message provides both the certificate chain terminating in its certificate as well as the delegated credential.
- * The initiator uses information from the peer's certificate to verify the delegated credential and that the peer is asserting an expected identity, determining an authentication result for the peer.
- * Peers accepting the delegated credential use it as the certificate key for the (D)TLS handshake.

As detailed in Section 4, the delegated credential is cryptographically bound to the end-entity certificate with which the credential may be used. This document specifies the use of delegated credentials in (D)TLS 1.3 or later; their use in prior versions of the protocol is not allowed.

Delegated credentials allow a peer to terminate (D)TLS connections on behalf of the certificate owner. If a credential is stolen, there is no mechanism for revoking it without revoking the certificate itself. To limit exposure in case of the compromise of a delegated credential's private key, delegated credentials have a maximum validity period. In the absence of an application profile standard specifying otherwise, the maximum validity period is set to 7 days. Peers MUST NOT issue credentials with a validity period longer than the maximum validity period or that extends beyond the validity period of the delegation certificate. This mechanism is described in detail in Section 4.1.

It was noted in [XPROT] that certificates in use by servers that support outdated protocols such as SSLv2 can be used to forge signatures for certificates that contain the keyEncipherment KeyUsage ([RFC5280] section 4.2.1.3). In order to reduce the risk of cross-protocol attacks on certificates that are not intended to be used with DC-capable TLS stacks, we define a new DelegationUsage extension to X.509 that permits use of delegated credentials. (See Section 4.2.)

3.1. Rationale

Delegated credentials present a better alternative than other delegation mechanisms like proxy certificates [RFC3820] for several reasons:

- * There is no change needed to certificate validation at the PKI layer.
- * X.509 semantics are very rich. This can cause unintended consequences if a service owner creates a proxy certificate where the properties differ from the leaf certificate. Proxy certificates can be useful in controlled environments, but remain a risk in scenarios where the additional flexibility they provide is not necessary. For this reason, delegated credentials have very restricted semantics that should not conflict with X.509 semantics.
- * Proxy certificates rely on the certificate path building process to establish a binding between the proxy certificate and the end-entity certificate. Since the certificate path building process

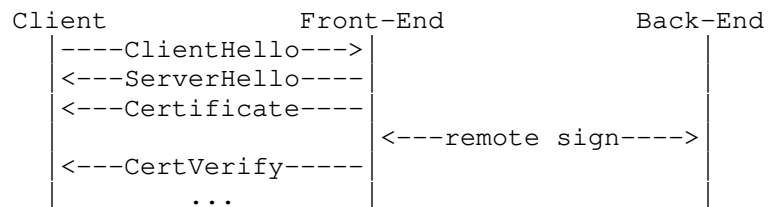
is not cryptographically protected, it is possible that a proxy certificate could be bound to another certificate with the same public key, with different X.509 parameters. Delegated credentials, which rely on a cryptographic binding between the entire certificate and the delegated credential, cannot.

- * Each delegated credential is bound to a specific signature algorithm for use in the (D)TLS handshake ([RFC8446] section 4.2.3). This prevents them from being used with other, perhaps unintended, signature algorithms. The signature algorithm bound to the delegated credential can be chosen independently of the set of signature algorithms supported by the end-entity certificate.

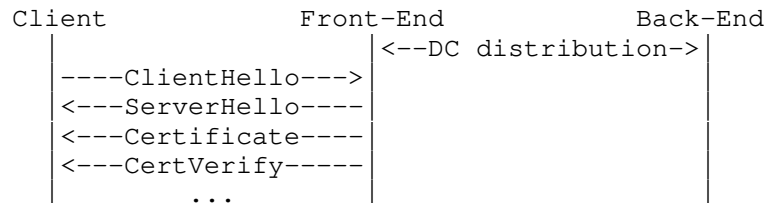
3.2. Related Work

Many of the use cases for delegated credentials can also be addressed using purely server-side mechanisms that do not require changes to client behavior (e.g., a PKCS#11 interface or a remote signing mechanism, [KEYLESS] being one example). These mechanisms, however, incur per-transaction latency, since the front-end server has to interact with a back-end server that holds a private key. The mechanism proposed in this document allows the delegation to be done off-line, with no per-transaction latency. The figure below compares the message flows for these two mechanisms with (D)TLS 1.3 [RFC8446] [I-D.ietf-tls-dtls13].

Remote key signing:



Delegated Credential:



These two mechanisms can be complementary. A server could use delegated credentials for clients that support them, while using a server-side mechanism to support legacy clients. Both mechanisms require a trusted relationship between the Front-End and Back-End -- the delegated credential can be used in place of a certificate private key.

Use of short-lived certificates with automated certificate issuance, e.g., with Automated Certificate Management Environment (ACME) [RFC8555], reduces the risk of key compromise, but has several limitations. Specifically, it introduces an operationally-critical dependency on an external party (the CA). It also limits the types of algorithms supported for (D)TLS authentication to those the CA is willing to issue a certificate for. Nonetheless, existing automated issuance APIs like ACME may be useful for provisioning delegated

credentials.

4. Delegated Credentials

While X.509 forbids end-entity certificates from being used as issuers for other certificates, it is valid to use them to issue other signed objects as long as the certificate contains the digitalSignature KeyUsage ([RFC5280] section 4.2.1.3). (All certificates compatible with TLS 1.3 are required to contain the digitalSignature KeyUsage.) We define a new signed object format that would encode only the semantics that are needed for this application. The Credential has the following structure:

```
struct {
    uint32 valid_time;
    SignatureScheme expected_cert_verify_algorithm;
    opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;
} Credential;
```

valid_time: Time, in seconds relative to the delegation certificate's notBefore value, after which the delegated credential is no longer valid. Endpoints will reject delegated credentials that expire more than 7 days from the current time (as described in Section 4.1) based on the default (see Section 3).

expected_cert_verify_algorithm: The signature algorithm of the Credential key pair, where the type SignatureScheme is as defined in [RFC8446]. This is expected to be the same as the sender's CertificateVerify.algorithm (as described in Section 4.1.3). Only signature algorithms allowed for use in CertificateVerify messages are allowed. When using RSA, the public key MUST NOT use the rsaEncryption OID. As a result, the following algorithms are not allowed for use with delegated credentials: rsa_pss_rsae_sha256, rsa_pss_rsae_sha384, rsa_pss_rsae_sha512.

ASN1_subjectPublicKeyInfo: The Credential's public key, a DER-encoded [X.690] SubjectPublicKeyInfo as defined in [RFC5280].

The DelegatedCredential has the following structure:

```
struct {
    Credential cred;
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} DelegatedCredential;
```

cred: The Credential structure as previously defined.

algorithm: The signature algorithm used to verify DelegatedCredential.signature.

signature: The delegation, a signature that binds the credential to the end-entity certificate's public key as specified below. The signature scheme is specified by DelegatedCredential.algorithm.

The signature of the DelegatedCredential is computed over the concatenation of:

1. A string that consists of octet 32 (0x20) repeated 64 times.
2. The context string "TLS, server delegated credentials" for server authentication and "TLS, client delegated credentials" for client

authentication.

3. A single 0 byte, which serves as the separator.
4. The DER-encoded X.509 end-entity certificate used to sign the DelegatedCredential.
5. DelegatedCredential.cred.
6. DelegatedCredential.algorithm.

The signature is computed by using the private key of the peer's end-entity certificate, with the algorithm indicated by DelegatedCredential.algorithm.

The signature effectively binds the credential to the parameters of the handshake in which it is used. In particular, it ensures that credentials are only used with the certificate and signature algorithm chosen by the delegator.

The code changes required in order to create and verify delegated credentials, and the implementation complexity this entails, are localized to the (D)TLS stack. This has the advantage of avoiding changes to the often-delicate security-critical PKI code.

4.1. Client and Server Behavior

This document defines the following (D)TLS extension code point.

```
enum {  
    ...  
    delegated_credential(34),  
    (65535)  
} ExtensionType;
```

4.1.1. Server Authentication

A client which supports this specification SHALL send a "delegated_credential" extension in its ClientHello. The body of the extension consists of a SignatureSchemeList (defined in [RFC8446]):

```
struct {  
    SignatureScheme supported_signature_algorithm<2..2^16-2>;  
} SignatureSchemeList;
```

If the client receives a delegated credential without having indicated support in its ClientHello, then the client MUST abort the handshake with an "unexpected_message" alert.

If the extension is present, the server MAY send a delegated credential; if the extension is not present, the server MUST NOT send a delegated credential. The server MUST ignore the extension unless (D)TLS 1.3 or a later version is negotiated. An example of when a server could choose not to send a delegated credential is when the SignatureSchemes listed only contain signature schemes for which a corresponding delegated credential does not exist or are otherwise unsuitable for the connection.

The server MUST send the delegated credential as an extension in the CertificateEntry of its end-entity certificate; the client SHOULD ignore delegated credentials sent as extensions to any other certificate.

The algorithm field MUST be of a type advertised by the client in the "signature_algorithms" extension of the ClientHello message and the expected_cert_verify_algorithm field MUST be of a type advertised by the client in the SignatureSchemeList and is considered invalid otherwise. Clients that receive invalid delegated credentials MUST terminate the connection with an "illegal_parameter" alert.

4.1.2. Client Authentication

A server that supports this specification SHALL send a "delegated_credential" extension in the CertificateRequest message when requesting client authentication. The body of the extension consists of a SignatureSchemeList. If the server receives a delegated credential without having indicated support in its CertificateRequest, then the server MUST abort with an "unexpected_message" alert.

If the extension is present, the client MAY send a delegated credential; if the extension is not present, the client MUST NOT send a delegated credential. The client MUST ignore the extension unless (D)TLS 1.3 or a later version is negotiated.

The client MUST send the delegated credential as an extension in the CertificateEntry of its end-entity certificate; the server SHOULD ignore delegated credentials sent as extensions to any other certificate.

The algorithm field MUST be of a type advertised by the server in the "signature_algorithms" extension of the CertificateRequest message and the expected_cert_verify_algorithm field MUST be of a type advertised by the server in the SignatureSchemeList and is considered invalid otherwise. Servers that receive invalid delegated credentials MUST terminate the connection with an "illegal_parameter" alert.

4.1.3. Validating a Delegated Credential

On receiving a delegated credential and certificate chain, the peer validates the certificate chain and matches the end-entity certificate to the peer's expected identity in the same way that it is done when delegated credentials are not in use. It then performs the following checks with expiry time set to the delegation certificate's notBefore value plus DelegatedCredential.cred.valid_time:

1. Verify that the current time is within the validity interval of the credential. This is done by asserting that the current time does not exceed the expiry time. (The start time of the credential is implicitly validated as part of certificate validation.)
2. Verify that the delegated credential's remaining validity period is no more than the maximum validity period. This is done by asserting that the expiry time does not exceed the current time plus the maximum validity period (7 days by default).
3. Verify that expected_cert_verify_algorithm matches the scheme indicated in the peer's CertificateVerify message and that the algorithm is allowed for use with delegated credentials.
4. Verify that the end-entity certificate satisfies the conditions

in Section 4.2.

5. Use the public key in the peer's end-entity certificate to verify the signature of the credential using the algorithm indicated by `DelegatedCredential.algorithm`.

If one or more of these checks fail, then the delegated credential is deemed invalid. Clients and servers that receive invalid delegated credentials MUST terminate the connection with an "illegal_parameter" alert.

If successful, the participant receiving the Certificate message uses the public key in `DelegatedCredential.cred` to verify the signature in the peer's `CertificateVerify` message.

4.2. Certificate Requirements

This document defines a new X.509 extension, `DelegationUsage`, to be used in the certificate when the certificate permits the usage of delegated credentials. What follows is the ASN.1 [X.680] for the `DelegationUsage` certificate extension.

```
ext-delegationUsage EXTENSION ::= {  
    SYNTAX DelegationUsage IDENTIFIED BY id-pe-delegationUsage  
}
```

```
DelegationUsage ::= NULL
```

```
id-pe-delegationUsage OBJECT IDENTIFIER ::= {  
    { iso(1) identified-organization(3) dod(6) internet(1)  
      private(4) enterprise(1) id-cloudflare(44363) 44 }  
}
```

The extension MUST be marked non-critical. (See Section 4.2 of [RFC5280].) An endpoint MUST NOT accept a delegated credential unless the peer's end-entity certificate satisfies the following criteria:

- * It has the `DelegationUsage` extension.
- * It has the `digitalSignature` `KeyUsage` (see the `KeyUsage` extension defined in [RFC5280]).

A new extension was chosen instead of adding a new Extended Key Usage (EKU) to be compatible with deployed (D)TLS and PKI software stacks without requiring CAs to issue new intermediate certificates.

5. Operational Considerations

The operational consideration documented in this section should be taken into consideration when using Delegated Certificates.

5.1. Client Clock Skew

One of the risks of deploying a short-lived credential system based on absolute time is client clock skew. If a client's clock is sufficiently ahead or behind of the server's clock, then clients will reject delegated credentials that are valid from the server's perspective. Clock skew also affects the validity of the original certificates. The lifetime of the delegated credential should be set taking clock skew into account. Clock skew may affect a delegated credential at the beginning and end of its validity periods, which should also be taken into account.

6. IANA Considerations

This document registers the "delegated_credential" extension in the "TLS ExtensionType Values" registry. The "delegated_credential" extension has been assigned a code point of 34. The IANA registry lists this extension as "Recommended" (i.e., "Y") and indicates that it may appear in the ClientHello (CH), CertificateRequest (CR), or Certificate (CT) messages in (D)TLS 1.3 [RFC8446] [I-D.ietf-tls-dtls13]. Additionally, the "DTLS-Only" column is assigned the value "N".

This document also defines an ASN.1 module for the DelegationUsage certificate extension in Appendix A. IANA has registered value 95 for "id-mod-delegated-credential-extn" in the "SMI Security for PKIX Module Identifier" (1.3.5.1.5.5.7.0) registry. An OID for the DelegationUsage certificate extension is not needed as it is already assigned to the extension from Cloudflare's IANA Private Enterprise Number (PEN) arc.

7. Security Considerations

The security consideration documented in this section should be taken into consideration when using Delegated Certificates.

7.1. Security of Delegated Credential's Private Key

Delegated credentials limit the exposure of the private key used in a (D)TLS connection by limiting its validity period. An attacker who compromises the private key of a delegated credential can impersonate the compromised party in new TLS connections until the delegated credential expires.

However, they cannot create new delegated credentials. Thus, delegated credentials should not be used to send a delegation to an untrusted party, but are meant to be used between parties that have some trust relationship with each other. The secrecy of the delegated credential's private key is thus important and access control mechanisms SHOULD be used to protect it, including file system controls, physical security, or hardware security modules.

7.2. Re-use of Delegated Credentials in Multiple Contexts

It is not possible to use the same delegated credential for both client and server authentication because issuing parties compute the corresponding signature using a context string unique to the intended role (client or server).

7.3. Revocation of Delegated Credentials

Delegated credentials do not provide any additional form of early revocation. Since it is short lived, the expiry of the delegated credential revokes the credential. Revocation of the long term private key that signs the delegated credential (from the end-entity certificate) also implicitly revokes the delegated credential.

7.4. Interactions with Session Resumption

If a peer decides to cache the certificate chain and re-validate it when resuming a connection, they SHOULD also cache the associated delegated credential and re-validate it. Failing to do so may result in resuming connections for which the DC has expired.

7.5. Privacy Considerations

Delegated credentials can be valid for 7 days (by default) and it is much easier for a service to create delegated credentials than a certificate signed by a CA. A service could determine the client time and clock skew by creating several delegated credentials with different expiry timestamps and observing whether the client would accept it. Client time could be unique and thus privacy sensitive clients, such as browsers in incognito mode, who do not trust the service might not want to advertise support for delegated credentials or limit the number of probes that a server can perform.

7.6. The Impact of Signature Forgery Attacks

Delegated credentials are only used in (D)TLS 1.3 connections. However, the certificate that signs a delegated credential may be used in other contexts such as (D)TLS 1.2. Using a certificate in multiple contexts opens up a potential cross-protocol attack against delegated credentials in (D)TLS 1.3.

When (D)TLS 1.2 servers support RSA key exchange, they may be vulnerable to attacks that allow forging an RSA signature over an arbitrary message [BLEI]. TLS 1.2 [RFC5246] (Section 7.4.7.1.) describes a mitigation strategy requiring careful implementation of timing resistant countermeasures for preventing these attacks. Experience shows that in practice, server implementations may fail to fully stop these attacks due to the complexity of this mitigation [ROBOT]. For (D)TLS 1.2 servers that support RSA key exchange using a DC-enabled end-entity certificate, a hypothetical signature forgery attack would allow forging a signature over a delegated credential. The forged delegated credential could then be used by the attacker as the equivalent of a man-in-the-middle certificate, valid for a maximum of 7 days (if the default `valid_time` is used).

Server operators should therefore minimize the risk of using DC-enabled end-entity certificates where a signature forgery oracle may be present. If possible, server operators may choose to use DC-enabled certificates only for signing credentials, and not for serving non-DC (D)TLS traffic. Furthermore, server operators may use elliptic curve certificates for DC-enabled traffic, while using RSA certificates without the `DelegationUsage` certificate extension for non-DC traffic; this completely prevents such attacks.

Note that if a signature can be forged over an arbitrary credential, the attacker can choose any value for the `valid_time` field. Repeated signature forgeries therefore allow the attacker to create multiple delegated credentials that can cover the entire validity period of the certificate. Temporary exposure of the key or a signing oracle may allow the attacker to impersonate a server for the lifetime of the certificate.

8. Acknowledgements

Thanks to David Benjamin, Christopher Patton, Kyle Nekritz, Anirudh Ramachandran, Benjamin Kaduk, Kazuho Oku, Daniel Kahn Gillmor, Watson Ladd, Robert Merget, Juraj Somorovsky, Nimrod Aviram for their discussions, ideas, and bugs they have found.

9. References

9.1. Normative References

- [I-D.ietf-tls-dtls13] Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-dtls13-43, 30 April 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-tls-dtls13-43>>.
- [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>>.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/rfc/rfc5280>>.
- [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>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/rfc/rfc8446>>.
- [X.680] ITU-T, "Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation", ISO/IEC 8824-1:2015, November 2015.
- [X.690] ITU-T, "Information technology - ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ISO/IEC 8825-1:2015, November 2015.

9.2. Informative References

- [BLEI] Bleichenbacher, D., "Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1", Advances in Cryptology -- CRYPTO'98, LNCS vol. 1462, pages: 1-12 , 1998.
- [KEYLESS] Sullivan, N. and D. Stebila, "An Analysis of TLS Handshake Proxying", IEEE Trustcom/BigDataSE/ISPA 2015 , 2015.
- [RFC3820] Tuecke, S., Welch, V., Engert, D., Pearlman, L., and M. Thompson, "Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile", RFC 3820, DOI 10.17487/RFC3820, June 2004, <<https://www.rfc-editor.org/rfc/rfc3820>>.
- [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/rfc/rfc5246>>.
- [RFC5912] Hoffman, P. and J. Schaad, "New ASN.1 Modules for the Public Key Infrastructure Using X.509 (PKIX)", RFC 5912, DOI 10.17487/RFC5912, June 2010, <<https://www.rfc-editor.org/rfc/rfc5912>>.

- [RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <<https://www.rfc-editor.org/rfc/rfc8032>>.
- [RFC8555] Barnes, R., Hoffman-Andrews, J., McCarney, D., and J. Kasten, "Automatic Certificate Management Environment (ACME)", RFC 8555, DOI 10.17487/RFC8555, March 2019, <<https://www.rfc-editor.org/rfc/rfc8555>>.
- [ROBOT] Boeck, H., Somorovsky, J., and C. Young, "Return Of Bleichenbacher's Oracle Threat (ROBOT)", 27th USENIX Security Symposium , 2018.
- [XPROT] Jager, T., Schwenk, J., and J. Somorovsky, "On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption", Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security , 2015.

Appendix A. ASN.1 Module

The following ASN.1 module provides the complete definition of the DelegationUsage certificate extension. The ASN.1 module makes imports from [RFC5912].

```

DelegatedCredentialExtn
{ iso(1) identified-organization(3) dod(6) internet(1)
  security(5) mechanisms(5) pkix(7) id-mod(0)
  id-mod-delegated-credential-extn(95) }

DEFINITIONS IMPLICIT TAGS ::=
BEGIN

-- EXPORT ALL

IMPORTS

EXTENSION
FROM PKIX-CommonTypes-2009 -- From RFC 5912
{ iso(1) identified-organization(3) dod(6) internet(1)
  security(5) mechanisms(5) pkix(7) id-mod(0)
  id-mod-pkixCommon-02(57) } ;

-- OID

id-cloudflare OBJECT IDENTIFIER ::=
{ iso(1) identified-organization(3) dod(6) internet(1) private(4)
  enterprise(1) 44363 }

-- EXTENSION

ext-delegationUsage EXTENSION ::=
{ SYNTAX DelegationUsage
  IDENTIFIED BY id-pe-delegationUsage }

id-pe-delegationUsage OBJECT IDENTIFIER ::= { id-cloudflare 44 }

DelegationUsage ::= NULL

END

```

Appendix B. Example Certificate

The following certificate has the Delegated Credentials OID.

-----BEGIN CERTIFICATE-----
MIIFRjCCBMugAwIBAgIQDGeVB+ly0o/OeCHFSJ6YnTAKBggqhkJOPQQDAzBMMQsw
CQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMSYwJAYDVQQDEx1EaWdp
Q2VydCBFQMgU2VjdXJlIFN1cnZlcjBDQTAEfw0xOTAzMjYwMDAwMDBaFw0yMTAz
MzAxMjAwdDBaMGoxCzAJBgNVBAYTA1VTMRMwEQYDVQQIEwpDYWxpZm9ybmlhMRYw
FAYDVQQHEw1TYW4gRnJhbmcNpc2NvMRkwFwYDVQQKEizXBDbG91ZGZsYXJ1LCBJbmRM
MRMwEQYDVQDDQEWprYzJrZG0uY29tMFkwEYHKOIExj0CAQYIKoZIzj0DAQcDQgAE
d4azt83Bw0fcpGfoeyZpZznwGuxjbvg++wze0zAj8vniUkKxOWSQuIGNLn+xlLWqL
lw9djRNrlmVmM2gb9GgdKOCA28bjwgNrMB8GA1UdIwQYMBaAFKOD5h/52jlPwG7o
kcuvpd0x4gqfMB0GA1UdDgQWBBSfcb7fs3fUFAYB9lfRcwDPtgTjJaJBgNVHREE
HDAaggprYzJrZG0uY29tggwqlmtjmmtkbS5jb20wDgYDVR0PAQH/BAQDAgeAMB0G
AlUdJQQWMBQGCSGAQUFBwMBBggrBgEFBQcDAjBpBgNVHR8EYjBgMC6gLKAqhiho
dHRwOi8vY3JsMy5kaWdpY2VydC5jb20vc3NjYS1lY2MtZzEuY3JsMC6gLKAqhiho
dHRwOi8vY3JsNC5kaWdpY2VydC5jb20vc3NjYS1lY2MtZzEuY3JsMEwGA1UdIARF
MEMwNwYJYIZIAyB9baEBMcowKAYIKwYBBQUHAQEWHGH0dHBzOi8vd3d3LmRpZ2lj
ZXJ0LmNvbS9DUFMwCAYGZ4EMAQICMHSGCCSGAQUFBwEBBG8wbTakBggrBgEFBQcw
AYYYaHR0cDovL29jc3AuZGlnaWNlcnQuY29tMEUGCCSGAQUFBzACHjlodHRwOi8v
Y2FjZXJ0cy5kaWdpY2VydC5jb20vRGlnaUNlcnRFQ0NTZWNNcmVTXXZjZXJDSQ5j
cnQwDAYDVROTAQH/BAlwADAPBgkrBgEEAYLaSywEAQUMIBfgyKKwYBBAHWQIE
AgSCAW4EggFqAGwAdgc72d+8H4pxtZOUI5eqknthOFevCqtS6BqQlmQ2jh7RhQA
AWm5hyJ5AAAEawBHMEUCICIgfq+hStHRL2m8H0awoDR8OpnEHnkF0nI6nL5yYL/j
AiEAXwebGs/T6EsOYarPzoQRjrvZqk+sHH/t+jrSrKd5TDjcAdgCHdb/nWXz4jEOZ
X73zbv9WjUdWNv9KtWDBtor/XqCDDwAAAWm5hYNgAAAEawBHMEUCIQD9OWA8KGL6
bxDKfgileHJWB0iWieRs88VgJyfag/aFDgIgQ/OsdSF9XOylfoqqe0DTDM2Fexuw
0JR0AGZWxonTjzMAdgBELGUusO7Or8RAB9io/iJa2uaCvtjLMbU/0zOWtbabQAAA
AWm5hYHgAAAEawBHMEUCIQc4vua1n3BqthEqPa/VBTcsNwMtAwpCuac2IHj9wx6X/
AIgb+o00k28JQo9TMpp4vzJ3BD3HXWSnc2Zizbq7mkUQYMwCgYIKoZIzj0EAwMD
AQAwZgIXAJsx7d0SAu8ddf/m7IWfnfs3MQfJyGkEezMJX1t6sRso5z50SS12LPXe
muGalFE2ZgIXAL+CUDF5pz7mhrAEijQ1Mqlpf9tH40dJGvYZZQ3W23cmZSkdfvlty
5S4RfWHIIPjbw==
-----END CERTIFICATE-----

Authors' Addresses

Richard Barnes
Cisco
Email: rlb@ipv.sx

Subodh Iyengar
Facebook
Email: subodh@fb.com

Nick Sullivan
Cloudflare
Email: nick@cloudflare.com

Eric Rescorla
Mozilla
Email: ekr@rtfm.com

Internet Engineering Task Force
Internet-Draft
Updates: 5246 7525 (if approved)
Intended status: Standards Track
Expires: November 30, 2019

L. Velvindron
cyberstorm.mu
K. Moriarty
Dell EMC
A. Ghedini
Cloudflare Inc.
May 29, 2019

Deprecating MD5 and SHA-1 signature hashes in TLS 1.2
draft-lvelvindron-tls-md5-sha1-deprecate-05

Abstract

The MD5 and SHA-1 hashing algorithms are steadily weakening in strength and their deprecation process should begin for their use in TLS 1.2 digital signatures. However, this document does not deprecate SHA-1 in HMAC for record protection.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on November 30, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Requirements Language	2
2. Signature Algorithms	2
3. Certificate Request	3
4. Server Key Exchange	3
5. Certificate Verify	3
6. Updates to RFC5246	3
7. Updates to RFC7525	3
8. Security Considerations	4
9. Acknowledgement	4
10. References	4
10.1. Normative References	4
10.2. Informative References	4
Authors' Addresses	5

1. Introduction

The usage of MD5 and SHA-1 for signature hashing in TLS 1.2 is specified in RFC 5246 [RFC5246]. MD5 and SHA-1 have been proven to be insecure, subject to collision attacks. RFC 6151 [RFC6151] details the security considerations, including collision attacks for MD5, published in 2011. NIST formally deprecated use of SHA-1 in 2011 [NISTSP800-131A-R2] and disallowed its use for digital signatures at the end of 2013, based on both the Wang, et. al, attack and the potential for brute-force attack. Further, in 2017, researchers from Google and CWI Amsterdam [SHA-1-Collision] proved SHA-1 collision attacks were practical. This document updates RFC 5246 [RFC5246] and RFC7525 [RFC7525] in such a way that MD5 and SHA1 MUST NOT be used for digital signatures. However, this document does not deprecate SHA-1 in HMAC for record protection.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Signature Algorithms

Clients SHOULD NOT include MD5 and SHA-1 in signature_algorithms extension. If a client does not send a signature_algorithms extension, then the server MUST abort the handshake and send a handshake_failure alert.

3. Certificate Request

Servers SHOULD NOT include MD5 and SHA-1 in CertificateRequest message.

4. Server Key Exchange

Servers MUST NOT include MD5 and SHA-1 in ServerKeyExchange message. If client does receive a MD5 or SHA-1 signature in the ServerKeyExchange message it MUST abort the connection with handshake_failure or insufficient_security alert.

5. Certificate Verify

Clients MUST NOT include MD5 and SHA-1 in CertificateVerify message.

6. Updates to RFC5246

OLD:

In Section 7.4.1.4.1: the text should be revised from " Note: this is a change from TLS 1.1 where there are no explicit rules, but as a practical matter one can assume that the peer supports MD5 and SHA-1."

NEW:

"Note: This is a change from TLS 1.1 where there are no explicit rules, but as a practical matter one can assume that the peer supports SHA-256."

7. Updates to RFC7525

RFC7525 [RFC7525], Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) recommends use of SHA-256 as a minimum requirement. This update moves the minimum recommendation to use stronger language deprecating use of both SHA-1 and MD5. The prior text did not explicitly include MD5 and this text adds it to ensure it is understood as having been deprecated.

Section 4.3:

OLD:

When using RSA, servers SHOULD authenticate using certificates with at least a 2048-bit modulus for the public key. In addition, the use of the SHA-256 hash algorithm is RECOMMENDED (see [CAB-Baseline] for

more details). Clients SHOULD indicate to servers that they request SHA-256, by using the "Signature Algorithms" extension defined in TLS 1.2.

NEW:

servers SHOULD authenticate using certificates with at least a 2048-bit modulus for the public key.

In addition, the use of the SHA-256 hash algorithm is RECOMMENDED, SHA-1 or MD5 MUST not be used (see [CAB-Baseline] for more details). Clients MUST indicate to servers that they request SHA-256, by using the "Signature Algorithms" extension defined in TLS 1.2.

8. Security Considerations

Concerns with TLS 1.2 implementations falling back to SHA-1 is an issue. This draft updates the TLS 1.2 specification to deprecate support for MD5 and SHA-1 for digital signatures. However, this document does not deprecate SHA-1 in HMAC for record protection.

9. Acknowledgement

The authors would like to thank Hubert Kario for his help in writing the initial draft. We are also grateful to Daniel Migault, Martin Thomson and David Cooper for their feedback.

10. References

10.1. Normative References

- [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>>.
- [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>>.

10.2. Informative References

- [CAB-Baseline] CA/Browser Forum, "Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates Version 1.1.6", 2013, <<https://www.cabforum.org/documents.html>>.

[NISTSP800-131A-R2]

Barker, E. and A. Roginsky, "Transitioning the Use of Cryptographic Algorithms and Key Lengths", March 2019, <<https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar2.pdf>>.

[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>>.

[RFC6151] Turner, S. and L. Chen, "Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms", RFC 6151, DOI 10.17487/RFC6151, March 2011, <<https://www.rfc-editor.org/info/rfc6151>>.

[SHA-1-Collision]

Stevens, M., Bursztein, E., Karpman, P., Albertini, A., and Y. Markov, "The first collision for full SHA-1", March 2019, <<http://shattered.io/static/shattered.pdf>>.

Authors' Addresses

Loganaden Velvindron
cyberstorm.mu
Rose Hill
MU

Phone: +230 59762817
Email: logan@cyberstorm.mu

Kathleen Moriarty
Dell EMC

Alessandro Ghedini
Cloudflare Inc.

TLS
Internet-Draft
Intended status: Standards Track
Expires: January 23, 2020

Y. Nir
Dell EMC
July 22, 2019

A Flags Extension for TLS 1.3
draft-nir-tls-tlsflags-02

Abstract

A number of extensions are proposed in the TLS working group that carry no interesting information except the 1-bit indication that a certain optional feature is supported. Such extensions take 4 octets each. This document defines a flags extension that can provide such indications at an average marginal cost of 1 bit each. More precisely, it provides as many flag extensions as needed at $4 + \frac{\text{order of the last set bit}}{8}$.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 23, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Requirements and Other Notation	2
2. The <code>tls_flags</code> Extension	3
3. IANA Considerations	4
4. Security Considerations	5
5. Acknowledgements	5
6. References	5
6.1. Normative References	5
6.2. Informative References	5
Appendix A. Change Log	6
Author's Address	6

1. Introduction

Since the publication of TLS 1.3 ([RFC8446]) there have been several proposals for extensions to this protocol, where the presence of the content-free extension in both the ClientHello and either the ServerHello or EncryptedExtensions indicates nothing except either support for the optional feature or an intent to use the optional feature. Examples:

- o An extension that allows the server to tell the client that cross-SNI resumption is allowed: [I-D.sy-tls-resumption-group].
- o An extension that is used to negotiate support for authentication using both certificates and external PSKs: [I-D.ietf-tls-tls13-cert-with-extern-psk].

This document proposes a single extension called `tls_flags` that can enumerate such flag extensions and allowing both client and server to indicate support for optional features in a concise way.

1.1. Requirements and Other Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD 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 term "flag extension" is used to denote an extension where the `extension_data` field is always zero-length in a particular context,

and the presence of the extension denotes either support for some feature or the intent to use that feature.

The term "flag-type feature" denotes an options TLS 1.3 feature the support for which is negotiated using a flag extension, whether that flag extension is its own extension or a value in the extension defined in this document.

2. The `tls_flags` Extension

This document defines the following extension code point:

```
enum {  
    ...  
    tls_flags(TBD),  
    (65535)  
} ExtensionType;
```

This document also defines the data for this extension as a variable-length bit string, allowing for the encoding of an unbounded number of features.

```
struct {  
    uint8 flags<0..31>;  
} FlagExtensions;
```

The `FlagExtensions` field 8 flags with each octet, and its length is the minimal length that allows it to encode all of the present flags. For example, if we want to encode only flag number zero, the `FlagExtension` field will be 1 octet long, that is encoded as follows:

```
10000000
```

If we want to encode flags 1 and 5, the field will still be 1 octet long:

```
01000100
```

If we want to encode flags 3, 5, and 23, the field will have to be 3 octets long:

```
00010100 00000000 00000001
```

Note that this document does not define any particular bits for this string. That is left to the protocol documents such as the ones in the examples from the previous section. Such documents will have to define which bit to set to show support, and the order of the bits within the bit string shall be enumerated in network order: bit zero

is the high-order bit of the first octet as the flags field is transmitted.

A client that supports this extension SHALL send this extension with the flags field having bits set only for those extensions that it intends to set. If it does not wish to set any such flags in the ClientHello message, then this extension MUST NOT be sent.

A server that supports this extension and also supports at least one of the flag-type features that use this extension and that were declared by the ClientHello extension SHALL send this extension with the intersection of the flags it supports with the flags declared by the client. The intersection operation MAY be implemented as a bitwise AND. The server may need to send two `tls_flags` extensions, one in the ServerHello and the other in the EncryptedExtensions message. It is up to the document for the specific feature to determine whether support should be acknowledged in the ServerHello or the EncryptedExtensions message.

3. IANA Considerations

IANA is requested to assign a new value from the TLS ExtensionType Values registry:

- o The Extension Name should be `tls_flags`
- o The TLS 1.3 value should be CH,SH,EE
- o The Recommended value should be Y
- o The Reference should be this document

IANA is also requested to create a new registry under the TLS namespace with name "TLS Flags" and the following fields:

- o Value, which is a number between 0 and 63. All potential values are available for assignment.
- o Flag Name, which is a string
- o Message, which like the "TLS 1.3" field in the ExtensionType registry contains the abbreviations of the messages that may contain the flag: CH, SH, EE, etc.
- o Recommended, which is a Y/N value determined in the document defining the optional feature.
- o Reference, which is a link to the document defining this flag.

The policy for this shall be "Specification Required" as described in [RFC8126].

4. Security Considerations

The extension described in this document provides a more concise way to express data that could otherwise be expressed in individual extensions. It does not send in the clear any information that would otherwise be sent encrypted, nor vice versa. For this reason this extension is neutral as far as security is concerned.

5. Acknowledgements

The idea for writing this was expressed at the mic during the TLS session at IETF 104 by Eric Rescorla.

The current bitwise formatting was suggested on the mailing list by Nikos Mavrogiannopoulos.

6. References

6.1. Normative References

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

6.2. Informative References

- [I-D.ietf-tls-tls13-cert-with-extern-psk]
Housley, R., "TLS 1.3 Extension for Certificate-based Authentication with an External Pre-Shared Key", draft-ietf-tls-tls13-cert-with-extern-psk-02 (work in progress), May 2019.
- [I-D.sy-tls-resumption-group]
Sy, E., "TLS Resumption across Server Name Indications for TLS 1.3", draft-sy-tls-resumption-group-00 (work in progress), March 2019.

[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/info/rfc8126>>.

Appendix A. Change Log

Version -02 replaced the fixed 64-bit string with an unlimited bitstring, where only the necessary octets are encoded.

Version -01 replaced the enumeration of 8-bit values with a 64-bit bitstring.

Version -00 was a quickly-thrown-together draft with the list of supported features encoded as an array of 8-bit values.

Author's Address

Yoav Nir
DellEMC
9 Andrei Sakharov St
Haifa 3190500
Israel

Email: ynir.ietf@gmail.com

HTTP Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 9, 2020

B. Schwartz
Google
M. Bishop
E. Nygren
Akamai Technologies
July 8, 2019

HTTPSSVC service location and parameter specification via the DNS (DNS
HTTPSSVC)
draft-nygren-httpbis-httpssvc-03

Abstract

This document specifies an "HTTPSSVC" DNS resource record type to facilitate the lookup of information needed to make connections for HTTPS URIs. The HTTPSSVC DNS RR mechanism allows an HTTPS origin hostname to be served from multiple network services, each with associated parameters (such as transport protocol and keying material for encrypting TLS SNI). It also provides a solution for the inability of the DNS to allow a CNAME to be placed at the apex of a domain name. Finally, it provides a way to indicate that the origin supports HTTPS without having to resort to redirects, allowing clients to remove HTTP from the bootstrapping process.

By allowing this information to be bootstrapped in the DNS, it allows for clients to learn of alternative services before their first contact with the origin. This arrangement offers potential benefits to both performance and privacy.

TO BE REMOVED: This proposal is inspired by and based on recent DNS usage proposals such as ALTSVC, ANAME, and ESNIKEYS (as well as long standing desires to have SRV or a functional equivalent implemented for HTTP). These proposals each provide an important function but are potentially incompatible with each other, such as when an origin is load-balanced across multiple hosting providers (multi-CDN). Furthermore, these each add potential cases for adding additional record lookups in-addition to AAAA/A lookups. This design attempts to provide a unified framework that encompasses the key functionality of these proposals, as well as providing some extensibility for addressing similar future challenges.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 9, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
1.1. Introductory Example	4
1.2. Goals of the HTTPSSVC RR	4
1.3. Overview of the HTTPSSVC RR	5
1.4. Additional Alt-Svc parameters	6
1.5. Terminology	6
2. The HTTPSSVC record type	7
2.1. HTTPSSVC RDATA Wire Format	7
2.2. RRNames	8
2.3. SvcRecordType	8
2.4. HTTPSSVC records: alias form	9
2.5. HTTPSSVC records: alternative service form	9
3. Differences from Alt-Svc as transmitted over HTTP	10
3.1. Omitting Max Age and Persist	10
3.2. Multiple records and preference ordering	11
3.3. Constructing Alt-Svc equivalent headers	11
3.4. Granularity and lifetime control	12
4. Client behaviors	12
4.1. Client resolution	12

4.2. HTTP Strict Transport Security	13
4.3. Cache interaction	14
5. DNS Server Behaviors	14
6. Performance optimizations	14
7. Extensions to enhance privacy	15
7.1. Alt-Svc parameter for ESNI keys	15
7.2. Interaction with other standards	15
8. Security Considerations	16
9. IANA Considerations	16
10. Acknowledgements and Related Proposals	17
11. References	17
11.1. Normative References	17
11.2. Informative References	19
Appendix A. Additional examples	20
A.1. Equivalence to Alt-Svc records	20
Appendix B. Comparison with alternatives	20
B.1. Differences from the SRV RRTYPE	20
B.2. Differences from the proposed HTTP record	21
B.3. Differences from the proposed ANAME record	21
B.4. Differences from the proposed ESNI record	21
B.5. SNI Alt-Svc parameter	22
Appendix C. Design Considerations and Open Issues	22
C.1. Record Name	22
C.2. Applicability to other schemes	22
C.3. Wire Format	22
C.4. Extensibility of SvcRecordType	22
C.5. Where to include Priority	22
C.6. Whether to include Weight	23
Appendix D. Change history	23
Authors' Addresses	23

1. Introduction

The HTTPSSVC RR is intended to address a number of challenges facing HTTPS clients and services, while also providing an extensible model to handle similar use-cases without forcing clients to perform additional DNS lookups and without forcing them to first make connections to a default service for the origin.

When clients need to make a connection to fetch resources associated with an HTTPS URI, they must first resolve A and/or AAAA address resource records for the origin hostname. This is adequate when clients default to TCP port 443, do not support Encrypted SNI [ESNI], and where the origin service operator does not have a desire to put an CNAME at a zone apex (such as for "https://example.com"). Handling situations beyond this within the DNS requires learning additional information, and it is highly desirable to minimize the

number of round-trip and lookups required to learn this additional information.

1.1. Introductory Example

As an introductory example, a set of example HTTPSSVC and associated A+AAAA records might be:

```
www.example.com.  2H  IN CNAME    svc.example.net.
example.com.      2H  IN HTTPSSVC 0 0 svc.example.net.
svc.example.net.  2H  IN HTTPSSVC 1 2 svc3.example.net. "h3=\":8003\"; \
                  esnikeys=\"...\""
svc.example.net.  2H  IN HTTPSSVC 1 3 svc2.example.net. "h2=\":8002\"; \
                  esnikeys=\"...\""
svc2.example.net. 300 IN A        192.0.2.2
svc2.example.net. 300 IN AAAA    2001:db8::2
svc3.example.net. 300 IN A        192.0.2.3
svc3.example.net. 300 IN AAAA    2001:db8::3
```

In the preceding example, both of the "example.com" and "www.example.com" origin names are aliased to use service endpoints offered as "svc.example.net" (with "www.example.com" continuing to use a CNAME alias). HTTP/2 is available on a cluster of machines located at svc2.example.net with TCP port 8002 and HTTP/3 is available on a cluster of machines located at svc3.example.net with UDP port 8003. An ESNI key is specified which allows the SNI values of "example.com" and "www.example.com" to be encrypted in the handshake with these service endpoints. When connecting, clients will continue to treat the authoritative origins as "https://example.com" and "https://www.example.com", respectively.

1.2. Goals of the HTTPSSVC RR

The goal of the HTTPSSVC RR is to allow clients to resolve a single additional DNS RR in a way that:

- o Provides service endpoints authoritative for an origin, along with parameters associated with each of these endpoints. In particular:
 - * to support connecting directly to [HTTP3] (QUIC transport) service endpoints
 - * to obtain the [ESNI] keys associated with a service endpoint
- o Does not assume that all service endpoints have the same parameters (such as ESNI keys) or capabilities (such as [HTTP3]) or are even operated by the same entity. This is important as DNS

does not provide any way to tie together multiple RRs for the same name. For example, if `www.example.com` is a CNAME alias that switches between one of three CDNs or hosting environments, records (such as A and AAAA) for that name may have been sourced from different environments.

- o Enables the functional equivalent of a CNAME at a zone apex (such as `"example.com"`) for HTTPS traffic, and generally enables delegation of operational authority for an HTTPS origin within the DNS to an alternate name. This addresses a set of long-standing issues due to HTTP(S) clients not implementing support for SRV records, as well as due to a limitation that a DNS name can not have both a CNAME record as well as NS RRs (as is the case for zone apex names)

1.3. Overview of the HTTPSSVC RR

This subsection briefly describes the HTTPSSVC RR in a non-normative manner.

The HTTPSSVC RR has four primary fields:

1. `SvcRecordType`: A numeric flag indicating how to interpret the subsequent fields. When `"0"`, it indicates that the RR contains an alias. When `"1"`, it indicates that the RR contains an alternative service definition.
2. `SvcFieldPriority`: The priority of this record (relative to others, with lower values preferred). Applicable when `SvcRecordType` is `"1"`, and otherwise has value `"0"`. (Described in Section 3.2.)
3. `SvcDomainName`: The domain name of either the alias target (when `SvcRecordType` is `"0"`) or the uri-host domain name of the alternative service endpoint (when `SvcRecordType` is `"1"`).
4. `SvcFieldValue`: An Alternative Service field value describing the alternative service endpoint for the domain name specified in `SvcDomainName` (only when `SvcRecordType` is `"1"` and otherwise empty).

Cooperating DNS recursive resolvers will perform subsequent record resolution (for HTTPSSVC, A, and AAAA records) and return them in the Additional Section of the response. Clients must either use responses included in the additional section returned by the recursive resolver or perform necessary HTTPSSVC, A, and AAAA record resolutions. DNS authoritative servers may attach in-bailiwick

HTTPSSVC, A, AAAA, and CNAME records in the Additional Section to responses for an HTTPSSVC query.

When SvcRecordType is "1", the HTTPSSVC RR extends the concept introduced in the HTTP Alternative Services proposed standard [AltSvc]. Alt-Svc defines:

- o an extensible data model for describing alternative network endpoints that are authoritative for an origin
- o the "Alt-Svc Field Value", a text format for representing this information
- o standards for sending information in this format from a server to a client over HTTP/1.1 and HTTP/2.

Together, these components provide a toolkit that has proven useful and effective for informing a client of alternative services for an origin. However, making use of an alternative service requires contacting the origin server first. This creates an obvious performance cost: users wait for a full HTTP connection initiation (multiple roundtrips) before learning of an alternative service that is preferred by the origin. The first connection also publicly reveals the user's intended destination to all entities along the network path.

The SvcFieldValue includes the Alt-Svc Field Value through the DNS. This is in its standard text format, with the uri-host portion of the alt-authority component moved into the SvcDomainName field of the HTTPSSVC RR. A client receiving this information during DNS resolution can skip the initial connection and proceed directly to an alternative service.

1.4. Additional Alt-Svc parameters

This document also defines one additional Alt-Svc parameter that can be used within SvcFieldValue:

- o esnikeys (Section 7.1): The ESNIKeys structure from Section 4.1 of [ESNI] for use in encrypting the actual origin hostname in the TLS handshake.

1.5. Terminology

For consistency with [AltSvc], we adopt the following definitions:

- o An "origin" is an information source as in [RFC6454].

- o The "origin server" is the server that the client would reach when accessing the origin in the absence of Alt-Svc.
- o An "alternative service" is a different server that can serve the origin.

Abstractly, the origin consists of a scheme (typically "https"), a host name, and a port (typically "443").

Additional DNS terminology intends to be consistent with [DNSTerm].

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.

2. The HTTPSSVC record type

The HTTPSSVC DNS resource record (RR) type (RRTYPE ???) is used to locate endpoints that can service an "https" origin. The presentation format of the record is:

```
RRName TTL Class HTTPSSVC SvcRecordType SvcFieldPriority \
SvcDomainName SvcFieldValue
```

where SvcRecordType is a numeric value of either "0" or "1", SvcFieldPriority is a number in the range 0-65535, SvcDomainName is a domain name, and SvcFieldValue is a string present when SvcRecordType is "1".

The algorithm for resolving HTTPSSVC records and associated address records is specified in Section 4.1.

2.1. HTTPSSVC RDATA Wire Format

The RDATA for the HTTPSSVC RR consists of:

- o a 1 octet flag field for SvcRecordType, interpreted as an unsigned numeric value (0 to 255, with only values "0" and "1" defined here)
- o a 2 octet field for SvcFieldPriority as an integer in network byte order. If SvcRecordType is "0", SvcFieldPriority MUST be 0.
- o the uncompressed SvcDomainName, represented as a sequence of length-prefixed labels as in Section 3.1 of [RFC1035].

- o the SvcFieldValue byte string, consuming the remainder of the record (so smaller than 65535 octets and constrained by the RRDATA and DNS message sizes).

When SvcRecordType is "0", the SvcFieldValue SHOULD be empty ("") and clients MUST ignore the contents of non-empty SvcFieldValue fields.

2.2. RRNames

In the case of the HTTPSSVC RR, an origin is translated into the RRName in the following manner:

1. If the scheme is "https" and the port is 443, then the RRName is equal to the origin host name. Otherwise the RRName is represented by prefixing the port and scheme with "_", then concatenating them with the host name, resulting in a domain name like "_8443._https.www.example.com."
2. When a prior CNAME or HTTPSSVC record has aliased to an HTTPSSVC record, RRName shall be the name of the alias target.

Note that none of these forms alter the HTTPS origin or authority. For example, clients MUST continue to validate TLS certificate hostnames based on the origin host.

As an example for schemes and ports other than "https" and port 443:

```
_8443._wss.api.example.com. 2H IN HTTPSSVC 0 0 svc4.example.net.  
svc4.example.net. 2H IN HTTPSSVC 1 3 svc4.example.net. "h2=\":8004\"; \\  
esnikeys=\\\"...\\\""
```

would indicate that "wss://api.example.com:8443" is aliased to use HTTP/2 service endpoints offered as "svc4.example.net" on port 8004.

2.3. SvcRecordType

The SvcRecordType field is a numeric value defined to be either "0" or "1". Within an HTTPSSVC RRSet, all RRs must have the same value for SvcRecordType. Clients and recursive servers MUST ignore HTTPSSVC resource records with other SvcRecordType values. If an RRSet contains a record with type "0", the client MUST ignore any records in the set with type "1".

When SvcRecordType is "0", the HTTPSSVC is defined to be in "alias form".

When SvcRecordType is "1", the HTTPSSVC is defined to be in "alternative service form".

2.4. HTTPSSVC records: alias form

When `SvcRecordType` is "0", the HTTPSSVC record is to be treated similar to a CNAME alias pointing to the domain name specified in `SvcDomainName`. HTTPSSVC RRsets MUST only have a single resource record in this form. If multiple are present, clients or recursive resolvers SHOULD pick one non-deterministically.

The common use-case for this form of the HTTPSSVC record is as an alternative to CNAMEs at the zone apex where they are not allowed. For example, if an operator of `https://example.com` wanted to point HTTPS requests to a service operating at `svc.example.net`, they would publish a record such as:

```
example.com. 3600 IN HTTPSSVC 0 0 svc.example.net.
```

The `SvcDomainName` MUST point to a domain name that contains another HTTPSSVC record and/or address (AAAA and/or A) records.

Note that the `RRName` and the `SvcDomainName` MAY themselves be CNAMEs. Clients and recursive resolvers MUST follow CNAMEs as normal.

Due to the risk of loops, clients and recursive resolvers MUST implement loop detection. Chains of consecutive HTTPSSVC and CNAME records SHOULD be limited to (8?) prior to reaching terminal address records.

The `SvcFieldValue` in this form SHOULD be an empty string and clients MUST ignore its contents.

As legacy clients will not know to use this record, service operators will likely need to retain fallback AAAA and A records alongside this HTTPSSVC record, although in a common case the target of the HTTPSSVC record might have better performance, and therefore would be preferable for clients implementing this specification to use.

2.5. HTTPSSVC records: alternative service form

When `SvcRecordType` is "1", the combination of `SvcDomainName` and `SvcFieldValue` within each resource record associates an Alternative Service Field Value with an origin.

The `SvcFieldValue` of the HTTPSSVC resource record contains an Alt-Svc Field Value, exactly as defined in Section 4 of [AltSvc], but with the `uri-host` moved to the `SvcDomainName` field.

For example, if the operator of `https://www.example.com` intends to include an HTTP response header like

```
Alt-Svc: h3="svc.example.net:8003"; ma=3600, \
        h2="svc.example.net:8002"; ma=3600
```

they could also publish an HTTPSSVC DNS RRSet like

```
www.example.com. 3600 IN HTTPSSVC 1 2 svc.example.net. "h3=\":8003\""  
                    HTTPSSVC 1 3 svc.example.net. "h2=\":8002\""
```

This data type can be represented as an Unknown RR as described in [RFC3597]:

```
www.example.com. 3600 IN TYPE??? \\\# TBD:WRITE ME
```

This construction is intended to be extensible in two ways. First, any extensions that are made to the Alt-Svc format for transmission over HTTPS are also applicable here, unless expressly mentioned otherwise.

Second, by defining a way to map non-HTTPS schemes and non-default ports (Section 2.2), we provide a way for the HTTPSSVC to be used for them as needed. However, by using the origin name for the RRName for scheme https and port 443 we allow HTTPSSVC records to be included at the end of CNAME chains for existing site implementations without requiring changes in the zone containing the origin.

3. Differences from Alt-Svc as transmitted over HTTP

Publishing an alternative services form HTTPSSVC record in DNS is intended to be equivalent to transmitting this field value over HTTPS, and receiving an HTTPSSVC record is intended to be equivalent to receiving this field value over HTTPS. However, there are some small differences in the intended client and server behavior.

3.1. Omitting Max Age and Persist

When publishing an HTTPSSVC record in DNS, server operators **MUST** omit the "ma" parameter, which encodes the "max age" (i.e. expiration time) of an Alt-Svc Field Value. Instead, server operators **SHOULD** encode the expiration time in the DNS TTL, and **MUST NOT** set a TTL longer than the intended "max age".

When receiving an HTTPSSVC record, clients **SHOULD** synthesize a new "ma" parameter from the DNS TTL if the resulting alt-value is being passed to a subsystem that might employ caching.

When publishing an HTTPSSVC record, server operators **MUST** omit the "persist" parameter, which indicates whether the client should use this record on other network paths. When receiving an HTTPSSVC

record, clients MUST discard any records that contain a "persist" flag. Disabling persistence is important to prevent a local adversary in one network from implanting a forged DNS record that allows them to track users or hinder their connections after they leave that network.

3.2. Multiple records and preference ordering

Server operators MAY publish multiple SvcRecordType "1" HTTPSSVC records as an RRSET. When converting a collection of alt-values into an HTTPSSVC RRSET, the server operator MUST set the overall TTL to a value no larger than the minimum of the "max age" values (following Section 5.2 of [RFC2181]).

Each RR MUST contain exactly one alt-value, as described in Section 3 of [AltSvc].

As RRs within an RRSET are explicitly unordered collections, the SvcFieldPriority value is introduced to indicate priority. HTTPSSVC RRs with a smaller SvcFieldPriority value SHOULD be given preference over RRs with a larger SvcFieldPriority value.

Alt-values received via HTTPS are preferred over any Alt-value received via DNS.

When receiving an RRSET containing multiple HTTPSSVC records with the same SvcFieldPriority value, clients SHOULD apply a random shuffle within a priority level to the records before using them, to ensure randomized load-balancing.

3.3. Constructing Alt-Svc equivalent headers

For a client to construct the equivalent of an Alt-Svc HTTP response header:

1. For each RR, the SvcDomainName MUST be inserted as the uri-host. If SvcDomainName is has the value "." then the RRNAME for the final HTTPSSVC record MUST be inserted as the uri-host. (In the case of a CNAME or a HTTPSSVC SvcRecordType "0" record pointing to an HTTPSSVC record with SvcRecordType "1" and SvcDomainName "." then it is the RRNAME for the terminal HTTPSSVC record that must be inserted as the uri-host.)
2. The RRs SHOULD be ordered by increasing SvcFieldPriority, with shuffling for equal SvcFieldPriority values. Clients MAY choose to further prioritize alt-values where address records are immediately available for the alt-value's SvcDomainName.

3. The client SHOULD concatenate the thus-transformed-and-ordered SvcFieldValues in the RRSET, separated by commas. (This is semantically equivalent to receiving multiple Alt-Svc HTTP response headers, according to Section 3.2.2 of [HTTP]).

3.4. Granularity and lifetime control

Sending Alt-Svc over HTTP allows the server to tailor the Alt-Svc Field Value specifically to the client. When using an HTTPSSVC DNS record, groups of clients will necessarily receive the same Alt-Svc Field Value. Therefore, this standard is not suitable for uses that require single-client granularity in Alt-Svc.

Some DNS caching systems incorrectly extend the lifetime of DNS records beyond the stated TTL. Server operators MUST NOT rely on HTTPSSVC records expiring on time, and MAY shorten the TTL to compensate.

4. Client behaviors

4.1. Client resolution

When attempting to resolve a name HOST, clients should follow in-order:

1. Issue parallel AAAA/A and HTTPSSVC queries for the name HOST. The answers for these may or may not include CNAME pointers before reaching one or more of these records.
2. If an HTTPSSVC record of SvcRecordType "0" is returned for HOST, clients should loop back to step 1 replacing HOST with SvcDomainName, subject to loop detection heuristics.
3. If one or more HTTPSSVC record of SvcRecordType "1" is returned for HOST, clients should synthesize equivalent Alt-Svc Field Values based on the SvcDomainName and SvcFieldValue. If one of these alt-values is selected to be used in a connection, the client will need to resolve AAAA and/or A records for SvcDomainName.
4. If only AAAA and/or A records are present for HOST (and no HTTPSSVC), clients should make a connection to one of the IP addresses contained in these records and proceed normally.

When selecting between AAAA and A records to use, clients may use an approach such as [HappyEyeballsV2]

Some possible optimizations are discussed in Section 6 to reduce latency impact in comparison to ordinary AAAA/A lookups.

4.2. HTTP Strict Transport Security

By publishing an HTTPSSVC record, the server operator indicates that all useful HTTP resources on that origin are reachable over HTTPS, similar to HTTP Strict Transport Security [HSTS]. When an HTTPSSVC record is present for an origin, all "http" scheme requests for that origin SHOULD logically be redirected to "https".

Prior to making an "http" scheme request, the client SHOULD perform a lookup to determine if an HTTPSSVC record is available for that origin. To do so, the client SHOULD construct a corresponding "https" URL as follows:

1. Replace the "http" scheme with "https".
2. If the "http" URL explicitly specifies port 80, specify port 443.
3. Do not alter any other aspect of the URL.

This construction is equivalent to Section 8.3 of [HSTS] , point 5.

If an HTTPSSVC record is present for this "https" URL, the client should treat this as the equivalent of receiving an HTTP "307 Temporary Redirect" redirect to the "https" URL. Because HTTPSSVC is received over an often insecure channel (DNS), clients MUST NOT place any more trust in this signal than if they had received a 307 redirect over cleartext HTTP.

If the HTTPSSVC query results in a SERVFAIL error, and the connection between the client and the recursive resolver is cryptographically protected (e.g. using TLS [RFC7858] or HTTPS [RFC8484]), the client SHOULD abandon the connection attempt and display an error message. A SERVFAIL error can occur if the domain is DNSSEC-signed, the recursive resolver is DNSSEC-validating, and an active attacker between the recursive resolver and the authoritative DNS server is attempting to prevent the upgrade to HTTPS.

Similarly, if the client enforces DNSSEC validation on A/AAAA RRs, it SHOULD abandon the connection attempt if the HTTPSSVC RR fails to validate.

4.3. Cache interaction

If the client has an Alt-Svc cache, and a usable Alt-Svc value is present in that cache, then the client SHOULD NOT issue an HTTPSSVC DNS query. Instead, the client SHOULD proceed with alternative service connection as usual.

If the client has a cached Alt-Svc entry that is expiring, the client MAY perform an HTTPSSVC query to refresh the entry.

5. DNS Server Behaviors

Recursive DNS servers SHOULD resolve SvcDomainName records and include them in the Additional Section (along with any relevant CNAME records). For SvcRecordType=0, recursive DNS servers SHOULD attempt to resolve and include A, AAAA, and HTTPSSVC records. For SvcRecordType=1, recursive DNS servers SHOULD attempt to resolve and include A and AAAA records.

Authoritative DNS servers SHOULD return A, AAAA, and HTTPSSVC records (as well as any relevant CNAME records) in the Additional Section for any in-bailiwick SvcDomainNames.

6. Performance optimizations

For optimal performance (i.e. minimum connection setup time), clients SHOULD issue address (AAAA and/or A) and HTTPSSVC queries simultaneously, and SHOULD implement a client-side DNS cache. With these optimizations in place, and conforming DNS servers, using HTTPSSVC does not add network latency to connection setup.

A nonconforming recursive resolver might return an HTTPSSVC response with a nonempty SvcDomainName, without the corresponding address records. If all the HTTPSSVC RRs in the response have nonempty SvcDomainName values, and the client does not have address records for any of these values in its DNS cache, the client SHOULD perform an additional address query for the selected SvcDomainName.

The additional DNS query in this case introduces a delay. To avoid causing a delay for clients using a nonconforming recursive resolver, domain owners SHOULD choose the SvcDomainName to be a name in the origin hostname's CNAME chain if possible. This will ensure that the required address records are already present in the client's DNS cache as part of the responses to the address queries that were issued in parallel.

Highly performance-sensitive clients MAY implement the following special- case shortcut to avoid increased connection time: if (1) one

of the HTTPSSVC records returned has `SvcRecordType=0`, (2) its `SvcDomainName` is not in the DNS cache, and (3) the address queries for the origin domain return usable IP addresses, then the client MAY ignore the HTTPSSVC records and connect directly to the origin domain. When the `SvcDomainNames` and any needed HTTPSSVC records are available, the client SHOULD make subsequent requests over connections specified by the HTTPSSVC records.

Server operators can therefore expect that publishing HTTPSSVC records with `SvcRecordType=0` should not cause an additional DNS query for performance-sensitive clients. Server operators who wish to prevent this optimization should use `SvcRecordType=1`.

7. Extensions to enhance privacy

7.1. Alt-Svc parameter for ESNI keys

An Alt-Svc "esnikeys" parameter is defined for specifying ESNI keys corresponding to an alternative service. The value of the parameter is an `ESNIKeys` structure [ESNI] encoded in [base64], or the empty string. ESNI-aware clients SHOULD prefer alt-values with nonempty esnikeys.

This parameter MAY also be sent in Alt-Svc HTTP response headers and HTTP/2 ALTSVC frames.

The Alt-Svc specification states that "the client MAY fall back to using the origin" in case of connection failure [AltSvc]. This behavior is not suitable for ESNI, because fallback would negate the privacy benefits of ESNI.

Accordingly, any connection attempt that uses ESNI MUST fall back only to another alt-value that also has the esnikeys parameter. If the parameter's value is the empty string, the client SHOULD connect as it would in the absence of any `ESNIKeys` information.

For example, suppose a server operator has two alternatives. Alternative A is reliably accessible but does not support ESNI. Alternative B supports ESNI but is not reliably accessible. The server operator could include a full esnikeys value in Alternative B, and mark Alternative A with `esnikeys=""` to indicate that fallback from B to A is allowed.

7.2. Interaction with other standards

The purpose of this standard is to reduce connection latency and improve user privacy. Server operators implementing this standard SHOULD also implement TLS 1.3 [RFC8446] and OCSP Stapling [RFC6066],

both of which confer substantial performance and privacy benefits when used in combination with HTTPSSVC records.

To realize the greatest privacy benefits, this proposal is intended for use with a privacy-preserving DNS transport (like DNS over TLS [RFC7858] or DNS over HTTPS [RFC8484]). However, performance improvements, and some modest privacy improvements, are possible without the use of those standards.

This RRTYPE could be extended to support schemes other than "https". Any such scheme MUST have an entry under the HTTPSSVC RRTYPE in the IANA DNS Underscore Global Scoped Entry Registry [Attrleaf] The scheme SHOULD have an entry in the IANA URI Schemes Registry [RFC7595]. The scheme SHOULD be one for which Alt-Svc is defined.

8. Security Considerations

Alt-Svc Field Values are intended for distribution over untrusted channels, and clients are REQUIRED to verify that the alternative service is authoritative for the origin (Section 2.1 of [AltSvc]). Therefore, DNSSEC signing and validation are OPTIONAL for publishing and using HTTPSSVC records.

TBD: expand this section in more detail. In particular: * Just as with [AltSvc], clients must validate the TLS server certificate against hostname associated with the origin. Clients MUST NOT use the SvcDomainName as any part of the server TLS certificate validation. * ...

9. IANA Considerations

Per [RFC6895], please add the following entry to the data type range of the Resource Record (RR) TYPEs registry:

TYPE	Meaning	Reference
HTTPSSVC	HTTPS Service Location	(This document)

Per [Attrleaf], please add the following entries to the DNS Underscore Global Scoped Entry Registry:

RR TYPE	_NODE NAME	Meaning	Reference
HTTPSSVC	_https	Alt-Svc for HTTPS	(This document)

Per [AltSvc], please add the following entries to the HTTP Alt-Svc Parameter Registry:

Alt-Svc Parameter	Meaning	Reference
esnikeys	Encrypted SNI keys	(This document)

10. Acknowledgements and Related Proposals

There have been a wide range of proposed solutions over the years to the "CNAME at the Zone Apex" challenge proposed. These include [I-D.draft-bellis-dnsop-http-record-00], [I-D.draft-ietf-dnsop-aname-03], and others.

Thank you to Ian Swett, Ralf Weber, Jon Reed, Martin Thompson, Lucas Pardue, Ilari Liusvaara, and others for their feedback and suggestions on this draft.

11. References

11.1. Normative References

- [AltSvc] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <<https://www.rfc-editor.org/info/rfc7838>>.
- [AltSvcSNI] Bishop, M., "The "SNI" Alt-Svc Parameter", draft-bishop-httpbis-sni-altsvc-02 (work in progress), May 2018.
- [Attrleaf] Crocker, D., "DNS Scoped Data Through "Underscore" Naming of Attribute Leaves", draft-ietf-dnsop-attrleaf-16 (work in progress), November 2018.
- [base64] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [ESNI] Rescorla, E., Oku, K., Sullivan, N., and C. Wood, "Encrypted Server Name Indication for TLS 1.3", draft-ietf-tls-esni-03 (work in progress), March 2019.

- [HappyEyeballsV2] Schinazi, D. and T. Pauly, "Happy Eyeballs Version 2: Better Connectivity Using Concurrency", RFC 8305, DOI 10.17487/RFC8305, December 2017, <<https://www.rfc-editor.org/info/rfc8305>>.
- [HSTS] Hodges, J., Jackson, C., and A. Barth, "HTTP Strict Transport Security (HSTS)", RFC 6797, DOI 10.17487/RFC6797, November 2012, <<https://www.rfc-editor.org/info/rfc6797>>.
- [HTTP3] Bishop, M., "Hypertext Transfer Protocol Version 3 (HTTP/3)", draft-ietf-quic-http-20 (work in progress), April 2019.
- [RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <<https://www.rfc-editor.org/info/rfc1035>>.
- [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>>.
- [RFC2181] Elz, R. and R. Bush, "Clarifications to the DNS Specification", RFC 2181, DOI 10.17487/RFC2181, July 1997, <<https://www.rfc-editor.org/info/rfc2181>>.
- [RFC3597] Gustafsson, A., "Handling of Unknown DNS Resource Record (RR) Types", RFC 3597, DOI 10.17487/RFC3597, September 2003, <<https://www.rfc-editor.org/info/rfc3597>>.
- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <<https://www.rfc-editor.org/info/rfc6066>>.
- [RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <<https://www.rfc-editor.org/info/rfc6454>>.
- [RFC7595] Thaler, D., Ed., Hansen, T., and T. Hardie, "Guidelines and Registration Procedures for URI Schemes", BCP 35, RFC 7595, DOI 10.17487/RFC7595, June 2015, <<https://www.rfc-editor.org/info/rfc7595>>.

- [RFC7858] Hu, Z., Zhu, L., Heidemann, J., Mankin, A., Wessels, D., and P. Hoffman, "Specification for DNS over Transport Layer Security (TLS)", RFC 7858, DOI 10.17487/RFC7858, May 2016, <<https://www.rfc-editor.org/info/rfc7858>>.
- [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>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.
- [RFC8484] Hoffman, P. and P. McManus, "DNS Queries over HTTPS (DoH)", RFC 8484, DOI 10.17487/RFC8484, October 2018, <<https://www.rfc-editor.org/info/rfc8484>>.

11.2. Informative References

- [DNSTerm] Hoffman, P., Sullivan, A., and K. Fujiwara, "DNS Terminology", BCP 219, RFC 8499, DOI 10.17487/RFC8499, January 2019, <<https://www.rfc-editor.org/info/rfc8499>>.
- [HTTP] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.
- [I-D.draft-bellis-dnsop-http-record-00] Bellis, R., "A DNS Resource Record for HTTP", draft-bellis-dnsop-http-record-00 (work in progress), November 2018.
- [I-D.draft-ietf-dnsop-aname-03] Finch, T., Hunt, E., Dijk, P., Eden, A., and W. Mekking, "Address-specific DNS aliases (ANAME)", draft-ietf-dnsop-aname-03 (work in progress), April 2019.
- [RFC2782] Gulbrandsen, A., Vixie, P., and L. Esibov, "A DNS RR for specifying the location of services (DNS SRV)", RFC 2782, DOI 10.17487/RFC2782, February 2000, <<https://www.rfc-editor.org/info/rfc2782>>.
- [RFC6895] Eastlake 3rd, D., "Domain Name System (DNS) IANA Considerations", BCP 42, RFC 6895, DOI 10.17487/RFC6895, April 2013, <<https://www.rfc-editor.org/info/rfc6895>>.

Appendix A. Additional examples

A.1. Equivalence to Alt-Svc records

The following:

```
www.example.com. 2H IN CNAME    svc.example.net.
example.com.     2H IN HTTPSSVC 0 0 svc.example.net.
svc.example.net. 2H IN HTTPSSVC 1 2 svc3.example.net. "h3=\":8003\"; \
                  esnikeys=\"ABC...\""
svc.example.net. 2H IN HTTPSSVC 1 3 . "h2=\":8002\"; \
                  esnikeys=\"123...\""
```

is equivalent to the Alt-Svc record:

```
Alt-Svc: h3="svc3.example.net:8003"; esnikeys="ABC..."; ma=7200, \
         h2="svc.example.net:8002"; esnikeys="123..."; ma=7200
```

for the origins of both "https://www.example.com" and
"https://example.com".

Appendix B. Comparison with alternatives

The HTTPSSVC record type closely resembles some existing record types and proposals. A complaint with all of the alternatives is that web clients have seemed unenthusiastic about implementing them. The hope here is that by providing an extensible solution that solves multiple problems we will overcome the inertia and have a path to achieve client implementation.

B.1. Differences from the SRV RRTYPE

An SRV record [RFC2782] can perform a similar function to the HTTPSSVC record, informing a client to look in a different location for a service. However, there are several differences:

- o SRV records are typically mandatory, whereas clients will always continue to function correctly without making use of Alt-Svc or HTTPSSVC.
- o SRV records cannot instruct the client to switch or upgrade protocols, whereas Alt-Svc can signal such an upgrade (e.g. to HTTP/2).
- o SRV records are not extensible, whereas Alt-Svc and thus HTTPSSVC can be extended with new parameters. For example, this is what allows the incorporation of ESNI keys in HTTPSSVC.

- o Using SRV records would not allow a client to skip processing of the Alt-Svc information in a subsequent connection, so it does not confer a performance advantage.

B.2. Differences from the proposed HTTP record

Unlike [I-D.draft-bellis-dnsop-http-record-00], this approach is extensible to cover Alt-Svc and ESNIKeys use-cases. Like that proposal, this addresses the zone apex CNAME challenge.

Like that proposal it remains necessary to continue to include address records at the zone apex for legacy clients.

B.3. Differences from the proposed ANAME record

Unlike [I-D.draft-ietf-dnsop-aname-03], this approach is extensible to cover Alt-Svc and ESNIKeys use-cases. This approach also does not require any changes or special handling on either authoritative or master servers, beyond optionally returning in-bailiwick additional records.

Like that proposal, this addresses the zone apex CNAME challenge for clients that implement this.

However with this HTTPSSVC proposal it remains necessary to continue to include address records at the zone apex for legacy clients. If deployment of this standard is successful, the number of legacy clients will fall over time. As the number of legacy clients declines, the operational effort required to serve these users without the benefit of HTTPSSVC indirection should fall. Server operators can easily observe how much traffic reaches this legacy endpoint, and may remove the apex's address records if the observed legacy traffic has fallen to negligible levels.

B.4. Differences from the proposed ESNI record

Unlike [ESNI], this approach is extensible and covers the Alt-Svc case as well as addresses the zone apex CNAME challenge.

By using the Alt-Svc model we also provide a way to solve the ESNI multi-CDN challenges in a general case.

Unlike ESNI, this is focused on the specific case of HTTPS, although this approach could be extended for other protocols. It also allows specifying ESNI keys for a specific port, not an entire host.

B.5. SNI Alt-Svc parameter

Defining an Alt-Svc sni= parameter (such as from [AltSvcSNI]) would have provided some benefits to clients and servers not implementing ESNI, such as for specifying that "_wildcard.example.com" could be sent as an SNI value rather than the full name. There is nothing precluding HTTPSSVC from being used with an sni= parameter if one were to be defined, but it is not included here to reduce scope, complexity, and additional potential security and tracking risks.

Appendix C. Design Considerations and Open Issues

This draft is intended to be a work-in-progress for discussion. Many details are expected to change with subsequent refinement. Some known issues or topics for discussion are listed below.

C.1. Record Name

Naming is hard. The "HTTPSSVC" is proposed as a placeholder. Other names for this record might include ALTSVC, HTTPS, HTTPSSRV, B, or something else.

C.2. Applicability to other schemes

The focus of this record is on optimizing the common case of the "https" scheme. It is worth discussing whether this is a valid assumption or if a more general solution is applicable. Past efforts to over-generalize have not met with broad success.

C.3. Wire Format

Advice from experts in DNS wire format best practices would be greatly appreciated to refine the proposed details, overall.

C.4. Extensibility of SvcRecordType

Only values of "0" and "1" are allowed for SvcRecordType. Should we give more thought to potential future values? The current version tries to leave this open by indicating that resource records with unknown SvcRecordType values should be ignored (and perhaps should be switched to MUST be ignored)?

C.5. Where to include Priority

The SvcFieldPriority could alternately be included as a pri= Alt-Svc attribute. It wouldn't be applicable for Alt-Svc returned via HTTP, but it is also not necessarily needed by DNS servers. It is also not used when SvcRecordType=0. A related question is whether to omit it

from the textual representation when SvcRecordType=0. Regardless, having a series of sequential numeric values in the textual representation has risk of user error, especially as MX, SRV, and others all have their own variations here.

C.6. Whether to include Weight

Some other similar mechanisms such as SRV have a weight in-addition to priority. That is excluded here for simplicity. It could always be added as an optional Alt-Svc attribute.

Appendix D. Change history

- o draft-nygren-httpbis-httpssvc-03
 - * Change redirect type for HSTS-style behavior from 302 to 307 to reduce ambiguities.
- o draft-nygren-httpbis-httpssvc-02
 - * Remove the redundant length fields from the wire format.
 - * Define a SvcDomainName of "." for SvcRecordType=1 as being the HTTPSSVC RRNAME.
 - * Replace "hq" with "h3".
- o draft-nygren-httpbis-httpssvc-01
 - * Fixes of record name. Replace references to "HTTPSVC" with "HTTPSSVC".
- o draft-nygren-httpbis-httpssvc-00
 - * Initial version

Authors' Addresses

Ben Schwartz
Google

Email: bemasc@google.com

Mike Bishop
Akamai Technologies

Email: mbishop@evequefou.be

Erik Nygren
Akamai Technologies

Email: erik+iETF@nygren.org

tls
Internet-Draft
Intended status: Standards Track
Expires: May 3, 2020

B. Schwartz
Google LLC
October 31, 2019

TLS Metadata for Load Balancers
draft-schwartz-tls-lb-02

Abstract

A load balancer that does not terminate TLS may wish to provide some information to the backend server, in addition to forwarding TLS data. This draft proposes a protocol between load balancers and backends that enables secure, efficient delivery of TLS with additional information. The need for such a protocol has recently become apparent in the context of split mode ESNI.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 3, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Conventions and Definitions	2
2. Background	2
3. Goals	3
4. Overview	4
5. Encoding	4
6. Defined ProxyExtensions	6
6.1. padding	6
6.2. client_address	6
6.3. destination_address	6
6.4. esni_inner	6
6.5. certificate_padding	7
6.6. overload	7
6.7. ratchet	8
7. Protocol wire format	9
8. Security considerations	10
8.1. Integrity	10
8.2. Confidentiality	10
8.3. Fingerprinting	11
9. IANA Considerations	11
10. References	11
10.1. Normative References	11
10.2. Informative References	12
Appendix A. Acknowledgements	12
Author's Address	12

1. Conventions and Definitions

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.

Data encodings are expressed in the TLS 1.3 presentation language, as defined in Section 3 of [TLS13].

2. Background

A load balancer is a server or bank of servers that acts as an intermediary between the client and a range of backend servers. As the name suggests, a load balancer's primary function is to ensure that client traffic is spread evenly across the available backend servers. However load balancers also serve many other functions,

such as identifying connections intended for different backends and forwarding them appropriately, or dropping connections that are deemed malicious.

A load balancer operates at a specific point in the protocol stack, forwarding e.g. IP packets, TCP streams, TLS contents, HTTP requests, etc. Most relevant to this proposal are TCP and TLS load balancers. TCP load balancers terminate the TCP connection with the client and establish a new TCP connection to the selected backend, bidirectionally copying the TCP contents between these two connections. TLS load balancers additionally terminate the TLS connection, forwarding the plaintext to the backend server (typically inside a new TLS connection). TLS load balancers must therefore hold the private keys for the domains they serve.

When a TCP load balancer forwards a TLS stream, the load balancer has no way to incorporate additional information into the stream. Insertion of any additional data would cause the connection to fail. However, the load-balancer and backend can share additional information if they agree to speak a new protocol. The most popular protocol used for this purpose is currently the PROXY protocol [PROXY], developed by HAProxy. This protocol prepends a plaintext collection of metadata (e.g. client IP address) onto the TCP socket. The backend can parse this metadata, then pass the remainder of the stream to its TLS library.

The PROXY protocol is effective and widely used, but it offers no confidentiality or integrity protection, and therefore might not be suitable when the load balancer and backend communicate over the public internet. It also does not offer a way for the backend to reply.

3. Goals

- o Enable TCP load balancers to forward metadata to the backend.
- o Enable backends to reply.
- o Reduce the need for TLS-terminating load balancers.
- o Ensure confidentiality and integrity for all forwarded metadata.
- o Enable split ESNI architectures.
- o Prove to the backend that the load balancer intended to associate this metadata with this connection.
- o Achieve good CPU and memory efficiency.

- o Don't impose additional latency.
- o Support backends that receive a mixture of direct and load-balanced TLS.
- o Enable simple and safe implementation.

4. Overview

The proposed protocol supports a two-way exchange between a load balancer and a backend server. It works by prepending information to the TLS handshake:

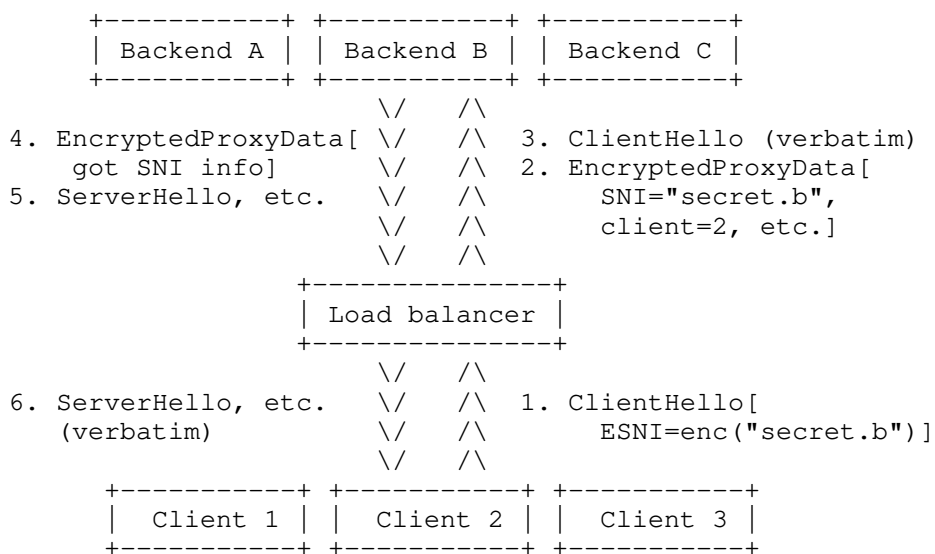


Figure 1: Data flow diagram

5. Encoding

A ProxyExtension is identical in form to a standard TLS Extension (Section 4.2 of [TLS13]), with a new identifier space for the extension types.

```

struct {
    ProxyExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} ProxyExtension;

```

ProxyExtensions can be sent in an upstream (to the backend) or downstream (to the load balancer) direction

```
enum {  
    upstream(0),  
    downstream(1),  
    (255)  
} ProxyDataDirection;
```

The ProxyData contains a set of ProxyExtensions.

```
struct {  
    ProxyDataDirection direction;  
    ProxyExtension proxy_data<0..2^16-1>;  
} ProxyData;
```

The EncryptedProxyData structure contains metadata associated with the original ClientHello (Section 4.1.2 of [TLS13]), encrypted with a pre-shared key that is configured out of band.

```
struct {  
    opaque psk_identity<1..2^16-1>;  
    opaque nonce<8..2^16-1>  
    opaque encrypted_proxy_data<1..2^16-1>;  
} EncryptedProxyData;
```

- o "psk_identity": The identity of a PSK previously agreed upon by the load balancer and the backend. Including the PSK identity allows for updating the PSK without disruption.
- o "nonce": Non-repeating initializer for the AEAD. This prevents an attacker from observing whether the same ClientHello is marked with different metadata over time.
- o "encrypted_proxy_data": "AEAD-Encrypt(key, nonce, additional_data, plaintext=ProxyData)". The key and AEAD function are agreed out of band and associated with "psk_identity". The "additional_data" is context-dependent.

When the load balancer receives a ClientHello, it serializes any relevant metadata into an upstream ProxyData, then encrypts it with the ClientHello as "additional_data" to produce the EncryptedProxyData. The backend's reply is a downstream ProxyData struct, also transmitted as an EncryptedProxyData, using the upstream EncryptedProxyData as "additional_data". Recipients in each case MUST verify that "ProxyData.direction" has the expected value, and discard the connection if it does not.

The downstream ProxyData SHOULD NOT contain any ProxyExtensionType values that were not present in the upstream ProxyData.

6. Defined ProxyExtensions

Like a standard TLS Extension, a ProxyExtension is identified by a uint16 type number. Load balancers MUST only include extensions that are registered for use in ProxyData. Backends MUST ignore any extensions that they do not recognize.

There are initially seven type numbers allocated:

```
enum {  
    padding(0),  
    client_address(1),  
    destination_address(2),  
    esni_inner(3),  
    certificate_padding(4),  
    overload(5),  
    ratchet(6),  
    (65535)  
} ProxyExtensionType;
```

6.1. padding

The "padding" extension functions as described in [RFC7685]. It is used here to avoid leaking information about the other extensions. It can be used in upstream and downstream ProxyData.

6.2. client_address

The "client_address" extension functions as described in [I-D.kinnear-tls-client-net-address]. It conveys the client IP address observed by the load balancer. Backends that make use of this extension SHOULD include an empty "client_address" extension in the downstream ProxyData.

6.3. destination_address

The "destination_address" extension is identical to the "client_address" extension, except that it contains the load balancer's server IP address that received this connection.

6.4. esni_inner

The "esni_inner" extension is only sent upstream, and can only be used if the ClientHello contains the encrypted_server_name extension [ESNI]. The "extension_data" is the ClientESNIInner (Section 5.1.1 of [ESNI]), which contains the true SNI and nonce. This is useful when the load balancer knows the ESNI private key and the backend does not, i.e. split mode ESNI.

6.5. certificate_padding

The "certificate_padding" extension always contains a single uint32 value. The upstream value conveys the padding granularity "G", and the downstream value indicates the unpadded size of the Certificate struct (Section 4.4.2 of [TLS13]).

To pad the Handshake message (Section 4 of [TLS13]) containing the Certificate struct, the backend SHOULD select the smallest "length_of_padding" (Section 5.2 of [TLS13]) such that "Handshake.length + length_of_padding" is a multiple of "G".

The load balancer SHOULD include this extension whenever it sends the "esni_inner" extension.

Padding certificates from many backends to the same length is important to avoid revealing which backend is responding to a ClientHello. Load balancer operators SHOULD ensure that no backend has a unique certificate size after padding, and MAY set "G" large enough to make all responses have equal size.

6.6. overload

In the upstream ProxyData, the "overload" extension contains a single uint16 indicating the approximate proportion of connections that are being routed to this server as a fraction of 65535. If there is only one server, load balancers SHOULD set the value to 65535.

In the downstream ProxyData, the value is an OverloadValue:

```
enum {
    accepted(0),
    overloaded(1),
    rejected(2),
    (255)
} OverloadState;
struct {
    OverloadState state;
    uint16 load;
    uint32 ttl;
} OverloadValue;
```

When "OverloadValue.state" is "accepted", the backend is accepting connections normally. The "overloaded" state indicates that the backend is accepting this connection, but would prefer not to receive additional connections. A value of "rejected" indicates that the backend did not accept this connection. When sending a "rejected"

response, the backend SHOULD close the connection without sending a ServerHello.

"OverloadValue.load" indicates the load fraction of the responding backend server, with 65535 indicating maximum load.

The load balancer SHOULD treat this information as valid for "OverloadValue.ttl" seconds, or until it receives another OverloadValue from that server.

Load balancers that have multiple available backends for an origin SHOULD avoid connecting to servers that are in the "overloaded" or "rejected" state. When a connection is rejected, the load balancer MAY retry that connection by sending the ClientHello to a different backend server. When multiple servers are in the "accepted" state, the load balancer MAY use "OverloadValue.load" to choose among them.

When there is a server in an unknown state (i.e. a new server or one whose last TTL has expired), the load balancer SHOULD direct at least one connection to it, in order to refresh its OverloadState.

If all servers are in the "overloaded" or "rejected" state, the load balancer SHOULD drop the connection.

6.7. ratchet

If the backend server is reachable without traversing the load balancer, and an adversary can observe packets on the link between the load balancer and the backend, then that adversary can execute a replay flooding attack, sending the backend server duplicate copies of observed EncryptedProxyData and ClientHello. This attack can waste server resources on the Diffie-Hellman operations required to process the ClientHello, resulting in denial of service.

The "ratchet" extension reduces the impact of such an attack on the backend server by allowing the backend to reject these duplicates after decrypting the ProxyData. (This decryption uses only a symmetric cipher, so it is expected to be much faster than typical Diffie-Hellman operations.) Its upstream payload consists of a RatchetValue:

```
struct {  
    uint64 index;  
    uint64 floor;  
} RatchetValue;
```

A RatchetValue is scoped to a single backend server and "psk_identity". Within that scope, the load balancer initializes "index" to a random value, and executes the following procedure:

1. For each new forwarded connection (to the same server under the same "psk_identity"), increment "index".
2. Set "floor" to the "index" of the earliest connection that has not yet been connected or closed.

The backend server initializes "floor" to the first "RatchetValue.floor" it receives (under a "psk_identity"), and then executes the following procedure for each incoming connection:

1. Define "a >= b" if the most significant bit of "a - b" is 0.
2. Let "newValue" be the RatchetValue in the ProxyData.
3. If "newValue.index < floor", ignore the connection.
4. If "newValue.floor >= floor", set "floor" to "newValue.floor".
5. OPTIONALLY, ignore the connection if "newValue.index" has been seen recently. This can be implemented efficiently by keeping track of any "index" values greater than "floor" that appear to have been skipped.

With these measures in place, replays can be rejected without processing the ClientHello.

In principle, this replay protection fails after 2^{64} connections when the "floor" value wraps. On a backend server that averages 10^9 new connections per second, this would occur after 584 years. To avoid this replay attack, load balancers and backends SHOULD establish a new PSK at least this often.

Backends that are making use of the "ratchet" extension SHOULD include an empty "ratchet" extension in their downstream ProxyData.

7. Protocol wire format

When forwarding a TLS stream over TCP, the load balancer SHOULD prepend a TLSPlaintext whose "content_type" is XX (proxy_header) and whose "fragment" is the EncryptedProxyData.

Following this proxy header, the load balancer MUST send the full contents of the TCP stream, exactly as received from the client. The backend will observe the proxy header, immediately followed by a

TLSPlainText containing the ClientHello. The backend will decrypt the EncryptedProxyData using the ClientHello as associated data, and process the ClientHello and the remainder of the stream as standard TLS.

Similarly, the backend SHOULD reply with the downstream EncryptedProxyData in a proxy header, followed by the normal TLS stream, beginning with a TLSPlainText frame containing the ServerHello. If the downstream ProxyHeader is not present, has an unrecognized version number, or produces an error, the load balancer SHOULD proxy the rest of the stream regardless.

8. Security considerations

8.1. Integrity

This protocol is intended to provide both parties with a strong guarantee of integrity for the metadata they receive. For example, an active attacker cannot take metadata intended for one stream and attach it to another, because each stream will have a unique ClientHello, and the metadata is bound to the ClientHello by AEAD.

One exception to this protection is in the case of an attacker who deliberately reissues identical ClientHello messages. An attacker who reuses a ClientHello can also reuse the metadata associated with it, if they can first observe the EncryptedProxyData transferred between the load balancer and the backend. This could be used by an attacker to reissue data originally generated by a true client (e.g. as part of a 0-RTT replay attack), or it could be used by a group of adversaries who are willing to share a single set of client secrets while initiating different sessions, in order to reuse metadata that they find helpful.

Backends that are sensitive to this attack SHOULD implement the "ratchet" mechanism in Section 6.7, including the optional defenses.

8.2. Confidentiality

This protocol is intended to maintain confidentiality of the metadata transferred between the load balancer and backend, especially the ESNI plaintext and the client IP address. An observer between the client and the load balancer does not observe this protocol at all, and an observer between the load balancer and backend observes only ciphertext.

However, an adversary who can monitor both of these links can easily observe that a connection from the client to the load balancer is shortly followed by a connection from the load balancer to a backend,

with the same ClientHello. This reveals which backend server the client intended to visit. In many cases, the choice of backend server could be the sensitive information that ESNI is intended to protect.

8.3. Fingerprinting

Connections to different domains might be distinguishable by the cleartext contents of the ServerHello, such as "cipher_suite" and "server_share.group". Load balancer operators with ESNI support SHOULD provide backend operators with a list of cipher suites and groups to support, and a preference order, to avoid different backends having distinctive behaviors.

9. IANA Considerations

IANA will be directed to add the following allocation to the TLS ContentType registry:

Value	Description	DTLS-OK	Reference
XX	proxy_header	N	This document

IANA will be directed to create a new "TLS ProxyExtensionType Values" registry on the TLS Extensions page. Values less than 0x8000 will be subject to the "RFC Required" registration procedure, and the rest will be "First Come First Served". To avoid codepoint exhaustion, proxy developers SHOULD pack all their nonstandard information into a single ProxyExtension.

10. References

10.1. Normative References

- [ESNI] Rescorla, E., Oku, K., Sullivan, N., and C. Wood, "Encrypted Server Name Indication for TLS 1.3", draft-ietf-tls-esni-04 (work in progress), July 2019.
- [I-D.kinnear-tls-client-net-address] Kinnear, E., Pauly, T., and C. Wood, "TLS Client Network Address Extension", draft-kinnear-tls-client-net-address-00 (work in progress), March 2019.

- [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>>.
- [RFC7685] Langley, A., "A Transport Layer Security (TLS) ClientHello Padding Extension", RFC 7685, DOI 10.17487/RFC7685, October 2015, <<https://www.rfc-editor.org/info/rfc7685>>.
- [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>>.

10.2. Informative References

- [PROXY] Tarreau, W., "The PROXY protocol", March 2017, <<https://www.haproxy.org/download/1.8/doc/proxy-protocol.txt>>.

Appendix A. Acknowledgements

This is an elaboration of an idea proposed by Eric Rescorla during the development of ESNI. Thanks to David Schinazi, David Benjamin, and Piotr Sikora for suggesting important improvements.

Author's Address

Benjamin M. Schwartz
Google LLC

Email: bemasc@google.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 15, 2020

D. Stebila
University of Waterloo
S. Fluhrer
Cisco Systems
S. Gueron
U. Haifa, Amazon Web Services
February 12, 2020

Hybrid key exchange in TLS 1.3
draft-stebila-tls-hybrid-design-03

Abstract

Hybrid key exchange refers to using multiple key exchange algorithms simultaneously and combining the result with the goal of providing security even if all but one of the component algorithms is broken. It is motivated by transition to post-quantum cryptography. This document provides a construction for hybrid key exchange in the Transport Layer Security (TLS) protocol version 1.3.

Discussion of this work is encouraged to happen on the TLS IETF mailing list tls@ietf.org or on the GitHub repository which contains the draft: <https://github.com/dstebila/draft-stebila-tls-hybrid-design>.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 15, 2020.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
1.1. Revision history	3
1.2. Terminology	4
1.3. Motivation for use of hybrid key exchange	5
1.4. Scope	5
1.5. Goals	6
2. Key encapsulation mechanisms	7
3. Construction for hybrid key exchange	8
3.1. Negotiation	8
3.2. Transmitting public keys and ciphertexts	9
3.3. Shared secret calculation	10
4. Open questions	12
5. IANA Considerations	13
6. Security Considerations	13
7. Acknowledgements	14
8. References	14
8.1. Normative References	14
8.2. Informative References	14
Appendix A. Related work	18
Appendix B. Design Considerations	19
B.1. (Neg) How to negotiate hybridization and component algorithms?	21
B.1.1. Key exchange negotiation in TLS 1.3	21
B.1.2. (Neg-Ind) Negotiating component algorithms individually	21
B.1.3. (Neg-Comb) Negotiating component algorithms as a combination	22
B.1.4. Benefits and drawbacks	23
B.2. (Num) How many component algorithms to combine?	24
B.2.1. (Num-2) Two	24
B.2.2. (Num-2+) Two or more	24
B.2.3. Benefits and Drawbacks	24
B.3. (Shares) How to convey key shares?	24
B.3.1. (Shares-Concat) Concatenate key shares	25
B.3.2. (Shares-Multiple) Send multiple key shares	25
B.3.3. (Shares-Ext-Additional) Extension carrying additional	

key shares	25
B.3.4. Benefits and Drawbacks	25
B.4. (Comb) How to use keys?	26
B.4.1. (Comb-Concat) Concatenate keys	26
B.4.2. (Comb-KDF-1) KDF keys	27
B.4.3. (Comb-KDF-2) KDF keys	28
B.4.4. (Comb-XOR) XOR keys	29
B.4.5. (Comb-Chain) Chain of KDF applications for each key .	30
B.4.6. (Comb-AltInput) Second shared secret in an alternate KDF input	31
B.4.7. Benefits and Drawbacks	31
Authors' Addresses	32

1. Introduction

This document gives a construction for hybrid key exchange in TLS 1.3. The overall design approach is a simple, "concatenation"-based approach: each hybrid key exchange combination should be viewed as a single new key exchange method, negotiated and transmitted using the existing TLS 1.3 mechanisms.

This document does not propose specific post-quantum mechanisms; see Section 1.4 for more on the scope of this document.

1.1. Revision history

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

Earlier versions of this document categorized various design decisions one could make when implementing hybrid key exchange in TLS 1.3. These have been moved to the appendix of the current draft, and will be eventually be removed.

o draft-03:

- * Add requirement for KEMs to provide protection against key reuse.
- * Clarify FIPS-compliance of shared secret concatenation method.

o draft-02:

- * Design considerations from draft-00 and draft-01 are moved to the appendix.
- * A single construction is given in the main body.

- o draft-01:
 - * Add (Comb-KDF-1) (Appendix B.4.2) and (Comb-KDF-2) (Appendix B.4.3) options.
 - * Add two candidate instantiations.
- o draft-00: Initial version.

1.2. Terminology

For the purposes of this document, it is helpful to be able to divide cryptographic algorithms into two classes:

- o "Traditional" algorithms: Algorithms which are widely deployed today, but which may be deprecated in the future. In the context of TLS 1.3 in 2019, examples of traditional key exchange algorithms include elliptic curve Diffie-Hellman using secp256r1 or x25519, or finite-field Diffie-Hellman.
- o "Next-generation" (or "next-gen") algorithms: Algorithms which are not yet widely deployed, but which may eventually be widely deployed. An additional facet of these algorithms may be that we have less confidence in their security due to them being relatively new or less studied. This includes "post-quantum" algorithms.

"Hybrid" key exchange, in this context, means the use of two (or more) key exchange algorithms based on different cryptographic assumptions, e.g., one traditional algorithm and one next-gen algorithm, with the purpose of the final session key being secure as long as at least one of the component key exchange algorithms remains unbroken. We use the term "component" algorithms to refer to the algorithms combined in a hybrid key exchange.

The primary motivation of this document is preparing for post-quantum algorithms. However, it is possible that public key cryptography based on alternative mathematical constructions will be required independent of the advent of a quantum computer, for example because of a cryptanalytic breakthrough. As such we opt for the more generic term "next-generation" algorithms rather than exclusively "post-quantum" algorithms.

Note that TLS 1.3 uses the phrase "groups" to refer to key exchange algorithms - for example, the "supported_groups" extension - since all key exchange algorithms in TLS 1.3 are Diffie-Hellman-based. As a result, some parts of this document will refer to data structures

or messages with the term "group" in them despite using a key exchange algorithm that is not Diffie-Hellman-based nor a group.

1.3. Motivation for use of hybrid key exchange

A hybrid key exchange algorithm allows early adopters eager for post-quantum security to have the potential of post-quantum security (possibly from a less-well-studied algorithm) while still retaining at least the security currently offered by traditional algorithms. They may even need to retain traditional algorithms due to regulatory constraints, for example FIPS compliance.

Ideally, one would not use hybrid key exchange: one would have confidence in a single algorithm and parameterization that will stand the test of time. However, this may not be the case in the face of quantum computers and cryptanalytic advances more generally.

Many (though not all) post-quantum algorithms currently under consideration are relatively new; they have not been subject to the same depth of study as RSA and finite-field or elliptic curve Diffie-Hellman, and thus the security community does not necessarily have as much confidence in their fundamental security, or the concrete security level of specific parameterizations.

Moreover, it is possible that even by the end of the NIST Post-Quantum Cryptography Standardization Project, and for a period of time thereafter, conservative users may not have full confidence in some algorithms.

As such, there may be users for whom hybrid key exchange is an appropriate step prior to an eventual transition to next-generation algorithms.

1.4. Scope

This document focuses on hybrid ephemeral key exchange in TLS 1.3 [TLS13]. It intentionally does not address:

- o Selecting which next-generation algorithms to use in TLS 1.3, nor algorithm identifiers nor encoding mechanisms for next-generation algorithms. This selection will be based on the recommendations by the Crypto Forum Research Group (CFRG), which is currently waiting for the results of the NIST Post-Quantum Cryptography Standardization Project [NIST].
- o Authentication using next-generation algorithms. If a cryptographic assumption is broken due to the advent of a quantum computer or some other cryptanalytic breakthrough, confidentiality

of information can be broken retroactively by any adversary who has passively recorded handshakes and encrypted communications. In contrast, session authentication cannot be retroactively broken.

1.5. Goals

The primary goal of a hybrid key exchange mechanism is to facilitate the establishment of a shared secret which remains secure as long as as one of the component key exchange mechanisms remains unbroken.

In addition to the primary cryptographic goal, there may be several additional goals in the context of TLS 1.3:

- o ***Backwards compatibility:** Clients and servers who are "hybrid-aware", i.e., compliant with whatever hybrid key exchange standard is developed for TLS, should remain compatible with endpoints and middle-boxes that are not hybrid-aware. The three scenarios to consider are:
 1. Hybrid-aware client, hybrid-aware server: These parties should establish a hybrid shared secret.
 2. Hybrid-aware client, non-hybrid-aware server: These parties should establish a traditional shared secret (assuming the hybrid-aware client is willing to downgrade to traditional-only).
 3. Non-hybrid-aware client, hybrid-aware server: These parties should establish a traditional shared secret (assuming the hybrid-aware server is willing to downgrade to traditional-only).

Ideally backwards compatibility should be achieved without extra round trips and without sending duplicate information; see below.

- o ***High performance:** Use of hybrid key exchange should not be prohibitively expensive in terms of computational performance. In general this will depend on the performance characteristics of the specific cryptographic algorithms used, and as such is outside the scope of this document. See [BCNS15], [CECPQ1], [FRODO] for preliminary results about performance characteristics.
- o ***Low latency:** Use of hybrid key exchange should not substantially increase the latency experienced to establish a connection. Factors affecting this may include the following.

- * The computational performance characteristics of the specific algorithms used. See above.
- * The size of messages to be transmitted. Public key and ciphertext sizes for post-quantum algorithms range from hundreds of bytes to over one hundred kilobytes, so this impact can be substantial. See [BCNS15], [FRODO] for preliminary results in a laboratory setting, and [LANGLEY] for preliminary results on more realistic networks.
- * Additional round trips added to the protocol. See below.
- o *No extra round trips:* Attempting to negotiate hybrid key exchange should not lead to extra round trips in any of the three hybrid-aware/non-hybrid-aware scenarios listed above.
- o *Minimal duplicate information:* Attempting to negotiate hybrid key exchange should not mean having to send multiple public keys of the same type.

2. Key encapsulation mechanisms

In the context of the NIST Post-Quantum Cryptography Standardization Project, key exchange algorithms are formulated as key encapsulation mechanisms (KEMs), which consist of three algorithms:

- o "KeyGen() -> (pk, sk)": A probabilistic key generation algorithm, which generates a public key "pk" and a secret key "sk".
- o "Encaps(pk) -> (ct, ss)": A probabilistic encapsulation algorithm, which takes as input a public key "pk" and outputs a ciphertext "ct" and shared secret "ss".
- o "Decaps(sk, ct) -> ss": A decapsulation algorithm, which takes as input a secret key "sk" and ciphertext "ct" and outputs a shared secret "ss", or in some cases a distinguished error value.

The main security property for KEMs is indistinguishability under adaptive chosen ciphertext attack (IND-CCA2), which means that shared secret values should be indistinguishable from random strings even given the ability to have arbitrary ciphertexts decapsulated. IND-CCA2 corresponds to security against an active attacker, and the public key / secret key pair can be treated as a long-term key or reused. A common design pattern for obtaining security under key reuse is to apply the Fujisaki-Okamoto (FO) transform [FO] or a variant thereof [HHK].

A weaker security notion is indistinguishability under chosen plaintext attack (IND-CPA), which means that the shared secret values should be indistinguishable from random strings given a copy of the public key. IND-CPA roughly corresponds to security against a passive attacker, and sometimes corresponds to one-time key exchange.

Key exchange in TLS 1.3 is phrased in terms of Diffie-Hellman key exchange in a group. DH key exchange can be modeled as a KEM, with "KeyGen" corresponding to selecting an exponent " x " as the secret key and computing the public key " g^x "; encapsulation corresponding to selecting an exponent " y ", computing the ciphertext " g^y " and the shared secret " g^{xy} ", and decapsulation as computing the shared secret " g^{xy} ". See [I-D.irtf-cfrg-hpke] for more details of such Diffie-Hellman-based key encapsulation mechanisms.

TLS 1.3 does not require that ephemeral public keys be used only in a single key exchange session; some implementations may reuse them, at the cost of limited forward secrecy. As a result, any KEM used in this document MUST explicitly be designed to be secure in the event that the public key is re-used, such as achieving IND-CCA2 security or having a transform like the Fujisaki-Okamoto transform [FO] [HHK] applied. While it is recommended that implementations avoid reuse of KEM public keys, implementations that do reuse KEM public keys MUST ensure that the number of reuses of a KEM public key abides by any bounds in the specification of the KEM or subsequent security analyses. Implementations MUST NOT reuse randomness in the generation of KEM ciphertexts.

3. Construction for hybrid key exchange

3.1. Negotiation

Each particular combination of algorithms in a hybrid key exchange will be represented as a "NamedGroup" and sent in the "supported_groups" extension. No internal structure or grammar is implied or required in the value of the identifier; they are simply opaque identifiers.

Each value representing a hybrid key exchange will correspond to an ordered pair of two algorithms. For example, a future document could specify that hybrid value 0x2000 corresponds to secp256r1+ntruhrss701, and 0x2001 corresponds to x25519+ntruhrss701. (We note that this is independent from future documents standardizing solely post-quantum key exchange methods, which would have to be assigned their own identifier.)

Specific values shall be standardized by IANA in the TLS Supported Groups registry. We suggest that values 0x2000 through 0x2EFF are

suitable for hybrid key exchange methods (the leading "2" suggesting that there are 2 algorithms), noting that 0x2A2A is reserved as a GREASE value [GREASE]. This document requests that values 0x2F00 through 0x2FFF be reserved for Private Use for hybrid key exchange.

```
enum {  
  
    /* Elliptic Curve Groups (ECDHE) */  
    secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019),  
    x25519(0x001D), x448(0x001E),  
  
    /* Finite Field Groups (DHE) */  
    ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102),  
    ffdhe6144(0x0103), ffdhe8192(0x0104),  
  
    /* Hybrid Key Exchange Methods */  
    TBD(0xTBD), ...,  
  
    /* Reserved Code Points */  
    ffdhe_private_use(0x01FC..0x01FF),  
    hybrid_private_use(0x2F00..0x2FFF),  
    ecdhe_private_use(0xFE00..0xFEFF),  
    (0xFFFF)  
} NamedGroup;
```

3.2. Transmitting public keys and ciphertexts

We take the relatively simple "concatenation approach": the messages from the two algorithms being hybridized will be concatenated together and transmitted as a single value, to avoid having to change existing data structures. However we do add structure in the concatenation procedure, specifically including length fields, so that the concatenation operation is unambiguous. Note that among the Round 2 candidates in the NIST Post-Quantum Cryptography Standardization Project, not all algorithms have fixed public key sizes; for example, the SIKE key encapsulation mechanism permits compressed or uncompressed public keys at each security level, and the compressed and uncompressed formats are interoperable.

Recall that in TLS 1.3 a KEM public key or KEM ciphertext is represented as a "KeyShareEntry":

```
struct {  
    NamedGroup group;  
    opaque key_exchange<1..2^16-1>;  
} KeyShareEntry;
```

These are transmitted in the "extension_data" fields of "KeyShareClientHello" and "KeyShareServerHello" extensions:

```
struct {
    KeyShareEntry client_shares<0..2^16-1>;
} KeyShareClientHello;

struct {
    KeyShareEntry server_share;
} KeyShareServerHello;
```

The client's shares are listed in descending order of client preference; the server selects one algorithm and sends its corresponding share.

For a hybrid key exchange, the "key_exchange" field of a "KeyShareEntry" is the following data structure:

```
struct {
    opaque key_exchange_1<1..2^16-1>;
    opaque key_exchange_2<1..2^16-1>;
} HybridKeyExchange
```

The order of shares in the "HybridKeyExchange" struct is the same as the order of algorithms indicated in the definition of the "NamedGroup".

For the client's share, the "key_exchange_1" and "key_exchange_2" values are the "pk" outputs of the corresponding KEMs' "KeyGen" algorithms, if that algorithm corresponds to a KEM; or the (EC)DH ephemeral key share, if that algorithm corresponds to an (EC)DH group. For the server's share, the "key_exchange_1" and "key_exchange_2" values are the "ct" outputs of the corresponding KEMs' "Encaps" algorithms, if that algorithm corresponds to a KEM; or the (EC)DH ephemeral key share, if that algorithm corresponds to an (EC)DH group.

3.3. Shared secret calculation

Here we also take a simple "concatenation approach": the two shared secrets are concatenated together and used as the shared secret in the existing TLS 1.3 key schedule. In this case, we do not add any additional structure (length fields) in the concatenation procedure: among all Round 2 candidates, once the algorithm and variant are specified, the shared secret output length is fixed.

In other words, the shared secret is calculated as

4. Open questions

Larger public keys and/or ciphertexts. The "HybridKeyExchange" struct in Section 3.2 limits public keys and ciphertexts to $2^{16}-1$ bytes; this is bounded by the same $(2^{16}-1)$ -byte limit on the "key_exchange" field in the "KeyShareEntry" struct. Some post-quantum KEMs have larger public keys and/or ciphertexts; for example, Classic McEliece's smallest parameter set has public key size 261,120 bytes. Hence this draft can not accommodate all current NIST Round 2 candidates.

If it is desired to accommodate algorithms with public keys or ciphertexts larger than $2^{16}-1$ bytes, options include a) revising the TLS 1.3 standard to allow longer "key_exchange" fields; b) creating an alternative extension which is sufficiently large; or c) providing a reference to an external public key, e.g. a URL at which to look up the public key (along with a hash to verify).

Duplication of key shares. Concatenation of public keys in the "HybridKeyExchange" struct as described in Section 3.2 can result in sending duplicate key shares. For example, if a client wanted to offer support for two combinations, say "secp256r1+sikep503" and "x25519+sikep503", it would end up sending two sikep503 public keys, since the "KeyShareEntry" for each combination contains its own copy of a sikep503 key. This duplication may be more problematic for post-quantum algorithms which have larger public keys.

If it is desired to avoid duplication of key shares, options include a) disconnect the use of a combination for the algorithm identifier from the use of concatenation of public keys by introducing new logic and/or data structures (see Appendix B.3.2 or Appendix B.3.3); or b) provide some back reference from a later key share entry to an earlier one.

Variable-length shared secrets. The shared secret calculation in Section 3.3 directly concatenates the shared secret values of each scheme, rather than encoding them with length fields. This implicitly assumes that the length of each shared secret is fixed once the algorithm is fixed. This is the case for all Round 2 candidates.

However, if it is envisioned that this specification be used with algorithms which do not have fixed-length shared secrets (after the variant has been fixed by the algorithm identifier in the "NamedGroup" negotiation in Section 3.1), then Section 3.3 should be revised to use an unambiguous concatenation method such as the following:

```
struct {  
    opaque shared_secret_1<1..2^16-1>;  
    opaque shared_secret_2<1..2^16-1>;  
} HybridSharedSecret
```

Guidance from the working group is particularly requested on this point.

Resumption. TLS 1.3 allows for session resumption via a PSK. When a PSK is used during session establishment, an ephemeral key exchange can also be used to enhance forward secrecy. If the original key exchange was hybrid, should an ephemeral key exchange in a resumption of that original key exchange be required to use the same hybrid algorithms?

Failures. Some post-quantum key exchange algorithms have non-trivial failure rates: two honest parties may fail to agree on the same shared secret with non-negligible probability. Does a non-negligible failure rate affect the security of TLS? How should such a failure be treated operationally? What is an acceptable failure rate?

5. IANA Considerations

Identifiers for specific key exchange algorithm combinations will be defined in later documents. This document requests IANA reserve values 0x2F00..0x2FFF in the TLS Supported Groups registry for private use for hybrid key exchange methods.

6. Security Considerations

The shared secrets computed in the hybrid key exchange should be computed in a way that achieves the "hybrid" property: the resulting secret is secure as long as at least one of the component key exchange algorithms is unbroken. See [GIACON] and [BINDEL] for an investigation of these issues. Under the assumption that shared secrets are fixed length once the combination is fixed, the construction from Section 3.3 corresponds to the dual-PRF combiner of [BINDEL] which is shown to preserve security under the assumption that the hash function is a dual-PRF.

As noted in Section 2, KEMs used in this document MUST explicitly be designed to be secure in the event that the public key is re-used, such as achieving IND-CCA2 security or having a transform like the Fujisaki-Okamoto transform applied. Some IND-CPA-secure post-quantum KEMs (i.e., without countermeasures such as the FO transform) are completely insecure under public key reuse; for example, some

lattice-based IND-CPA-secure KEMs are vulnerable to attacks that recover the private key after just a few thousand samples [FLUHRER].

7. Acknowledgements

These ideas have grown from discussions with many colleagues, including Christopher Wood, Matt Campagna, Eric Crockett, authors of the various hybrid Internet-Drafts and implementations cited in this document, and members of the TLS working group. The immediate impetus for this document came from discussions with attendees at the Workshop on Post-Quantum Software in Mountain View, California, in January 2019. Martin Thomson suggested the (Comb-KDF-1) (Appendix B.4.2) approach. Daniel J. Bernstein and Tanja Lange commented on the risks of reuse of ephemeral public keys.

8. References

8.1. Normative References

- [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>>.

8.2. Informative References

- [BCNS15] Bos, J., Costello, C., Naehrig, M., and D. Stebila, "Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem", 2015 IEEE Symposium on Security and Privacy, DOI 10.1109/sp.2015.40, May 2015.
- [BERNSTEIN] "Post-Quantum Cryptography", Springer Berlin Heidelberg book, DOI 10.1007/978-3-540-88702-7, 2009.
- [BINDEL] Bindel, N., Brendel, J., Fischlin, M., Goncalves, B., and D. Stebila, "Hybrid Key Encapsulation Mechanisms and Authenticated Key Exchange", Post-Quantum Cryptography pp. 206-226, DOI 10.1007/978-3-030-25510-7_12, 2019.
- [CECPQ1] Braithwaite, M., "Experimenting with Post-Quantum Cryptography", July 2016, <<https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html>>.
- [CECPQ2] Langley, A., "CECPQ2", December 2018, <<https://www.imperialviolet.org/2018/12/12/cecpq2.html>>.

- [DFGS15] Dowling, B., Fischlin, M., Guenther, F., and D. Stebila, "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol Candidates", Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15, DOI 10.1145/2810103.2813653, 2015.
- [DODIS] Dodis, Y. and J. Katz, "Chosen-Ciphertext Security of Multiple Encryption", Theory of Cryptography pp. 188-209, DOI 10.1007/978-3-540-30576-7_11, 2005.
- [DOWLING] Dowling, B., "Provable Security of Internet Protocols", Queensland University of Technology dissertation, DOI 10.5204/thesis.eprints.108960, n.d..
- [ETSI] Campagna, M., Ed. and . others, "Quantum safe cryptography and security: An introduction, benefits, enablers and challengers", ETSI White Paper No. 8 , June 2015, <<https://www.etsi.org/images/files/ETSIWhitePapers/QuantumSafeWhitepaper.pdf>>.
- [EVEN] Even, S. and O. Goldreich, "On the Power of Cascade Ciphers", Advances in Cryptology pp. 43-50, DOI 10.1007/978-1-4684-4730-9_4, 1984.
- [EXTERN-PSK] Housley, R., "TLS 1.3 Extension for Certificate-based Authentication with an External Pre-Shared Key", draft-ietf-tls-tls13-cert-with-extern-psk-07 (work in progress), December 2019.
- [FLUHRER] Fluhrer, S., "Cryptanalysis of ring-LWE based key exchange with key share reuse", Cryptology ePrint Archive, Report 2016/085 , January 2016, <<https://eprint.iacr.org/2016/085>>.
- [FO] Fujisaki, E. and T. Okamoto, "Secure Integration of Asymmetric and Symmetric Encryption Schemes", Journal of Cryptology Vol. 26, pp. 80-101, DOI 10.1007/s00145-011-9114-1, December 2011.
- [FRODO] Bos, J., Costello, C., Ducas, L., Mironov, I., Naehrig, M., Nikolaenko, V., Raghunathan, A., and D. Stebila, "Frodo", Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16, DOI 10.1145/2976749.2978425, 2016.

- [GIACON] Giacon, F., Heuer, F., and B. Poettering, "KEM Combiners", Public-Key Cryptography – PKC 2018 pp. 190–218, DOI 10.1007/978-3-319-76578-5_7, 2018.
- [GREASE] Benjamin, D., "Applying GREASE to TLS Extensibility", draft-ietf-tls-grease-04 (work in progress), August 2019.
- [HARNIK] Harnik, D., Kilian, J., Naor, M., Reingold, O., and A. Rosen, "On Robust Combiners for Oblivious Transfer and Other Primitives", Lecture Notes in Computer Science pp. 96–113, DOI 10.1007/11426639_6, 2005.
- [HHK] Hofheinz, D., Hoewelmanns, K., and E. Kiltz, "A Modular Analysis of the Fujisaki-Okamoto Transformation", Theory of Cryptography pp. 341–371, DOI 10.1007/978-3-319-70500-2_12, 2017.
- [HOFFMAN] Hoffman, P., "The Transition from Classical to Post-Quantum Cryptography", draft-hoffman-c2pq-06 (work in progress), November 2019.
- [I-D.irtf-cfrg-hpke]
Barnes, R. and K. Bhargavan, "Hybrid Public Key Encryption", draft-irtf-cfrg-hpke-02 (work in progress), November 2019.
- [IKE-HYBRID]
Tjhai, C., Tomlinson, M., grbartle@cisco.com, g., Fluhrer, S., Geest, D., Garcia-Morchon, O., and V. Smyslov, "Framework to Integrate Post-quantum Key Exchanges into Internet Key Exchange Protocol Version 2 (IKEv2)", draft-tjhai-ipsecme-hybrid-qske-ikev2-04 (work in progress), July 2019.
- [IKE-PSK] Fluhrer, S., Kampanakis, P., McGrew, D., and V. Smyslov, "Mixing Preshared Keys in IKEv2 for Post-quantum Security", draft-ietf-ipsecme-qr-ikev2-11 (work in progress), January 2020.
- [KIEFER] Kiefer, F. and K. Kwiatkowski, "Hybrid ECDHE-SIDH Key Exchange for TLS", draft-kiefer-tls-ecdhe-sidh-00 (work in progress), November 2018.
- [KPW13] Krawczyk, H., Paterson, K., and H. Wee, "On the Security of the TLS Protocol: A Systematic Analysis", Advances in Cryptology – CRYPTO 2013 pp. 429–448, DOI 10.1007/978-3-642-40041-4_24, 2013.

- [LANGLEY] Langley, A., "Post-quantum confidentiality for TLS", April 2018, <<https://www.imperialviolet.org/2018/04/11/pqconftls.html>>.
- [NIELSEN] Nielsen, M. and I. Chuang, "Quantum Computation and Quantum Information", Cambridge University Press , 2000.
- [NIST] National Institute of Standards and Technology (NIST), "Post-Quantum Cryptography", n.d., <<https://www.nist.gov/pqcrypto>>.
- [NIST-FAQ] National Institute of Standards and Technology (NIST), "Post-Quantum Cryptography - FAQs", January 2020, <<https://csrc.nist.gov/Projects/post-quantum-cryptography/faqs>>.
- [NIST-SP-800-135] National Institute of Standards and Technology (NIST), "Recommendation for Existing Application-Specific Key Derivation Functions", December 2011, <<https://doi.org/10.6028/NIST.SP.800-135r1>>.
- [NIST-SP-800-56C] National Institute of Standards and Technology (NIST), "Recommendation for Key-Derivation Methods in Key-Establishment Schemes", April 2018, <<https://doi.org/10.6028/NIST.SP.800-56Cr1>>.
- [OQS-102] Open Quantum Safe Project, "OQS-OpenSSL-1-0-2_stable", November 2018, <https://github.com/open-quantum-safe/openssl/tree/OQS-OpenSSL_1_0_2-stable>.
- [OQS-111] Open Quantum Safe Project, "OQS-OpenSSL-1-1-1_stable", November 2018, <https://github.com/open-quantum-safe/openssl/tree/OQS-OpenSSL_1_1_1-stable>.
- [SCHANCK] Schanck, J. and D. Stebila, "A Transport Layer Security (TLS) Extension For Establishing An Additional Shared Secret", draft-schanck-tls-additional-keyshare-00 (work in progress), April 2017.
- [WHYTE12] Schanck, J., Whyte, W., and Z. Zhang, "Quantum-Safe Hybrid (QSH) Ciphersuite for Transport Layer Security (TLS) version 1.2", draft-whyte-qsh-tls12-02 (work in progress), July 2016.

- [WHYTE13] Whyte, W., Zhang, Z., Fluhrer, S., and O. Garcia-Morchon, "Quantum-Safe Hybrid (QSH) Key Exchange for Transport Layer Security (TLS) version 1.3", draft-whyte-qsh-tls13-06 (work in progress), October 2017.
- [XMSS] Huelsing, A., Butin, D., Gazdag, S., Rijneveld, J., and A. Mohaisen, "XMSS: eXtended Merkle Signature Scheme", RFC 8391, DOI 10.17487/RFC8391, May 2018, <<https://www.rfc-editor.org/info/rfc8391>>.
- [ZHANG] Zhang, R., Hanaoka, G., Shikata, J., and H. Imai, "On the Security of Multiple Encryption or CCA-security+CCA-security=CCA-security?", Public Key Cryptography - PKC 2004 pp. 360-374, DOI 10.1007/978-3-540-24632-9_26, 2004.

Appendix A. Related work

Quantum computing and post-quantum cryptography in general are outside the scope of this document. For a general introduction to quantum computing, see a standard textbook such as [NIELSEN]. For an overview of post-quantum cryptography as of 2009, see [BERNSTEIN]. For the current status of the NIST Post-Quantum Cryptography Standardization Project, see [NIST]. For additional perspectives on the general transition from classical to post-quantum cryptography, see for example [ETSI] and [HOFFMAN], among others.

There have been several Internet-Drafts describing mechanisms for embedding post-quantum and/or hybrid key exchange in TLS:

- o Internet-Drafts for TLS 1.2: [WHYTE12]
- o Internet-Drafts for TLS 1.3: [KIEFER], [SCHANCK], [WHYTE13]

There have been several prototype implementations for post-quantum and/or hybrid key exchange in TLS:

- o Experimental implementations in TLS 1.2: [BCNS15], [CECPQ1], [FRODO], [OQS-102]
- o Experimental implementations in TLS 1.3: [CECPQ2], [OQS-111]

These experimental implementations have taken an ad hoc approach and not attempted to implement one of the drafts listed above.

Unrelated to post-quantum but still related to the issue of combining multiple types of keying material in TLS is the use of pre-shared keys, especially the recent TLS working group document on including an external pre-shared key [EXTERN-PSK].

Considering other IETF standards, there is work on post-quantum preshared keys in IKEv2 [IKE-PSK] and a framework for hybrid key exchange in IKEv2 [IKE-HYBRID]. The XMSS hash-based signature scheme has been published as an informational RFC by the IRTF [XMSS].

In the academic literature, [EVEN] initiated the study of combining multiple symmetric encryption schemes; [ZHANG], [DODIS], and [HARNIK] examined combining multiple public key encryption schemes, and [HARNIK] coined the term "robust combiner" to refer to a compiler that constructs a hybrid scheme from individual schemes while preserving security properties. [GIACON] and [BINDEL] examined combining multiple key encapsulation mechanisms.

Appendix B. Design Considerations

This appendix discusses choices one could make along four distinct axes when integrating hybrid key exchange into TLS 1.3:

1. How to negotiate the use of hybridization in general and component algorithms specifically?
2. How many component algorithms can be combined?
3. How should multiple key shares (public keys / ciphertexts) be conveyed?
4. How should multiple shared secrets be combined?

The construction in the main body illustrates one selection along each of these axes. The remainder of this appendix outlines various options we have identified for each of these choices. Immediately below we provide a summary list. Options are labelled with a short code in parentheses to provide easy cross-referencing.

1. (Neg) (Appendix B.1) How to negotiate the use of hybridization in general and component algorithms specifically?
 - * (Neg-Ind) (Appendix B.1.2) Negotiating component algorithms individually
 - + (Neg-Ind-1) (Appendix B.1.2.1) Traditional algorithms in "ClientHello" "supported_groups" extension, next-gen algorithms in another extension
 - + (Neg-Ind-2) (Appendix B.1.2.2) Both types of algorithms in "supported_groups" with external mapping to tradition/next-gen.

- + (Neg-Ind-3) (Appendix B.1.2.3) Both types of algorithms in "supported_groups" separated by a delimiter.
 - * (Neg-Comb) (Appendix B.1.3) Negotiating component algorithms as a combination
 - + (Neg-Comb-1) (Appendix B.1.3.1) Standardize "NamedGroup" identifiers for each desired combination.
 - + (Neg-Comb-2) (Appendix B.1.3.2) Use placeholder identifiers in "supported_groups" with an extension defining the combination corresponding to each placeholder.
 - + (Neg-Comb-3) (Appendix B.1.3.3) List combinations by inserting grouping delimiters into "supported_groups" list.
2. (Num) (Appendix B.2) How many component algorithms can be combined?
- * (Num-2) (Appendix B.2.1) Two.
 - * (Num-2+) (Appendix B.2.2) Two or more.
3. (Shares) (Appendix B.3) How should multiple key shares (public keys / ciphertexts) be conveyed?
- * (Shares-Concat) (Appendix B.3.1) Concatenate each combination of key shares.
 - * (Shares-Multiple) (Appendix B.3.2) Send individual key shares for each algorithm.
 - * (Shares-Ext-Additional) (Appendix B.3.3) Use an extension to convey key shares for component algorithms.
4. (Comb) (Appendix B.4) How should multiple shared secrets be combined?
- * (Comb-Concat) (Appendix B.4.1) Concatenate the shared secrets then use directly in the TLS 1.3 key schedule.
 - * (Comb-KDF-1) (Appendix B.4.2) and (Comb-KDF-2) (Appendix B.4.3) KDF the shared secrets together, then use the output in the TLS 1.3 key schedule.
 - * (Comb-XOR) (Appendix B.4.4) XOR the shared secrets then use directly in the TLS 1.3 key schedule.

- * (Comb-Chain) (Appendix B.4.5) Extend the TLS 1.3 key schedule so that there is a stage of the key schedule for each shared secret.
- * (Comb-AltInput) (Appendix B.4.6) Use the second shared secret in an alternate (otherwise unused) input in the TLS 1.3 key schedule.

B.1. (Neg) How to negotiate hybridization and component algorithms?

B.1.1. Key exchange negotiation in TLS 1.3

Recall that in TLS 1.3, the key exchange mechanism is negotiated via the "supported_groups" extension. The "NamedGroup" enum is a list of standardized groups for Diffie-Hellman key exchange, such as "secp256r1", "x25519", and "ffdhe2048".

The client, in its "ClientHello" message, lists its supported mechanisms in the "supported_groups" extension. The client also optionally includes the public key of one or more of these groups in the "key_share" extension as a guess of which mechanisms the server might accept in hopes of reducing the number of round trips.

If the server is willing to use one of the client's requested mechanisms, it responds with a "key_share" extension containing its public key for the desired mechanism.

If the server is not willing to use any of the client's requested mechanisms, the server responds with a "HelloRetryRequest" message that includes an extension indicating its preferred mechanism.

B.1.2. (Neg-Ind) Negotiating component algorithms individually

In these three approaches, the parties negotiate which traditional algorithm and which next-gen algorithm to use independently. The "NamedGroup" enum is extended to include algorithm identifiers for each next-gen algorithm.

B.1.2.1. (Neg-Ind-1)

The client advertises two lists to the server: one list containing its supported traditional mechanisms (e.g. via the existing "ClientHello" "supported_groups" extension), and a second list containing its supported next-generation mechanisms (e.g., via an additional "ClientHello" extension). A server could then select one algorithm from the traditional list, and one algorithm from the next-generation list. (This is the approach in [SCHANCK].)

B.1.2.2. (Neg-Ind-2)

The client advertises a single list to the server which contains both its traditional and next-generation mechanisms (e.g., all in the existing "ClientHello" "supported_groups" extension), but with some external table provides a standardized mapping of those mechanisms as either "traditional" or "next-generation". A server could then select two algorithms from this list, one from each category.

B.1.2.3. (Neg-Ind-3)

The client advertises a single list to the server delimited into sublists: one for its traditional mechanisms and one for its next-generation mechanisms, all in the existing "ClientHello" "supported_groups" extension, with a special code point serving as a delimiter between the two lists. For example, "supported_groups = secp256r1, x25519, delimiter, nextgen1, nextgen4".

B.1.3. (Neg-Comb) Negotiating component algorithms as a combination

In these three approaches, combinations of key exchange mechanisms appear as a single monolithic block; the parties negotiate which of several combinations they wish to use.

B.1.3.1. (Neg-Comb-1)

The "NamedGroup" enum is extended to include algorithm identifiers for each *combination* of algorithms desired by the working group. There is no "internal structure" to the algorithm identifiers for each combination, they are simply new code points assigned arbitrarily. The client includes any desired combinations in its "ClientHello" "supported_groups" list, and the server picks one of these. This is the approach in [KIEFER] and [OQS-111].

B.1.3.2. (Neg-Comb-2)

The "NamedGroup" enum is extended to include algorithm identifiers for each next-gen algorithm. Some additional field/extension is used to convey which combinations the parties wish to use. For example, in [WHYTE13], there are distinguished "NamedGroup" called "hybrid_marker 0", "hybrid_marker 1", "hybrid_marker 2", etc. This is complemented by a "HybridExtension" which contains mappings for each numbered "hybrid_marker" to the set of component key exchange algorithms (2 or more) for that proposed combination.

B.1.3.3. (Neg-Comb-3)

The client lists combinations in "supported_groups" list, using a special delimiter to indicate combinations. For example, "supported_groups = combo_delimiter, secp256r1, nextgen1, combo_delimiter, secp256r1, nextgen4, standalone_delimiter, secp256r1, x25519" would indicate that the client's highest preference is the combination secp256r1+nextgen1, the next highest preference is the combination secp256r1+nextgen4, then the single algorithm secp256r1, then the single algorithm x25519. A hybrid-aware server would be able to parse these; a hybrid-unaware server would see "unknown, secp256r1, unknown, unknown, secp256r1, unknown, unknown, secp256r1, x25519", which it would be able to process, although there is the potential that every "projection" of a hybrid list that is tolerable to a client does not result in list that is tolerable to the client.

B.1.4. Benefits and drawbacks

Combinatorial explosion. (Neg-Comb-1) (Appendix B.1.3.1) requires new identifiers to be defined for each desired combination. The other 4 options in this section do not.

Extensions. (Neg-Ind-1) (Appendix B.1.2.1) and (Neg-Comb-2) (Appendix B.1.3.2) require new extensions to be defined. The other options in this section do not.

New logic. All options in this section except (Neg-Comb-1) (Appendix B.1.3.1) require new logic to process negotiation.

Matching security levels. (Neg-Ind-1) (Appendix B.1.2.1), (Neg-Ind-2) (Appendix B.1.2.2), (Neg-Ind-3) (Appendix B.1.2.3), and (Neg-Comb-2) (Appendix B.1.3.2) allow algorithms of different claimed security level from their corresponding lists to be combined. For example, this could result in combining ECDH secp256r1 (classical security level 128) with NewHope-1024 (classical security level 256). Implementations dissatisfied with a mismatched security levels must either accept this mismatch or attempt to renegotiate. (Neg-Ind-1) (Appendix B.1.2.1), (Neg-Ind-2) (Appendix B.1.2.2), and (Neg-Ind-3) (Appendix B.1.2.3) give control over the combination to the server; (Neg-Comb-2) (Appendix B.1.3.2) gives control over the combination to the client. (Neg-Comb-1) (Appendix B.1.3.1) only allows standardized combinations, which could be set by TLS working group to have matching security (provided security estimates do not evolve separately).

Backwards-compability. TLS 1.3-compliant hybrid-unaware servers should ignore unrecognized elements in "supported_groups" (Neg-Ind-2)

(Appendix B.1.2.2), (Neg-Ind-3) (Appendix B.1.2.3), (Neg-Comb-1) (Appendix B.1.3.1), (Neg-Comb-2) (Appendix B.1.3.2) and unrecognized "ClientHello" extensions (Neg-Ind-1) (Appendix B.1.2.1), (Neg-Comb-2) (Appendix B.1.3.2). In (Neg-Ind-3) (Appendix B.1.2.3) and (Neg-Comb-3) (Appendix B.1.3.3), a server that is hybrid-unaware will ignore the delimiters in "supported_groups", and thus might try to negotiate an algorithm individually that is only meant to be used in combination; depending on how such an implementation is coded, it may also encounter bugs when the same element appears multiple times in the list.

B.2. (Num) How many component algorithms to combine?

B.2.1. (Num-2) Two

Exactly two algorithms can be combined together in hybrid key exchange. This is the approach taken in [KIEFER] and [SCHANCK].

B.2.2. (Num-2+) Two or more

Two or more algorithms can be combined together in hybrid key exchange. This is the approach taken in [WHYTE13].

B.2.3. Benefits and Drawbacks

Restricting the number of component algorithms that can be hybridized to two substantially reduces the generality required. On the other hand, some adopters may want to further reduce risk by employing multiple next-gen algorithms built on different cryptographic assumptions.

B.3. (Shares) How to convey key shares?

In ECDH ephemeral key exchange, the client sends its ephemeral public key in the "key_share" extension of the "ClientHello" message, and the server sends its ephemeral public key in the "key_share" extension of the "ServerHello" message.

For a general key encapsulation mechanism used for ephemeral key exchange, we imagine that that client generates a fresh KEM public key / secret pair for each connection, sends it to the client, and the server responds with a KEM ciphertext. For simplicity and consistency with TLS 1.3 terminology, we will refer to both of these types of objects as "key shares".

In hybrid key exchange, we have to decide how to convey the client's two (or more) key shares, and the server's two (or more) key shares.

B.3.1. (Shares-Concat) Concatenate key shares

The client concatenates the bytes representing its two key shares and uses this directly as the "key_exchange" value in a "KeyShareEntry" in its "key_share" extension. The server does the same thing. Note that the "key_exchange" value can be an octet string of length at most $2^{16}-1$. This is the approach taken in [KIEFER], [OQS-111], and [WHYTE13].

B.3.2. (Shares-Multiple) Send multiple key shares

The client sends multiple key shares directly in the "client_shares" vectors of the "ClientHello" "key_share" extension. The server does the same. (Note that while the existing "KeyShareClientHello" struct allows for multiple key share entries, the existing "KeyShareServerHello" only permits a single key share entry, so some modification would be required to use this approach for the server to send multiple key shares.)

B.3.3. (Shares-Ext-Additional) Extension carrying additional key shares

The client sends the key share for its traditional algorithm in the original "key_share" extension of the "ClientHello" message, and the key share for its next-gen algorithm in some additional extension in the "ClientHello" message. The server does the same thing. This is the approach taken in [SCHANCK].

B.3.4. Benefits and Drawbacks

Backwards compatibility. (Shares-Multiple) (Appendix B.3.2) is fully backwards compatible with non-hybrid-aware servers. (Shares-Ext-Additional) (Appendix B.3.3) is backwards compatible with non-hybrid-aware servers provided they ignore unrecognized extensions. (Shares-Concat) (Appendix B.3.1) is backwards-compatible with non-hybrid aware servers, but may result in duplication / additional round trips (see below).

Duplication versus additional round trips. If a client wants to offer multiple key shares for multiple combinations in order to avoid retry requests, then the client may ended up sending a key share for one algorithm multiple times when using (Shares-Ext-Additional) (Appendix B.3.3) and (Shares-Concat) (Appendix B.3.1). (For example, if the client wants to send an ECDH-secp256r1 + McEliece123 key share, and an ECDH-secp256r1 + NewHope1024 key share, then the same ECDH public key may be sent twice. If the client also wants to offer a traditional ECDH-only key share for non-hybrid-aware implementations and avoid retry requests, then that same ECDH public

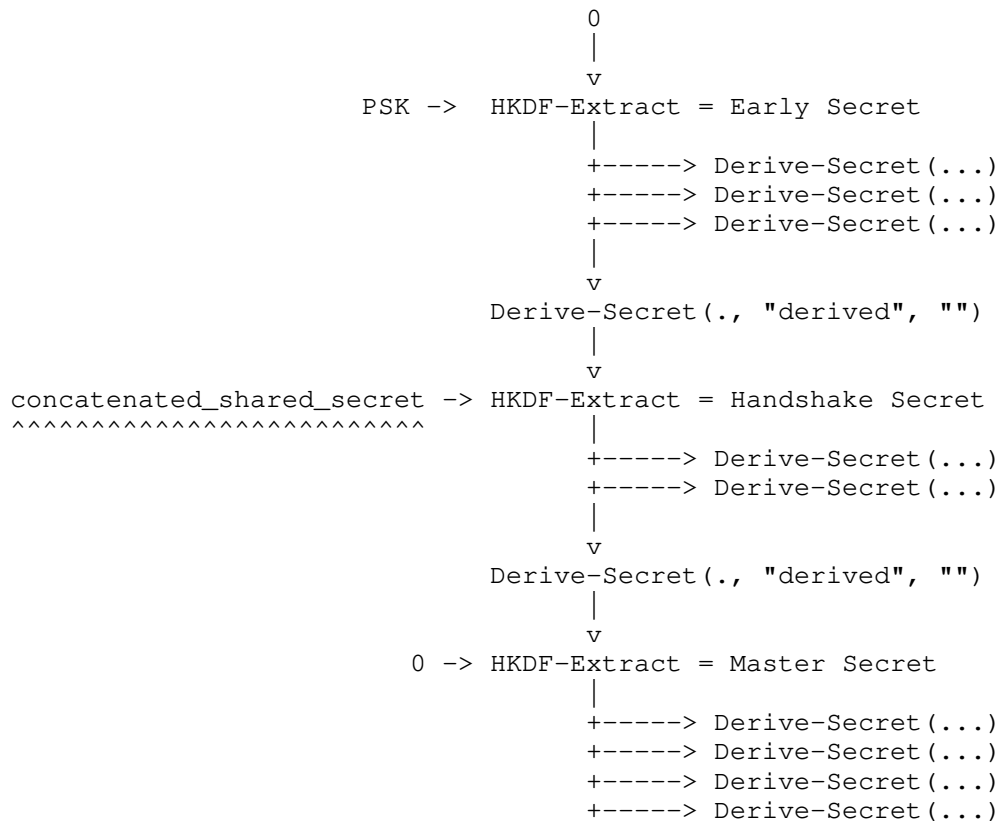
key may be sent another time.) (Shares-Multiple) (Appendix B.3.2)
does not result in duplicate key shares.

B.4. (Comb) How to use keys?

Each component key exchange algorithm establishes a shared secret. These shared secrets must be combined in some way that achieves the "hybrid" property: the resulting secret is secure as long as at least one of the component key exchange algorithms is unbroken.

B.4.1. (Comb-Concat) Concatenate keys

Each party concatenates the shared secrets established by each component algorithm in an agreed-upon order, then feeds that through the TLS key schedule. In the context of TLS 1.3, this would mean using the concatenated shared secret in place of the (EC)DHE input to the second call to "HKDF-Extract" in the TLS 1.3 key schedule:



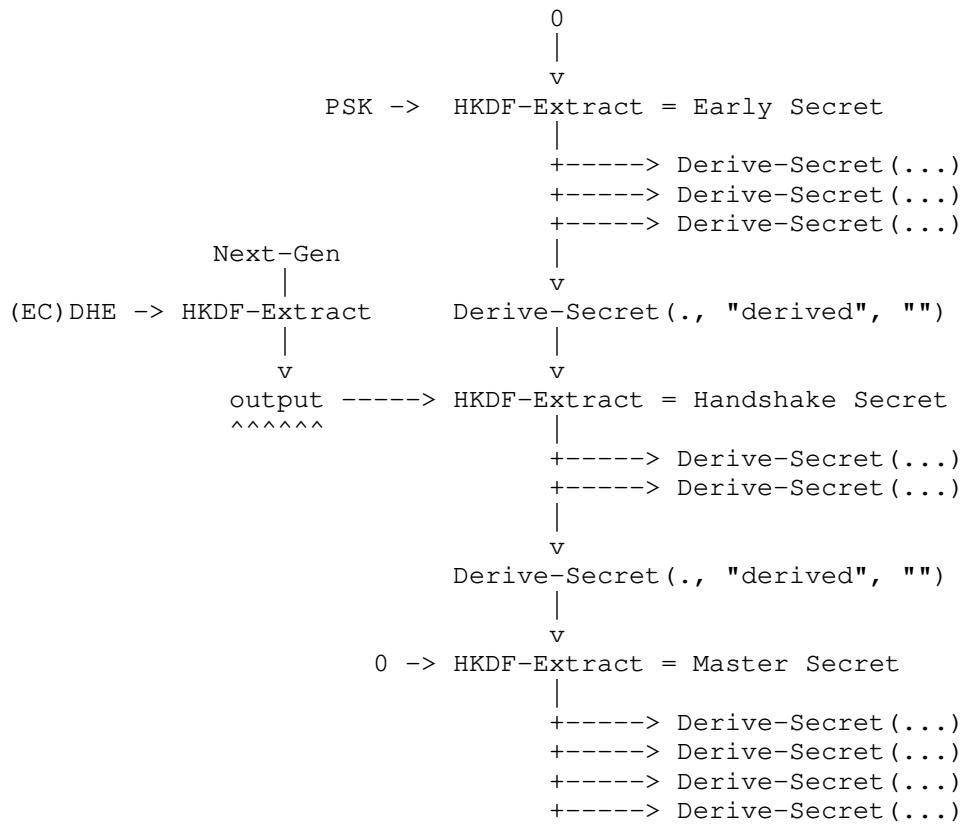
This is the approach used in [KIEFER], [OQS-111], and [WHYTE13].

[GIACON] analyzes the security of applying a KDF to concatenated KEM shared secrets, but their analysis does not exactly apply here since the transcript of ciphertexts is included in the KDF application (though it should follow relatively straightforwardly).

[BINDEL] analyzes the security of the (Comb-Concat) approach as abstracted in their "dualPRF" combiner. They show that, if the component KEMs are IND-CPA-secure (or IND-CCA-secure), then the values output by "Derive-Secret" are IND-CPA-secure (respectively, IND-CCA-secure). An important aspect of their analysis is that each ciphertext is input to the final PRF calls; this holds for TLS 1.3 since the "Derive-Secret" calls that derive output keys (application traffic secrets, and exporter and resumption master secrets) include the transcript hash as input.

B.4.2. (Comb-KDF-1) KDF keys

Each party feeds the shared secrets established by each component algorithm in an agreed-upon order into a KDF, then feeds that through the TLS key schedule. In the context of TLS 1.3, this would mean first applying "HKDF-Extract" to the shared secrets, then using the output in place of the (EC)DHE input to the second call to "HKDF-Extract" in the TLS 1.3 key schedule:

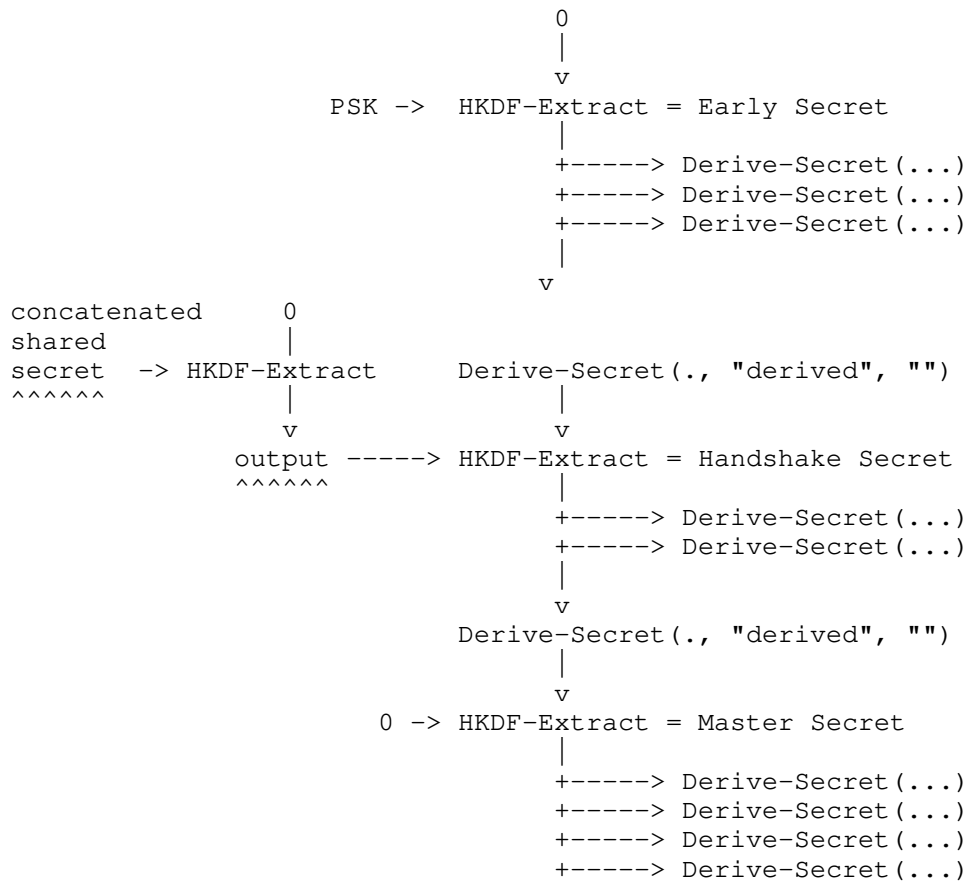


B.4.3. (Comb-KDF-2) KDF keys

Each party concatenates the shared secrets established by each component algorithm in an agreed-upon order then feeds that into a KDF, then feeds the result through the TLS key schedule.

Compared with (Comb-KDF-1) (Appendix B.4.2), this method concatenates the (2 or more) shared secrets prior to input to the KDF, whereas (Comb-KDF-1) puts the (exactly 2) shared secrets in the two different input slots to the KDF.

Compared with (Comb-Concat) (Appendix B.4.1), this method has an extract KDF application. While this adds computational overhead, this may provide a cleaner abstraction of the hybridization mechanism for the purposes of formal security analysis.



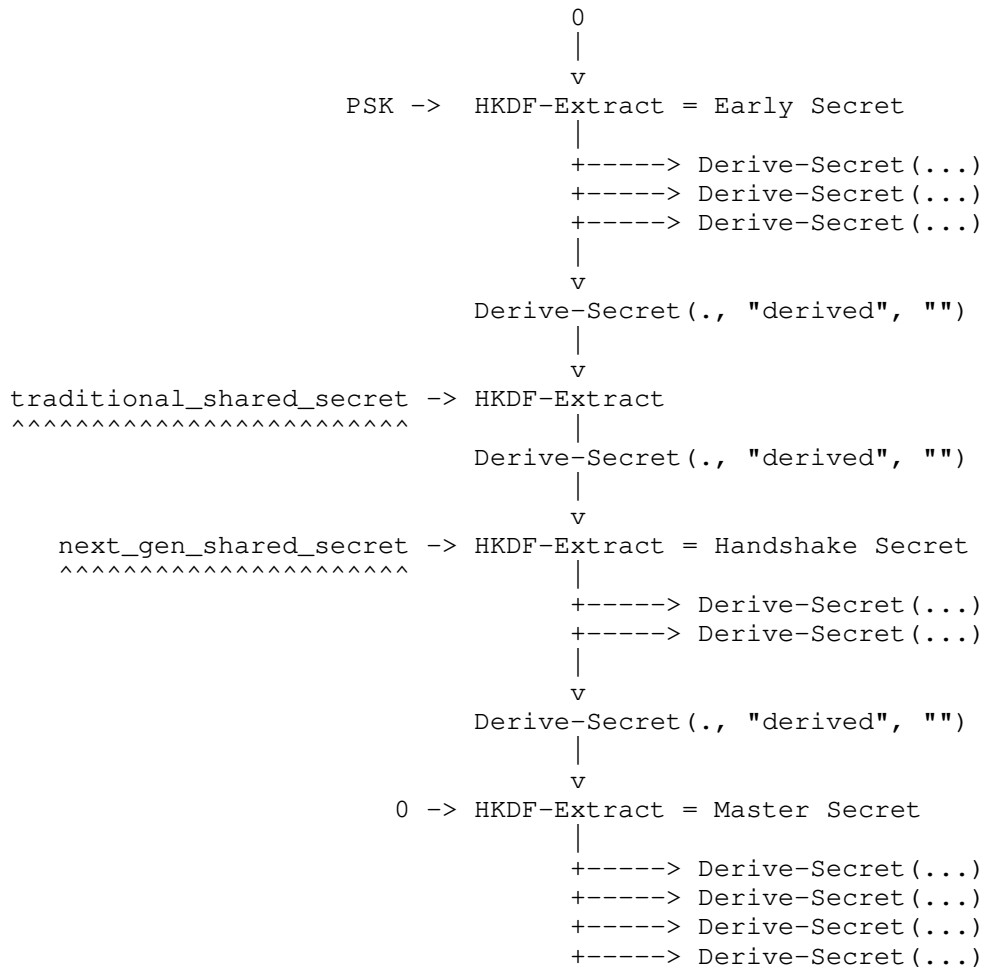
B.4.4. (Comb-XOR) XOR keys

Each party XORs the shared secrets established by each component algorithm (possibly after padding secrets of different lengths), then feeds that through the TLS key schedule. In the context of TLS 1.3, this would mean using the XORed shared secret in place of the (EC)DHE input to the second call to "HKDF-Extract" in the TLS 1.3 key schedule.

[GIACON] analyzes the security of applying a KDF to the XORed KEM shared secrets, but their analysis does not quite apply here since the transcript of ciphertexts is included in the KDF application (though it should follow relatively straightforwardly).

B.4.5. (Comb-Chain) Chain of KDF applications for each key

Each party applies a chain of key derivation functions to the shared secrets established by each component algorithm in an agreed-upon order; roughly speaking: $F(k_1 || F(k_2))$. In the context of TLS 1.3, this would mean extending the key schedule to have one round of the key schedule applied for each component algorithm's shared secret:



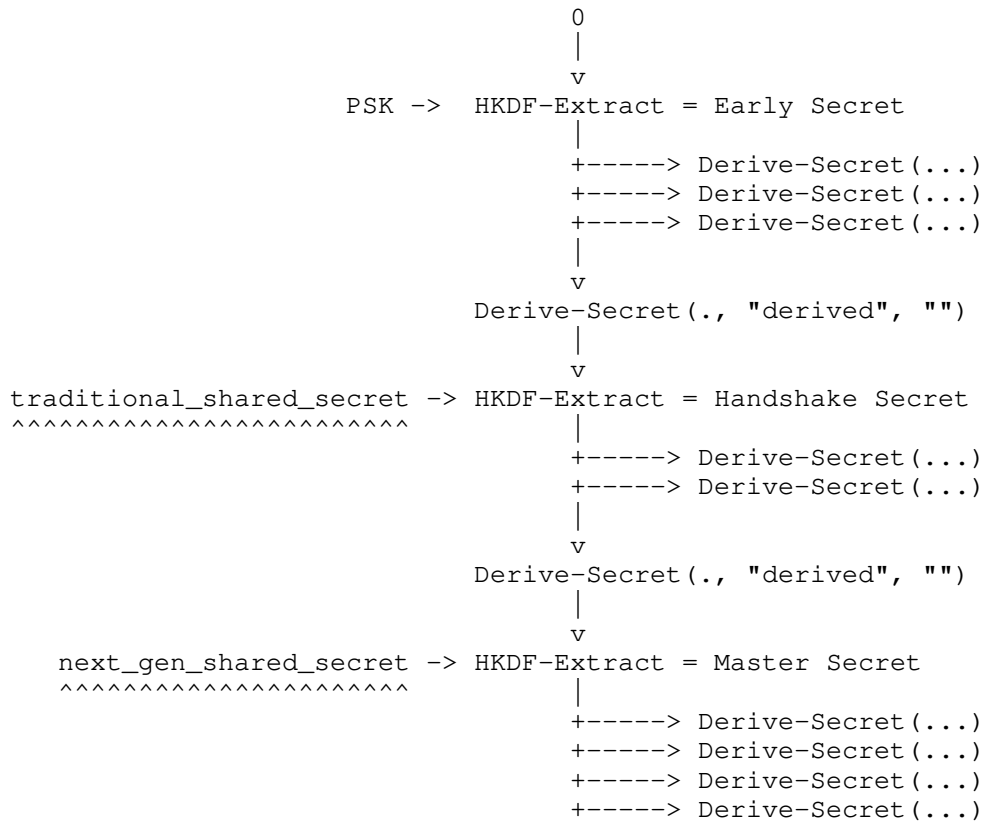
This is the approach used in [SCHANCK].

[BINDEL] analyzes the security of this approach as abstracted in their nested dual-PRF "N" combiner, showing a similar result as for the dualPRF combiner that it preserves IND-CPA (or IND-CCA) security.

Again their analysis depends on each ciphertext being input to the final PRF ("Derive-Secret") calls, which holds for TLS 1.3.

B.4.6. (Comb-AltInput) Second shared secret in an alternate KDF input

In the context of TLS 1.3, the next-generation shared secret is used in place of a currently unused input in the TLS 1.3 key schedule, namely replacing the "0" "IKM" input to the final "HKDF-Extract":



This approach is not taken in any of the known post-quantum/hybrid TLS drafts. However, it bears some similarities to the approach for using external PSKs in [EXTERN-PSK].

B.4.7. Benefits and Drawbacks

New logic. While (Comb-Concat) (Appendix B.4.1), (Comb-KDF-1) (Appendix B.4.2), and (Comb-KDF-2) (Appendix B.4.3) require new logic to compute the concatenated shared secret, this value can then be used by the TLS 1.3 key schedule without changes to the key schedule

logic. In contrast, (Comb-Chain) (Appendix B.4.5) requires the TLS 1.3 key schedule to be extended for each extra component algorithm.

Philosophical. The TLS 1.3 key schedule already applies a new stage for different types of keying material (PSK versus (EC)DHE), so (Comb-Chain) (Appendix B.4.5) continues that approach.

Efficiency. (Comb-KDF-1) (Appendix B.4.2), (Comb-KDF-2) (Appendix B.4.3), and (Comb-Chain) (Appendix B.4.5) increase the number of KDF applications for each component algorithm, whereas (Comb-Concat) (Appendix B.4.1) and (Comb-AltInput) (Appendix B.4.6) keep the number of KDF applications the same (though with potentially longer inputs).

Extensibility. (Comb-AltInput) (Appendix B.4.6) changes the use of an existing input, which might conflict with other future changes to the use of the input.

More than 2 component algorithms. The techniques in (Comb-Concat) (Appendix B.4.1) and (Comb-Chain) (Appendix B.4.5) can naturally accommodate more than 2 component shared secrets since there is no distinction to how each shared secret is treated. (Comb-AltInput) (Appendix B.4.6) would have to make some distinct, since the 2 component shared secrets are used in different ways; for example, the first shared secret is used as the "IKM" input in the 2nd "HKDF-Extract" call, and all subsequent shared secrets are concatenated to be used as the "IKM" input in the 3rd "HKDF-Extract" call.

Authors' Addresses

Douglas Stebila
University of Waterloo

Email: dstebila@uwaterloo.ca

Scott Fluhrer
Cisco Systems

Email: sfluhrer@cisco.com

Shay Gueron
University of Haifa and Amazon Web Services

Email: shay.gueron@gmail.com

TLS
Internet-Draft
Intended status: Standards Track
Expires: September 28, 2019

M. Thomson
Mozilla
March 27, 2019

Suppressing Intermediate Certificates in TLS
draft-thomson-tls-sic-00

Abstract

A TLS client that has access to the complete set of published intermediate certificates can inform servers of this fact so that the server can avoid sending intermediates, reducing the size of the TLS handshake.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 28, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	2
2. Terms and Definitions	2
3. Got Intermediates Flag	2
4. Security Considerations	3
5. IANA Considerations	3
6. References	3
6.1. Normative References	3
6.2. Informative References	3
Author's Address	4

1. Introduction

In some uses of public key infrastructure (PKI) intermediate certificates are used to sign end-entity certificates. In the web PKI, clients require that certificate authorities disclose all intermediate certificates that they create. Though the set of intermediate certificates is large, the size is bounded, so it is possible to provide a complete set of certificates.

For a client that has all intermediates, having the server send intermediates in the TLS handshake increases the size of the handshake unnecessarily. This document creates a signal that a client can send that informs the server that it has a complete set of intermediates. A server that receives this signal can limit the certificate chain it sends to just the end-entity certificate, saving on handshake size.

This mechanism is intended to be complementary with certificate compression [COMPRESS] in that it reduces the size of the handshake.

2. Terms and Definitions

The keywords "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.

3. Got Intermediates Flag

A client that believes that it has a current, complete set of intermediate certificates sends the `tls_flags` extension [TLS-FLAGS] with the `0xTBD` flag set to 1. A server can also set the flag in a `CertificateRequest` extension.

A server that receives a value of 1 in the 0xTBD flag from a ClientHello message SHOULD omit all certificates other than the end-entity certificate from its Certificate message. A client that receives a value of 1 in the 0xTBD flag in a CertificateRequest message SHOULD omit all certificates other than the end-entity certificate from the Certificate message that it sends in response.

The 0xTBD flag can only be send in a ClientHello or CertificateRequest message. Endpoints that receive a value of 1 in any other handshake message MUST generate a fatal `illegal_parameter` alert.

4. Security Considerations

This creates an unencrypted signal that might be used to identify which clients believe that they have all intermediates. This might allow cilents to be more effectively fingerprinted by peers and any elements on the network path.

5. IANA Considerations

This document registers the 0xTBD flag in the registry created by [TLS-FLAGS].

6. References

6.1. Normative References

- [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>>.
- [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>>.
- [TLS-FLAGS] Nir, Y., "A Flags Extension for TLS 1.3", draft-nir-tls-tlsflags-00 (work in progress), March 2019.

6.2. Informative References

- [COMPRESS] Ghedini, A. and V. Vasiliev, "TLS Certificate Compression", draft-ietf-tls-certificate-compression-04 (work in progress), October 2018.

Author's Address

Martin Thomson
Mozilla

Email: mt@lowentropy.net

TLS
Internet-Draft
Updates: 6347 (if approved)
Intended status: Standards Track
Expires: September 3, 2020

T. Fossati
H. Tschofenig, Ed.
Arm Limited
March 2, 2020

Return Routability Check for DTLS 1.2 and DTLS 1.3
draft-tschofenig-tls-dtls-rrc-01

Abstract

This document specifies a return routability check for use in context of the Connection ID (CID) construct for the Datagram Transport Layer Security (DTLS) protocol versions 1.2 and 1.3.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 3, 2020.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

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

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Introduction	2
2. Conventions and Terminology	3
3. The Return Routability Check Message	3
4. RRC Example	4
5. Security and Privacy Considerations	7
6. IANA Considerations	7
7. Open Issues	7
8. Normative References	7
Appendix A. History	8
Appendix B. Acknowledgements	8
Authors' Addresses	8

1. Introduction

In "classical" DTLS, selecting a security context of an incoming DTLS record is accomplished with the help of the 5-tuple, i.e. source IP address, source port, transport protocol, destination IP address, and destination port. Changes to this 5 tuple can happen for a variety reasons over the lifetime of the DTLS session. In the IoT context, NAT rebinding is common with sleepy devices. Other examples include end host mobility and multi-homing. Without CID, if the source IP address and/or source port changes during the lifetime of an ongoing DTLS session then the receiver will be unable to locate the correct security context. As a result, the DTLS handshake has to be re-run. Of course, it is not necessary to re-run the full handshake if session resumption is supported and negotiated.

A CID is an identifier carried in the record layer header of a DTLS datagram that gives the receiver additional information for selecting the appropriate security context. The CID mechanism has been specified in [I-D.ietf-tls-dtls-connection-id] for DTLS 1.2 and in [I-D.ietf-tls-dtls13] for DTLS 1.3.

Section 6 of [I-D.ietf-tls-dtls-connection-id] describes how the use of CID increases the attack surface by providing both on-path and off-path attackers an opportunity for (D)DoS. It then goes on describing the steps a DTLS principal must take when a record with a CID is received that has a source address (and/or port) different from the one currently associated with the DTLS connection. However, the actual mechanism for ensuring that the new peer address is willing to receive and process DTLS records is left open. This document standardizes a return routability check (RRC) as part of the DTLS protocol itself.

The return routability check is performed by the receiving peer before the CID-to-IP address/port binding is updated in that peer's session state database. This is done in order to provide more confidence to the receiving peer that the sending peer is reachable at the indicated address and port.

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.

This document assumes familiarity with the CID format and protocol defined for DTLS 1.2 [I-D.ietf-tls-dtls-connection-id] and for DTLS 1.3 [I-D.ietf-tls-dtls13].

3. The Return Routability Check Message

When a record with CID is received that has the source address of the enclosing UDP datagram different from the one previously associated with that CID, the receiver MUST NOT update its view of the peer's IP address and port number with the source specified in the UDP datagram before cryptographically validating the enclosed record(s) but instead perform a return routability check.

```
enum {
    invalid(0),
    change_cipher_spec(20),
    alert(21),
    handshake(22),
    application_data(23),
    heartbeat(24), /* RFC 6520 */
    return_routability_check(TBD), /* NEW */
    (255)
} ContentType;
```

```
struct {  
    opaque cookie<1..2^16-1>;  
} Cookie;  
  
struct {  
    Cookie cookie;  
} return_routability_check;
```

The newly introduced `return_routability_check` message contains a cookie. The semantic of the cookie is similar to the cookie used in the `HelloRetryRequest` message defined in [RFC8446].

The `return_routability_check` message MUST be authenticated and encrypted using the currently active security context.

The receiver that observes the peer's address and or port update MUST stop sending any buffered application data (or limit the sending rate to a TBD threshold) and initiate the return routability check that proceeds as follows:

1. A cookie is placed in the `return_routability_check` message;
2. The message is sent to the observed new address and a timeout `T` is started;
3. The peer endpoint, after successfully verifying the received `return_routability_check` message echoes it back;
4. When the initiator receives and verifies the `return_routability_check` message, it updates the peer address binding;
5. If `T` expires, or the address confirmation fails, the peer address binding is not updated.

After this point, any pending send operation is resumed to the bound peer address.

4. RRC Example

The example shown in Figure 1 illustrates a client and a server exchanging application payloads protected by DTLS with an unilaterally used CIDs. At some point in the communication interaction the IP address used by the client changes and, thanks to the CID usage, the security context to interpret the record is successfully located by the server. However, the server wants to test the reachability of the client at his new IP address, to avoid

being abused (e.g., as an amplifier) by an attacker impersonating the client.

```

Client                                     Server
-----
Application Data      =====>
<CID=100>
Src-IP=A
Dst-IP=Z

                                     <=====
                                     Application Data
                                     Src-IP=Z
                                     Dst-IP=A

                                     <<----->>
                                     <<  Some  >>
                                     <<  Time  >>
                                     <<  Later  >>
                                     <<----->>

Application Data      =====>
<CID=100>
Src-IP=B
Dst-IP=Z

                                     <<< Unverified IP
                                     Address B >>

                                     <----- Return Routability Check
                                     (cookie)
                                     Src-IP=Z
                                     Dst-IP=B

Return Routability Check  ----->
(cookie)
Src-IP=B
Dst-IP=Z

                                     <<< IP Address B
                                     Verified >>

                                     <=====
                                     Application Data
                                     Src-IP=Z
                                     Dst-IP=B

```

Figure 1: Return Routability Example

5. Security and Privacy Considerations

Note that the return routability checks do not protect against flooding of third-parties if the attacker is on-path, as the attacker can redirect the return routability checks to the real peer (even if those datagrams are cryptographically authenticated). On-path adversaries can, in general, pose a harm to connectivity.

6. IANA Considerations

IANA is requested to allocate an entry to the existing TLS "ContentType" registry, for the return_routability_check(TBD) defined in this document.

7. Open Issues

- Should the return routability check use separate sequence numbers and replay windows?
- Should the heartbeat message be re-used instead of the proposed new message exchange?

8. Normative References

- [I-D.ietf-tls-dtls-connection-id]
Rescorla, E., Tschofenig, H., and T. Fossati, "Connection Identifiers for DTLS 1.2", draft-ietf-tls-dtls-connection-id-07 (work in progress), October 2019.
- [I-D.ietf-tls-dtls13]
Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", draft-ietf-tls-dtls13-34 (work in progress), November 2019.
- [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>>.
- [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>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

Appendix A. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

- 01: Removed text that overlapped with draft-ietf-tls-dtls-connection-id
- 00: Initial version

Appendix B. Acknowledgements

We would like to thank Achim Kraus, Hanno Becker and Manuel Pegourie-Gonnard for their input to this document.

Authors' Addresses

Thomas Fossati
Arm Limited

EMail: thomas.fossati@arm.com

Hannes Tschofenig (editor)
Arm Limited

EMail: hannes.tschofenig@arm.com