The Datagram Transport Layer Security (DTLS) Protocol Version 1.3
draft-ietf-tls-dtls13-33

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.

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 April 13, 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.

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 .............................................. 3
2. Conventions and Terminology ............................. 4
3. DTLS Design Rationale and Overview ..................... 5
   3.1. Packet Loss ........................................ 6
   3.2. Reordering ......................................... 7
   3.3. Message Size ....................................... 7
   3.4. Replay Detection .................................. 7
4. The DTLS Record Layer ................................... 7
   4.1. Determining the Header Format ....................... 11
   4.2. Sequence Number and Epoch ......................... 11
      4.2.1. Processing Guidelines .......................... 11
      4.2.2. Reconstructing the Sequence Number and Epoch ... 12
      4.2.3. Sequence Number Encryption .................... 13
   4.3. Transport Layer Mapping ............................ 14
   4.4. PMTU Issues ...................................... 14
   4.5. Record Payload Protection ......................... 16
      4.5.1. Anti-Replay .................................. 16
      4.5.2. Handling Invalid Records ..................... 17
5. The DTLS Handshake Protocol ............................ 17
   5.1. Denial-of-Service Countermeasures ................. 18
   5.2. DTLS Handshake Message Format ................... 21
   5.3. ClientHello Message ................................ 22
   5.4. Handshake Message Fragmentation and Reassembly .... 23
   5.5. End Of Early Data ................................. 24
1. Introduction

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

The primary goal of the TLS protocol is to provide privacy and data integrity between two communicating peers. The TLS protocol is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol. However, TLS must run over a reliable transport channel - typically TCP [RFC0793].
There are applications that use UDP [RFC0768] as a transport and to offer communication security protection for those applications the Datagram Transport Layer Security (DTLS) protocol has been designed. DTLS is deliberately designed to be as similar to TLS as possible, both to minimize new security invention and to maximize the amount of code and infrastructure reuse.

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

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

2. Conventions and Terminology

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

The following terms are used:

- client: The endpoint initiating the DTLS connection.
- connection: A transport-layer connection between two endpoints.
- endpoint: Either the client or server of the connection.
- handshake: An initial negotiation between client and server that establishes the parameters of their transactions.
- peer: An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is remote to the primary subject of discussion.
- receiver: An endpoint that is receiving records.
- sender: An endpoint that is transmitting records.
- session: An association between a client and a server resulting from a handshake.
- server: The endpoint which did not initiate the DTLS connection.
- CID: Connection ID

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

Figures in this document illustrate various combinations of the DTLS protocol exchanges and the symbols have the following meaning:
- `+` indicates noteworthy extensions sent in the previously noted message.
- `*` indicates optional or situation-dependent messages/extensions that are not always sent.
- `{}` indicates messages protected using keys derived from a [sender]_handshake_traffic_secret.
- `[]` indicates messages protected using keys derived from traffic_secret_N.

3. DTLS Design Rationale and Overview

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

TLS cannot be used directly in datagram environments for 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.
2. The TLS handshake is a lock-step cryptographic handshake. Messages must be transmitted and received in a defined order; any other order is an error. DTLS handshake messages are also assigned sequence numbers to enable reassembly in the correct order in case datagrams are lost or reordered.

3. During the handshake, messages are implicitly acknowledged by other handshake messages, but the last flight of messages and post-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 that uses the TLS 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:

```
Client                                   Server
------                                   -----
ClientHello           ------>            
                       X<-- HelloRetryRequest
                              (lost)
[Timer Expires]
ClientHello           ------>            (retransmit)
```

Figure 1: DTLS retransmission example

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

Note that timeout and retransmission do not apply to the HelloRetryRequest since this would require creating state on the server. The HelloRetryRequest is designed to be small enough that it will not itself be fragmented, thus avoiding concerns about interleaving multiple HelloRetryRequests.

3.2. 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. Message Size

TLS and DTLS handshake messages can be quite large (in theory up to $2^{24}-1$ bytes, in practice many kilobytes). By contrast, UDP datagrams are often limited to less than 1500 bytes if IP fragmentation is not desired. In order to compensate for this limitation, each DTLS handshake message may be fragmented over several DTLS records, each of which is intended to fit in a single IP packet. 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 record layer is different from the TLS 1.3 record layer.

1. The DTLSRecordstructure 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.

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

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

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

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

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

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

Figure 2: DTLS 1.3 Record Format

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

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

The DTLSCiphertext header is tightly bit-packed, as shown below:
Figure 3: DTLS 1.3 CipherText Header

Fixed Bits: The three high bits of the first byte of the DTLSCiphertext header are set to 001.

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.

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 concept is described in [DTLS-CID]. An example can be found in Section 9.1.

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

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

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

Figure 4 illustrates different record layer header types.
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 DTLSPlaintext format records without length fields in the same datagram.

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

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 above is used as it appears on the wire as the additional data value for the AEAD function. 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. Determining the Header Format

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

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

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

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

4.2. Sequence Number and Epoch

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. In general, implementations SHOULD discard packets from earlier epochs, but if packet loss causes noticeable problems implementations MAY choose to retain keying material from previous epochs for up to the default MSL specified for TCP [RFC0793] to allow for packet reordering.
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 packets, though when DTLS is used over reliable transports (e.g., SCTP [RFC4960]), they SHOULD be buffered and processed once the handshake completes. Note that TLS’s restrictions on when packets may be sent still apply, and the receiver treats the packets as if they were sent in the right order.

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

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

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

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

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

$$\text{Mask} = \text{AES-ECB}(\text{sn key}, \text{Ciphertext}[0..15])$$

When the AEAD is based on ChaCha20, then the mask is generated by treating the first 12 bytes of the ciphertext as the Nonce and the next 4 bytes as the counter, passing them to the ChaCha20 block function (Section 2.3 of [CHACHA]):

$$\text{Mask} = \text{ChaCha20}(\text{sn key}, \text{Ciphertext}[0..12], \text{Ciphertext}[13..15])$$

The sn_key is computed as follows:

$$[\text{sender}]_\text{sn key} = \text{HKDF-Expand-Label}(\text{Secret}, "sn", ",", \text{key_length})$$

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

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

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.

Note that sequence number encryption is only applied to the DTLS ciphertext structure and not to the DTLS plaintext 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 PMTU estimates obtained from the record layer.

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

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

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

4.4. PMTU Issues

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

- The DTLS record framing expands the datagram size, thus lowering the effective PMTU from the application’s perspective.
- In some implementations, the application may not directly talk to the network, in which case the DTLS stack may absorb ICMP [RFC1191] "Datagram Too Big" indications or ICMPv6 [RFC4443] "Packet Too Big" indications.

- The DTLS handshake messages can exceed the PMTU.

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

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

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

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

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

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.

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

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

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

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

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

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

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

4.5. Record Payload Protection

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

4.5.1. Anti-Replay

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

The received packet counter for a session MUST be initialized to zero when that session is established. For each received record, the receiver MUST verify that the record contains a sequence number that does not duplicate the sequence number of any other record received during the lifetime of the session. This check SHOULD happen after deprotecting the packet; otherwise the packet discard might itself serve as a timing channel for the record number. Note that decompressing the records number is still a potential timing channel for the record number, though a less powerful one than whether it 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 on the session. Records that contain sequence numbers lower than the "left" edge of the window are rejected. Packets falling within the window are checked against a list of received packets within the window. An efficient means for performing this check, based on the use of a bit mask, is described in Section 3.4.3 of [RFC4303]. If the received record falls within the window and is new, or if the packet is to the right of the window, then the packet is new.

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

4.5.2. Handling Invalid Records

Unlike TLS, DTLS is resilient in the face of invalid records (e.g., invalid formatting, length, MAC, etc.). In general, invalid records SHOULD be silently discarded, thus preserving the association; however, an error MAY be logged for diagnostic purposes. Implementations which choose to generate an alert instead, MUST generate error alerts to avoid attacks where the attacker repeatedly probes the implementation to see how it responds to various types of error. Note that if DTLS is run over UDP, then any implementation which does this will be extremely susceptible to denial-of-service (DoS) attacks because UDP forgery is so easy. Thus, this practice is NOT RECOMMENDED for such transports, both to increase the reliability of DTLS service and to avoid the risk of spoofing attacks sending traffic to unrelated third parties.

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

5. The DTLS Handshake Protocol

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

1. To handle message loss, reordering, and fragmentation modifications to the handshake header are necessary.
2. Retransmission timers are introduced to handle message loss.
3. A new ACK content type has been added for reliable message delivery of handshake messages.
Note that TLS 1.3 already supports a cookie extension, which is used to prevent denial-of-service attacks. This DoS prevention mechanism is described in more detail below since UDP-based protocols are more vulnerable to amplification attacks than a connection-oriented transport like TCP that performs return-routability checks as part of the connection establishment.

DTLS implementations do not use the TLS 1.3 "compatibility mode" described in Section D.4 of [TLS13]. DTLS servers MUST NOT echo the "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 of the 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 it is the case for a Certificate message.

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

The DTLS 1.3 specification changes the way how cookies are exchanged compared to DTLS 1.2. DTLS 1.3 re-uses the HelloRetryRequest message
and conveys the cookie to the client via an extension. The client receiving the cookie uses the same extension to place the cookie subsequently into a ClientHello message. DTLS 1.2 on the other hand used a separate message, namely the HelloVerifyRequest, to pass a cookie to the client and did not utilize the extension mechanism. For backwards compatibility 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 5. Note that the figure focuses on the cookie exchange; all other extensions are omitted.

```
Client                        Server
------                        ------
ClientHello                   ---->
<----- HelloRetryRequest      + cookie
                  + cookie

[Rest of handshake]
```

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

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

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

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

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

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

One potential attack on this scheme is for the attacker to collect a number of cookies from different addresses where it controls endpoints and then reuse them to attack the server. The server can defend against this attack by changing the secret value frequently, thus invalidating those cookies. If the server wishes to allow legitimate clients to handshake through the transition (e.g., a client received a cookie with Secret 1 and then sent the second ClientHello after the server has changed to Secret 2), the server can have a limited window during which it accepts both secrets. [RFC7296] suggests adding a key identifier to cookies to detect this case. An alternative approach is simply to try verifying with both secrets. It is RECOMMENDED that servers implement a key rotation scheme that allows the server to manage keys with overlapping lifetime.

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

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

If a server receives a ClientHello with an invalid cookie, it MUST NOT 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).
5.2. DTLS Handshake Message Format

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

```
enum {
  client_hello(1),
  server_hello(2),
  new_session_ticket(4),
  end_of_early_data(5),
  encrypted_extensions(8),
  certificate(11),
  certificate_request(13),
  certificate_verify(15),
  finished(20),
  key_update(24),
  message_hash(254),
  (255)
} HandshakeType;
```

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

The first message each side transmits in each association always has message_seq = 0. Whenever a new message is generated, the message_seq value is incremented by one. When a message is retransmitted, the old message_seq value is re-used, i.e., not incremented. From the perspective of the DTLS record layer, the retransmission is a new record. This record will have a new DTLSPlaintext.sequence_number value.
DTLS implementations maintain (at least notionally) a next_receive_seq counter. This counter is initially set to zero. When a handshake message is received, if its message_seq value matches next_receive_seq, next_receive_seq is incremented and the message is processed. If the sequence number is less than next_receive_seq, the message MUST be discarded. If the sequence number is greater than next_receive_seq, the implementation SHOULD queue the message but MAY discard it. (This is a simple space/bandwidth tradeoff).

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

5.3.  ClientHello Message

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

```c
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 version fields for DTLS 1.0 and
DTLS 1.2 are 0xfeff and 0xfefd (to match the wire versions) but the version field for DTLS 1.3 is 0x0304.

random: Same as for TLS 1.3.

legacy_session_id: Same as for TLS 1.3.

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

legacy_compression_methods: Same as for TLS 1.3.

extensions: Same as for TLS 1.3.

5.4. Handshake Message Fragmentation and Reassembly

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

When transmitting the handshake message, the sender divides the message into a series of N contiguous data ranges. 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 range MUST be delivered in a single packet.

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

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

The DTLS 1.3 handshake has one important difference from the TLS 1.3 handshake: the EndOfEarlyData message is omitted both from the wire and the handshake transcript: because DTLS records have epochs, EndOfEarlyData is not necessary to determine when the early data is complete, and because DTLS is lossy, attackers can trivially mount the deletion attacks that EndOfEarlyData prevents in TLS. Servers SHOULD aggressively age out the epoch 1 keys upon receiving the first epoch 2 record and SHOULD NOT accept epoch 1 data after the first epoch 3 record is received. (See Section 6.1 for the definitions of each epoch.)

5.6. DTLS Handshake Flights

DTLS messages are grouped into a series of message flights, according to the diagrams below.
Figure 6: Message flights for a full DTLS Handshake (with cookie exchange)
Figure 7: Message flights for resumption and PSK handshake (without cookie exchange)
Client Hello
+ early_data
+ psk_key_exchange_modes
+ key_share*
+ pre_shared_key
(Application Data*) -------->

Server Hello
+ pre_shared_key
+ key_share*
(EncryptedExtensions) | Flight 2 |
{Finished} +----------+
--------------

(Finished) -------->
{Application Data*}

[Application Data*]  +----------+

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

[Application Data]  <-------- [Application Data]

Figure 8: Message flights for the Zero-RTT handshake

Client

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

[ACK]  -------->

[Application Data*]  +----------+

Figure 9: Message flights for the new session ticket message

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

5.7.1. State Machine

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

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

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

There are four ways to exit the WAITING state:

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

2. The implementation reads 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, resets the retransmit timer, and returns to the WAITING state. Upon receiving an ACK for a complete flight, the implementation cancels all retransmissions and either remains in WAITING, or, if the ACK was for the final flight, transitions to FINISHED.

3. The implementation reads a retransmitted flight from the peer: the implementation transitions to the SENDING state, where it retransmits the flight, resets the retransmit timer, and returns to the WAITING state. The rationale here is that the receipt of
a duplicate message is the likely result of timer expiry on the peer and therefore suggests that part of one's previous flight was lost.

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

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

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

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

5.7.2. Timer Values

Though timer values are the choice of the implementation, mishandling of the timer can lead to serious congestion problems; for example, if many instances of a DTLS time out early and retransmit too quickly on a congested link. Implementations SHOULD use an initial timer value of 100 msec (the minimum defined in RFC 6298 [RFC6298]) and double the value at each retransmission, up to no less than the RFC 6298 maximum of 60 seconds. Application specific profiles, such as those used for the Internet of Things environment, may recommend longer timer values. Note that a 100 msec timer is recommended rather than the 3-second RFC 6298 default in order to improve latency for time-sensitive applications. Because DTLS only uses retransmission for handshake and not dataflow, the effect on congestion should be minimal.
Implementations SHOULD retain the current timer value until a transmission without loss occurs, at which time the value may be reset to the initial value. After a long period of idleness, no less than 10 times the current timer value, implementations may reset the timer to the initial value.

5.8. CertificateVerify and Finished Messages

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

5.9. 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".

5.10. Alert Messages

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

5.11. Establishing New Associations with Existing Parameters

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

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

6. Example of Handshake with Timeout and Retransmission

The following is an example of a handshake with lost packets and retransmissions.
Figure 11: Example DTLS exchange illustrating message loss
6.1. Epoch Values and Rekeying

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

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

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

- epoch value (1) is used for messages protected using keys derived from client_early_traffic_secret. 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 payload with an epoch value for which no corresponding cipher state can be determined MUST generate a "unexpected_message" alert. For example, client incorrectly uses epoch value 5 when sending early application data in a 0-RTT exchange. A server will not be able to compute the appropriate keys and will therefore have to respond with an alert.
Note that epoch values do not wrap. If a DTLS implementation would need to wrap the epoch value, it MUST terminate the connection.

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

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

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

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

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

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

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

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

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

[Application Data]  (epoch=3)

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

Some time later ...
```
7. ACK Message

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

```c
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, in numerically increasing order. ACKs only cover the current outstanding flight (this is possible because DTLS is generally a lockstep protocol). Thus, an ACK from the server would not cover both the ClientHello and the client’s Certificate. Implementations can accomplish this by clearing their ACK list upon receiving the start of the next flight.

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

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

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

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

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

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

Note that in some cases it may be necessary to send an ACK which does not contain any record numbers. For instance, a client might receive an EncryptedExtensions message prior to receiving a ServerHello. Because it cannot decrypt the EncryptedExtensions, it cannot safely 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 for only some messages from a flight, an implementation SHOULD retransmit the remaining messages or fragments. Note that this requires implementations to track which messages appear in which records. Once all the messages in a flight have been acknowledged, the implementation MUST cancel all retransmissions of that flight. As noted above, the receipt of any packet responding to a given flight MUST be taken as an implicit acknowledgement for the entire flight.

8. Key Updates

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

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

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

9. Connection ID Updates

If the client and server have negotiated the "connection_id" extension [DTLS-CID], either side can send a new 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. Endpoints MUST NOT have more than one NewConnectionId message outstanding.

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

struct {
    uint8 num_cids;
} RequestConnectionId;

num_cids The number of CIDs desired.

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

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 [DTLS-CID], which is used in ClientHello and ServerHello messages.
Figure 13: 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 fragmented and encrypted based on the current connection state. The messages are treated as transparent data to the record layer.

11. Security Considerations

Security issues are discussed primarily in [TLS13].

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

DTLS implementations MUST NOT update their sending address 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 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.

If implementations process out-of-epoch records as recommended in Section 8, then this creates a denial of service risk since an adversary could inject packets with fake epoch values, forcing the recipient to compute the next-generation application_traffic_secret using the HKDF-Expand-Label construct to only find out that the message was does not pass the AEAD cipher processing. The impact of this attack is small since the HKDF-Expand-Label only performs symmetric key hashing operations. Implementations which are concerned about this form of attack can discard out-of-epoch records.
The security and privacy properties of the CID for DTLS 1.3 builds on top of what is described in [DTLS-CID]. There are, however, several improvements:

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

- With multi-homing, an adversary is able to correlate the communication interaction over the two paths, which adds further privacy concerns. In order to prevent this, implementations SHOULD attempt to use fresh 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.

- Switching CID based on certain events, or even regularly, helps against tracking by on-path adversaries but the sequence numbers can still allow linkability. For this reason this specification defines an algorithm for encrypting sequence numbers, see Section 4.2.3. Note that sequence number encryption is used for all encrypted DTLS 1.3 records irrespectively of the use of a CID.

- 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 to DTLS 1.2

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

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

- 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. IANA Considerations

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

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

14. References

14.1. Normative References


14.2. Informative References

[DTLS-CID]

[I-D.ietf-tls-dtls-connection-id]

[RFC2522]

[RFC4303]

[RFC4340]


14.3. URIs

[1] mailto:tls@ietf.org


Appendix A. Protocol Data Structures and Constant Values

This section provides the normative protocol types and constants definitions.

A.1. Record Layer

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

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

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

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

A.2. Handshake Protocol

```c
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),
enrypted_extensions(8),
certificate(11),
server_key_exchange_RESERVED(12),
certificate_request(13),
server_hello_done_RESERVED(14),
certificate_verify(15),
client_key_exchange_RESERVED(16),
finished(20),
key_update(24),
message_hash(254),
(255)
} HandshakeType;

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

uint16 ProtocolVersion;
opaque Random[32];

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

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

A.3. ACKs

struct {
    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. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

IETF Drafts

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 packets and also be a record rather than a handshake message.
draft-00 - First IETF Draft

Personal Drafts
draft-01 - Alignment with version -19 of the TLS 1.3 specification
draft-00
- Initial version using TLS 1.3 as a baseline.
- Use of epoch values instead of KeyUpdate message
- Use of cookie extension instead of cookie field in ClientHello and HelloVerifyRequest messages
- Added ACK message
- Text about sequence number handling

Appendix C. Working Group Information

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

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

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

In addition, we would like to thank:

* David Benjamin
  Google
davidben@google.com

* Thomas Fossati
  Nokia
  thomas.fossati@nokia.com

* Tobias Gondrom
  Huawei
tobias.gondrom@gondrom.org

* 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

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
Abstract

This document defines a simple mechanism for encrypting the Server Name Indication for TLS 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 May 7, 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
2. Conventions and Definitions ................................. 4
3. Overview ...................................................... 4
   3.1. Topologies .................................................. 4
   3.2. SNI Encryption .............................................. 5
4. Encrypted SNI Configuration ................................. 5
5. The "encrypted_server_name" extension ....................... 7
   5.1. Client Behavior ............................................ 8
      5.1.1. Sending an encrypted SNI ............................. 8
      5.1.2. Handling the server response ......................... 11
      5.1.3. Authenticating for the public name .................. 13
      5.1.4. GREASE extensions .................................. 13
   5.2. Client-Facing Server Behavior .......................... 14
   5.3. Shared Mode Server Behavior ............................ 16
   5.4. Split Mode Server Behavior .............................. 16
6. Compatibility Issues .......................................... 17
   6.1. Misconfiguration and Deployment Concerns ............... 17
   6.2. Middleboxes .............................................. 18
7. Security Considerations ....................................... 18
   7.1. Why is cleartext DNS OK? ................................. 18
   7.2. Optional Record Digests and Trial Decryption .......... 19
   7.3. Encrypting other Extensions .............................. 19
   7.4. Related Privacy Leaks .................................... 19
   7.5. Comparison Against Criteria ................................ 19
      7.5.1. Mitigate against replay attacks .................... 20
      7.5.2. Avoid widely-deployed shared secrets ............... 20
      7.5.3. Prevent SNI-based DoS attacks ....................... 20
      7.5.4. Do not stick out .................................... 20
      7.5.5. Forward secrecy ...................................... 20
      7.5.6. Proper security context ............................. 21
      7.5.7. Split server spoofing ................................ 21
      7.5.8. Supporting multiple protocols ....................... 21
   7.6. Misrouting ................................................. 21
8. IANA Considerations ........................................... 21
   8.1. Update of the TLS ExtensionType Registry ............... 21
   8.2. Update of the TLS Alert Registry ....................... 22
   8.3. Update of the Resource Record (RR) TYPES Registry .... 22
9. References ..................................................... 22
   9.1. Normative References ...................................... 22
   9.2. Informative References .................................... 23
Appendix A. Communicating SNI and Nonce to Backend Server .... 24
Appendix B. Alternative SNI Protection Designs .................. 24
   B.1. TLS-layer .................................................. 24
1. Introduction

DISCLAIMER: This is very early a work-in-progress design and has not yet seen significant (or really any) security analysis. It should not be used as a basis for building production systems.

Although TLS 1.3 [RFC8446] encrypts most of the handshake, including the server certificate, there are several other channels that allow an on-path attacker to determine the domain name the client is trying to connect to, including:

- Cleartext client DNS queries.
- Visible server IP addresses, assuming the server is not doing domain-based virtual hosting.
- Cleartext Server Name Indication (SNI) [RFC6066] in ClientHello messages.

DoH [I-D.ietf-doh-dns-over-https] 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. In such environments, SNI is an explicit signal used to determine the server’s identity. Indirect mechanisms such as traffic analysis also exist.

The TLS WG has extensively studied the problem of protecting SNI, but has been unable to develop a completely generic solution. [I-D.ietf-tls-sni-encryption] provides a description of the problem space and some of the proposed techniques. One of the more difficult problems is "Do not stick out" ([I-D.ietf-tls-sni-encryption]; Section 3.4): if only sensitive/private services use SNI encryption, then SNI encryption is a signal that a client is going to such a service. For this reason, much recent work has focused on concealing the fact that SNI is being protected. Unfortunately, the result often has undesirable performance consequences, incomplete coverage, or both.

The design in this document takes a different approach: it assumes that private origins will co-locate with or hide behind a provider
(CDN, app server, etc.) which is able to activate encrypted SNI (ESNI) for all of the domains it hosts. Thus, the use of encrypted SNI does not indicate that the client is attempting to reach a private origin, but only that it is going to a particular service provider, which the observer could already tell from the IP address.

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 document is designed to operate in one of two primary topologies shown below, which we call "Shared Mode" and "Split Mode"

3.1. Topologies

```
+---------------------+ 2001:DB8::1111
|                     |
|   Client <-----     |
|         private.example.org |
|         public.example.com |
+---------------------+  Server
```

Figure 1: Shared Mode Topology

In Shared Mode, the provider is the origin server for all the domains whose DNS records point to it and clients form a TLS connection directly to that provider, which has access to the plaintext of the connection.
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 backend server, which is the true origin server. The provider does not have access to the plaintext of the connection.

3.2. SNI Encryption

SNI encryption requires that each provider publish a public key and metadata which is used for SNI encryption for all the domains for which it serves directly or indirectly (via Split Mode). This document defines the format of the SNI encryption public key and metadata, referred to as an ESNI configuration, and delegates DNS publication details to [HTTPSSVC], though other delivery mechanisms are possible. In particular, if some of the clients of a private server are applications rather than Web browsers, those applications might have the public key and metadata preconfigured.

When a client wants to form a TLS connection to any of the domains served by an ESNI-supporting provider, it sends an "encrypted_server_name" extension, which contains the true extension encrypted under the provider’s public key. The provider can then decrypt the extension and either terminate the connection (in Shared Mode) or forward it to the backend server (in Split Mode).

4. Encrypted SNI Configuration

SNI Encryption configuration information is conveyed with the following ESNIConfig structure.
// Copied from TLS 1.3
struct {
    NamedGroup group;
    opaque key_exchange<1..2^16-1>;
} KeyShareEntry;

struct {
    uint16 version;
    opaque public_name<1..2^16-1>;
    KeyShareEntry keys<4..2^16-1>;
    CipherSuite cipher_suites<2..2^16-2>;
    uint16 padded_length;
    Extension extensions<0..2^16-1>;
} ESNIConfig;

The ESNIConfig structure contains the following fields:

version  The version of the structure. For this specification, that value SHALL be 0xff03. Clients MUST ignore any ESNIConfig structure with a version they do not understand. [[NOTE: This means that the RFC will presumably have a nonzero value.]]

public_name  The non-empty name of the entity trusted to update these encryption keys. This is used to repair misconfigurations, as described in Section 5.1.2.

keys  The list of keys which can be used by the client to encrypt the SNI. Every key being listed MUST belong to a different group.

padded_length The length to pad the ServerNameList value to prior to encryption. This value SHOULD be set to the largest ServerNameList the server expects to support rounded up the nearest multiple of 16. If the server supports arbitrary wildcard names, it SHOULD set this value to 260. Clients SHOULD reject ESNIConfig as invalid if padded_length is greater than 260.

extensions A list of extensions that the client can take into consideration when generating a Client Hello message. The format is defined in [RFC8446]; Section 4.2. The purpose of the field is to provide room for additional features in the future. An extension may be tagged as mandatory by using an extension type codepoint with the high order bit set to 1. A client which receives a mandatory extension they do not understand must reject the ESNIConfig content.

Any of the listed keys in the ESNIConfig value may be used to encrypt the SNI for the associated domain name. The cipher suite list is orthogonal to the list of keys, so each key may be used with any
cipher suite. Clients MUST parse the extension list and check for unsupported mandatory extensions. If an unsupported mandatory extension is present, clients MUST reject the ESNIConfig value.

5. The "encrypted_server_name" extension

The encrypted SNI is carried in an "encrypted_server_name" extension, defined as follows:

```c
enum {
    encrypted_server_name(0xffce), (65535)
} ExtensionType;
```

For clients (in ClientHello), this extension contains the following ClientEncryptedSNI structure:

```c
struct {
    CipherSuite suite;
    KeyShareEntry key_share;
    opaque record_digest<0..2^16-1>;
    opaque encrypted_sni<0..2^16-1>;
} ClientEncryptedSNI;
```

- **suite** The cipher suite used to encrypt the SNI.
- **key_share** The KeyShareEntry carrying the client’s public ephemeral key share used to derive the ESNI key.
- **record_digest** A cryptographic hash of the ESNIConfig structure from which the ESNI key was obtained, i.e., from the first byte of "version" to the end of the structure. This hash is computed using the hash function associated with "suite".
- **encrypted_sni** The ClientESNIInner structure, AEAD-encrypted using cipher suite "suite" and the key generated as described below.

For servers (in EncryptedExtensions), this extension contains the following structure:
enum {
    esni_accept(0),
    esni_retry_request(1),
} ServerESNIResponseType;

struct {
    ServerESNIResponseType response_type;
    select (response_type) {
        case esni_accept:        uint8 nonce[16];
        case esni_retry_request: ESNIConfig retry_keys<1..2^16-1>;
    }
} ServerEncryptedSNI;

response_type  Indicates whether the server processed the client ESNI extension. (See Section 5.1.2 and Section 5.2.)
nonce  The contents of ClientESNIInner.nonce. (See Section 5.1.)
retry_keys  One or more ESNIConfig structures containing the keys that the client should use on subsequent connections to encrypt the ClientESNIInner structure.

This protocol also defines the "esni_required" alert, which is sent by the client when it offered an "encrypted_server_name" extension which was not accepted by the server.

enum {
    esni_required(121),
} AlertDescription;

Finally, requirements in Section 5.1 and Section 5.2 require implementations to track, alongside each PSK established by a previous connection, whether the connection negotiated this extension with the "esni_accept" response type. If so, this is referred to as an "ESNI PSK". Otherwise, it is a "non-ESNI PSK". This may be implemented by adding a new field to client and server session states.

5.1.  Client Behavior

5.1.1.  Sending an encrypted SNI

In order to send an encrypted SNI, the client MUST first select one of the server ESNIKeyShareEntry values and generate an (EC)DHE share in the matching group. This share will then be sent to the server in the "encrypted_server_name" extension and used to derive the SNI encryption key. It does not affect the (EC)DHE shared secret used in the TLS key schedule. The client MUST also select an appropriate
cipher suite from the list of suites offered by the server. If the
client is unable to select an appropriate group or suite it SHOULD
ignore that ESNIConfig value and MAY attempt to use another value
provided by the server. The client MUST NOT send encrypted SNI using
groups or cipher suites not advertised by the server.

When offering an encrypted SNI, the client MUST NOT offer to resume
any non-ESNI PSKs. It additionally MUST NOT offer to resume any
sessions for TLS 1.2 or below.

Let \( Z \) be the DH shared secret derived from a key share in ESNIConfig
and the corresponding client share in ClientEncryptedSNI.key_share.
The SNI encryption key is computed from \( Z \) as follows:

\[
Zx = \text{HKDF-Extract}(0, Z)
\]

\[
\text{key} = \text{HKDF-Expand-Label}(Zx, \text{KeyLabel}, \text{Hash}(\text{ESNIContents}), \text{key_length})
\]

\[
\text{iv} = \text{HKDF-Expand-Label}(Zx, \text{IVLabel}, \text{Hash}(\text{ESNIContents}), \text{iv_length})
\]

where \( \text{ESNIContents} \) is as specified below and \( \text{Hash} \) is the hash
function associated with the HKDF instantiation. The salt argument
for HKDF-Extract is a string consisting of \( \text{Hash}.\text{length} \) bytes set to
zeros. For a client’s first ClientHello, KeyLabel = "esni key" and
IVLabel = "esni iv", whereas for a client’s second ClientHello, sent
in response to a HelloRetryRequest, KeyLabel = "hrr esni key" and
IVLabel = "hrr esni iv". (This label variance is done to prevent
nonce re-use since the client’s ESNI key share, and thus the value of
\( Zx \), does not change across ClientHello retries.)

Note that \( \text{ESNIContents} \) will not be directly transmitted to the server
in the ClientHello. The server will instead reconstruct the same
object by obtaining its values from ClientEncryptedSNI and
ClientHello.

[[TODO: label swapping fixes a bug in the spec, though this may not
be the best way to deal with HRR. See https://github.com/tlswg/
draft-ietf-tls-esni/issues/121 and https://github.com/tlswg/draft-
ietf-tls-esni/pull/170 for more details.]]

```
struct {
    opaque record_digest<0..2^16-1>;
    KeyShareEntry esni_key_share;
    Random client_hello_random;
} ESNIContents;
```

record_digest  Same value as ClientEncryptedSNI.record_digest.
esni_key_share  Same value as ClientEncryptedSNI.key_share.
The client then creates a ClientESNIInner structure:

```c
struct {
    opaque dns_name<1..2^16-1>;
    opaque zeros[ESNIConfig.padded_length - length(dns_name)];
} PaddedServerNameList;

struct {
    uint8 nonce[16];
    PaddedServerNameList realSNI;
} ClientESNIInner;
```

- **nonce**: A random 16-octet value to be echoed by the server in the "encrypted_server_name" extension.
- **dns_name**: The true SNI DNS name, that is, the HostName value that would have been sent in the plaintext "server_name" extension. (NameType values other than "host_name" are unsupported since SNI extensibility failed [SNIExtensibilityFailed]).
- **zeros**: Zero padding whose length makes the serialized PaddedServerNameList struct have a length equal to ESNIConfig.padded_length.

This value consists of the serialized ServerNameList from the "server_name" extension, padded with enough zeroes to make the total structure ESNIConfig.padded_length bytes long. The purpose of the padding is to prevent attackers from using the length of the "encrypted_server_name" extension to determine the true SNI. If the serialized ServerNameList is longer than ESNIConfig.padded_length, the client MUST NOT use the "encrypted_server_name" extension.

The ClientEncryptedSNI.encrypted_sni value is then computed using AEAD-Encrypt ([RFC5116]; Section 2.1) with the AEAD corresponding to ClientEncryptedSNI.suite as follows:

```
encrypted_sni = AEAD-Encrypt(key, iv, KeyShareClientHello, ClientESNIInner)
```

Where KeyShareClientHello is the "extension_data" field of the "key_share" extension in a Client Hello (Section 4.2.8 of [RFC8446])). Including KeyShareClientHello in the AAD of AEAD-Encrypt binds the ClientEncryptedSNI value to the ClientHello and prevents cut-and-paste attacks.

Note: future extensions may end up reusing the server’s ESNIKeyShareEntry for other purposes within the same message (e.g.,
encrypting other values). Those usages MUST have their own HKDF labels to avoid reuse.

[[OPEN ISSUE: If in the future you were to reuse these keys for 0-RTT priming, then you would have to worry about potentially expanding twice of $Z_{extracted}$. We should think about how to harmonize these to make sure that we maintain key separation.]]

This value is placed in an "encrypted_server_name" extension.

The client MUST place the value of ESNIConfig.public_name in the "server_name" extension. (This is required for technical conformance with [RFC7540]; Section 9.2.) The client MUST NOT send a "cached_info" extension [RFC7924] with a CachedObject entry whose CachedInformationType is "cert", since this indication would divulge the true server name.

5.1.2. Handling the server response

If the server negotiates TLS 1.3 or above and provides an "encrypted_server_name" extension in EncryptedExtensions, the client then processes the extension’s "response_type" field:

- If the value is "esni_accept", the client MUST check that the extension’s "nonce" field matches ClientESNIInner.nonce and otherwise abort the connection with an "illegal_parameter" alert. The client then proceeds with the connection as usual, authenticating the connection for the origin server.

- If the value is "esni_retry_request", the client proceeds with the handshake, authenticating for ESNIConfig.public_name as described in Section 5.1.3. If authentication or the handshake fails, the client MUST return a failure to the calling application. It MUST NOT use the retry keys.

Otherwise, when the handshake completes successfully with the public name authenticated, the client MUST abort the connection with an "esni_required" alert. It then processes the "retry_keys" field from the server’s "encrypted_server_name" extension.

If one of the values contains a version supported by the client, it can regard the ESNI keys as securely replaced by the server. It SHOULD retry the handshake with a new transport connection, using that value to encrypt the SNI. The value may only be applied to the retry connection. The client MUST continue to use the previously-advertised keys for subsequent connections. This avoids introducing pinning concerns or a tracking vector, should a
malicious server present client-specific retry keys to identify clients.

If none of the values provided in "retry_keys" contains a supported version, the client can regard ESNI as securely disabled by the server. As below, it SHOULD then retry the handshake with a new transport connection and ESNI disabled.

- If the field contains any other value, the client MUST abort the connection with an "illegal_parameter" alert.

If the server negotiates an earlier version of TLS, or if it does not provide an "encrypted_server_name" extension in EncryptedExtensions, the client proceeds with the handshake, authenticating for ESNICfg.public_name as described in Section 5.1.3. If an earlier version was negotiated, the client MUST NOT enable the False Start optimization [RFC7918] for this handshake. If authentication or the handshake fails, the client MUST return a failure to the calling application. It MUST NOT treat this as a secure signal to disable ESNI.

Otherwise, when the handshake completes successfully with the public name authenticated, the client MUST abort the connection with an "esni_required" alert. The client can then regard ESNI as securely disabled by the server. It SHOULD retry the handshake with a new transport connection and ESNI disabled.

[[TODO: Key replacement is significantly less scary than saying that ESNI-naive servers bounce ESNI off. Is it worth defining a strict mode toggle in the ESNI keys, for a deployment to indicate it is ready for that? ]]

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

If the server sends a HelloRetryRequest in response to the ClientHello and the client can send a second updated ClientHello per the rules in [RFC8446], the "encrypted_server_name" extension values which do not depend on the (possibly updated) KeyShareClientHello, i.e., ClientEncryptedSNI.suite, ClientEncryptedSNI.key_share, and ClientEncryptedSNI.record_digest, MUST NOT change across ClientHello messages. Moreover, ClientESNIInner MUST not change across ClientHello messages. Informally, the values of all unencrypted extension information, as well as the inner extension plaintext, must be consistent between the first and second ClientHello messages.
5.1.3. Authenticating for the public name

When the server cannot decrypt or does not process the "encrypted_server_name" extension, it continues with the handshake using the cleartext "server_name" extension instead (see Section 5.2). Clients that offer ESNI then authenticate the connection with the public name, as follows:

- If the server resumed a session or negotiated a session that did not use a certificate for authentication, the client MUST abort the connection with an "illegal_parameter" alert. This case is invalid because Section 5.1.1 requires the client to only offer ESNI-established sessions, and Section 5.2 requires the server to decline ESNI-established sessions if it did not accept ESNI.

- The client MUST verify that the certificate is valid for ESNIConfig.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.

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 5.1.2. This may be implemented, for instance, by reporting a failed connection with a dedicated error code.

5.1.4. GREASE extensions

If the client attempts to connect to a server and does not have an ESNIConfig structure available for the server, it SHOULD send a GREASE [I-D.ietf-tls-grease] "encrypted_server_name" extension as follows:

- Select a supported cipher suite, named group, and padded_length value. The padded_length value SHOULD be 260 (sum of the maximum DNS name length and TLS encoding overhead) or a multiple of 16 less than 260. Set the "suite" field to the selected cipher suite. These selections SHOULD vary to exercise all supported configurations, but MAY be held constant for successive connections to the same server in the same session.
o Set the "key_share" field to a randomly-generated valid public key for the named group.

o Set the "record_digest" field to a randomly-generated string of hash_length bytes, where hash_length is the length of the hash function associated with the chosen cipher suite.

o Set the "encrypted_sni" field to a randomly-generated string of 16 + padded_length + tag_length bytes, where tag_length is the tag length of the chosen cipher suite’s associated AEAD.

If the server sends an "encrypted_server_name" extension, the client MUST check the extension syntactically and abort the connection with a "decode_error" alert if it is invalid. If the "response_type" field contains "esni_retry_requested", the client MUST ignore the extension and proceed with the handshake. If it contains "esni_accept" or any other value, the client MUST abort the connection with an "illegal_parameter" alert.

Offering a GREASE extension is not considered offering an encrypted SNI for purposes of requirements in Section 5.1. In particular, the client MAY offer to resume sessions established without ESNI.

5.2. Client-Facing Server Behavior

Upon receiving an "encrypted_server_name" extension, the client-facing server MUST check that it is able to negotiate TLS 1.3 or greater. If not, it MUST abort the connection with a "handshake_failure" alert.

The ClientEncryptedSNI value is said to match a known ESNIConfig if there exists an ESNIConfig that can be used to successfully decrypt ClientEncryptedSNI.encrypted_sni. This matching procedure should be done using one of the following two checks:

1. Compare ClientEncryptedSNI.record_digest against cryptographic hashes of known ESNIConfig and choose the one that matches.

2. Use trial decryption of ClientEncryptedSNI.encrypted_sni with known ESNIConfig and choose the one that succeeds.

Some uses of ESNI, such as local discovery mode, may omit the ClientEncryptedSNI.record_digest since it can be used as a tracking vector. In such cases, trial decryption should be used for matching ClientEncryptedSNI to known ESNIConfig. Unless specified by the application using (D)TLS or externally configured on both sides, implementations MUST use the first method.
If the ClientEncryptedSNI value does not match any known ESNIConfig structure, it MUST ignore the extension and proceed with the connection, with the following added behavior:

- It MUST include the "encrypted_server_name" extension in EncryptedExtensions message with the "response_type" field set to "esni_retry_requested" and the "retry_keys" field set to one or more ESNIConfig structures with up-to-date keys. Servers MAY supply multiple ESNIConfig values of different versions. This allows a server to support multiple versions at once.

- The server MUST ignore all PSK identities in the ClientHello which correspond to ESNI PSKs. ESNI PSKs offered by the client are associated with the ESNI name. The server was unable to decrypt then ESNI name, so it should not resume them when using the cleartext SNI name. This restriction allows a client to reject resumptions in Section 5.1.3.

Note that an unrecognized ClientEncryptedSNI.record_digest value may be a GREASE ESNI extension (see Section 5.1.4), so it is necessary for servers to proceed with the connection and rely on the client to abort if ESNI was required. In particular, the unrecognized value alone does not indicate a misconfigured ESNI advertisement (Section 6.1). Instead, servers can measure occurrences of the "esni_required" alert to detect this case.

If the ClientEncryptedSNI value does match a known ESNIConfig, the server performs the following checks:

- If the ClientEncryptedSNI.key_share group does not match one in the ESNIConfig.keys, it MUST abort the connection with an "illegal_parameter" alert.

- If the length of the "encrypted_server_name" extension is inconsistent with the advertised padding length (plus AEAD expansion) the server MAY abort the connection with an "illegal_parameter" alert without attempting to decrypt.

Assuming these checks succeed, the server then computes $K_{sni}$ and decrypts the ServerName value. If decryption fails, the server MUST abort the connection with a "decrypt_error" alert.

If the decrypted value’s length is different from the advertised ESNIConfig.padded_length or the padding consists of any value other than 0, then the server MUST abort the connection with an "illegal_parameter" alert. Otherwise, the server uses the PaddedServerNameList.sni value as if it were the "server_name" extension. Any actual "server_name" extension is ignored, which also
means the server MUST NOT send the "server_name" extension to the client.

Upon determining the true SNI, the client-facing server then either serves the connection directly (if in Shared Mode), in which case it executes the steps in the following section, or forwards the TLS connection to the backend server (if in Split Mode). In the latter case, it does not make any changes to the TLS messages, but just blindly forwards them.

If the ClientHello is the result of a HelloRetryRequest, servers MUST abort the connection with an "illegal_parameter" alert if any of the ClientEncryptedSNI.suite, ClientEncryptedSNI.key_share, ClientEncryptedSNI.record_digest, or decrypted ClientESNIInner values from the second ClientHello do not match that of the first ClientHello.

5.3. Shared Mode Server Behavior

A server operating in Shared Mode uses PaddedServerNameList.sni as if it were the "server_name" extension to finish the handshake. It SHOULD pad the Certificate message, via padding at the record layer, such that its length equals the size of the largest possible Certificate (message) covered by the same ESNI key. Moreover, the server MUST include the "encrypted_server_name" extension in EncryptedExtensions with the "response_type" field set to "esni_accept" and the "nonce" field set to the decrypted PaddedServerNameList.nonce value from the client "encrypted_server_name" extension.

If the server sends a NewSessionTicket message, the corresponding ESNI PSK MUST be ignored by all other servers in the deployment when not negotiating ESNI, including servers which do not implement this specification.

This restriction provides robustness for rollbacks (see Section 6.1).

5.4. Split Mode Server Behavior

In Split Mode, the backend server must know PaddedServerNameList.nonce to echo it back in EncryptedExtensions and complete the handshake. Appendix A describes one mechanism for sending both PaddedServerNameList.sni and ClientESNIInner.nonce to the backend server. Thus, backend servers function the same as servers operating in Shared Mode.

As in Shared Mode, if the backend server sends a NewSessionTicket message, the corresponding ESNI PSK MUST be ignored by other servers
in the deployment when not negotiating ESNI, including servers which do not implement this specification.

6. Compatibility Issues

Unlike most TLS extensions, placing the SNI value in an ESNI extension is not interoperable with existing servers, which expect the value in the existing cleartext extension. Thus server operators SHOULD ensure servers understand a given set of ESNI 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 ESNI keys and servers, at the cost of extra round-trips due to a retry. Two specific scenarios are detailed below.

6.1. Misconfiguration and Deployment Concerns

It is possible for ESNI 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 ESNI keys, or if a deployment of ESNI 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 respond with esni_retry_requested. If the server does not understand the "encrypted_server_name" extension at all, it will ignore it as required by [RFC8446]; Section 4.1.2. Provided the server can present a certificate valid for the public name, the client can safely retry with updated settings, as described in Section 5.1.2.

Unless ESNI is disabled as a result of successfully establishing a connection to the public name, the client MUST NOT fall back to cleartext SNI, as this allows a network attacker to disclose the SNI. It MAY attempt to use another server from the DNS results, if one is provided.

Client-facing servers with non-uniform cryptographic configurations across backend origin servers segment the ESNI anonymity set based on these configurations. For example, if a client-facing server hosts k backend origin servers, and exactly one of those backend origin servers supports a different set of cryptographic algorithms than the other (k - 1) servers, it may be possible to identify this single
server based on the contents of the ServerHello as this message is not encrypted.

6.2. Middleboxes

A more serious problem is MITM proxies which do not support this extension. [RFC8446]; Section 9.3 requires that such proxies remove any extensions they do not understand. The handshake will then present a certificate based on the public name, without echoing the "encrypted_server_name" extension to the client.

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 5.1.2 or result in a connection failure. A proxy which is not authoritative for the public name cannot forge a signal to disable ESNI.

A non-conformant MITM proxy which instead forwards the ESNI extension, substituting its own KeyShare value, will result in the client-facing server recognizing the key, but failing to decrypt the SNI. This causes a hard failure. Clients SHOULD NOT attempt to repair the connection in this case.

7. Security Considerations

7.1. Why is cleartext DNS OK?

In comparison to [I-D.kazuho-protected-sni], wherein DNS Resource Records are signed via a server private key, ESNI 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 ESNI records (so that the client encrypts SNI to them) or strip the ESNI record from the response. However, in the face of an attacker that controls DNS, no SNI encryption scheme can work because the attacker can replace the IP address, thus blocking client connections, or substituting a unique IP address which is 1:1 with the DNS name that was looked up (modulo DNS wildcards). Thus, allowing the ESNI 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 SNI is desired. Moreover, as noted in the introduction, SNI encryption is less useful without encryption of DNS queries in transit via DoH or DPRIVE mechanisms.
7.2. Optional Record_DIGESTS and Trial Decryption

Supporting optional record digests and trial decryption opens oneself up to DoS attacks. Specifically, an adversary may send malicious ClientHello messages, i.e., those which will not decrypt with any known ESNI key, in order to force decryption. Servers that support this feature should, for example, implement some form of rate limiting mechanism to limit the damage caused by such attacks.

7.3. Encrypting Other Extensions

ESNI protects only the SNI in transit. Other ClientHello extensions, such as ALPN, might also reveal privacy-sensitive information to the network. As such, it might be desirable to encrypt other extensions alongside the SNI. However, the SNI extension is unique in that non-TLS-terminating servers or load balancers may act on its contents. Thus, using keys specifically for SNI encryption promotes key separation between client-facing servers and endpoints party to TLS connections. Moreover, the ESNI design described herein does not preclude a mechanism for generic ClientHello extension encryption.

7.4. Related Privacy Leaks

ESNI 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.

7.5. Comparison Against Criteria

[I-D.ietf-tls-sni-encryption] lists several requirements for SNI encryption. In this section, we re-iterate these requirements and assess the ESNI design against them.
7.5.1. Mitigate against replay attacks

Since the SNI encryption key is derived from a (EC)DH operation between the client’s ephemeral and server’s semi-static ESNI key, the ESNI encryption is bound to the Client Hello. It is not possible for an attacker to "cut and paste" the ESNI value in a different Client Hello, with a different ephemeral key share, as the terminating server will fail to decrypt and verify the ESNI value.

7.5.2. Avoid widely-deployed 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 ESNI 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 ESNIConfig values with different keys using a short TTL.

7.5.3. Prevent SNI-based DoS attacks

This design requires servers to decrypt ClientHello messages with ClientEncryptedSNI 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.

7.5.4. Do not stick out

As more clients enable ESNI support, e.g., as normal part of Web browser functionality, with keys supplied by shared hosting providers, the presence of ESNI extensions becomes less suspicious and part of common or predictable client behavior. In other words, if all Web browsers start using ESNI, the presence of this value does not signal suspicious behavior to passive eavesdroppers.

Additionally, this specification allows for clients to send GREASE ESNI extensions (see Section 5.1.4), which helps ensure the ecosystem handles the values correctly.

7.5.5. Forward secrecy

This design is not forward secret because the server’s ESNI key is static. However, the window of exposure is bound by the key lifetime. It is RECOMMENDED that servers rotate keys frequently.
7.5.6. Proper security context

This design permits servers operating in Split Mode to forward connections directly to backend origin servers, thereby avoiding unnecessary MiTM attacks.

7.5.7. Split server spoofing

Assuming ESNI records retrieved from DNS are authenticated, e.g., via DNSSEC or fetched from a trusted Recursive Resolver, spoofing a server operating in Split Mode is not possible. See Section 7.1 for more details regarding cleartext DNS.

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

7.5.8. Supporting 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 protocols.

7.6. Misrouting

Note that the backend server has no way of knowing what the SNI was, but that does not lead to additional privacy exposure because the backend server also only has one identity. This does, however, change the situation slightly in that the backend server might previously have checked SNI and now cannot (and an attacker can route a connection with an encrypted SNI to any backend server and the TLS connection will still complete). However, the client is still responsible for verifying the server’s identity in its certificate.

[[TODO: Some more analysis needed in this case, as it is a little odd, and probably some precise rules about handling ESNI and no SNI uniformly?]]

8. IANA Considerations

8.1. Update of the TLS ExtensionType Registry

IANA is requested to create an entry, encrypted_server_name(0xffce), in the existing registry for ExtensionType (defined in [RFC8446]), with "TLS 1.3" column values being set to "CH, EE", and "Recommended" column being set to "Yes".
8.2. Update of the TLS Alert Registry

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

8.3. Update of the Resource Record (RR) TYPEs Registry

IANA is requested to create an entry, ESNI(0xff9f), in the existing registry for Resource Record (RR) TYPEs (defined in [RFC6895]) with "Meaning" column value being set to "Encrypted SNI".

9. References

9.1. Normative References


9.2. Informative References

[I-D.ietf-doh-dns-over-https]

[I-D.ietf-tls-grease]

[I-D.ietf-tls-sni-encryption]
Appendix A. Communicating SNI and Nonce to Backend Server

When operating in Split Mode, backend servers will not have access to PaddedServerNameList.sni or ClientESNIInner.nonce without access to the ESNI keys or a way to decrypt ClientEncryptedSNI.encrypted_sni.

One way to address this for a single connection, at the cost of having communication not be unmodified TLS 1.3, is as follows. Assume there is a shared (symmetric) key between the client-facing server and the backend server and use it to AEAD-encrypt Z and send the encrypted blob at the beginning of the connection before the ClientHello. The backend server can then decrypt ESNI to recover the true SNI and nonce.

Appendix B. 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.

B.1. TLS-layer

B.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.

B.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.

B.2. Application-layer

B.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 C. Total Client Hello Encryption

The design described here only provides encryption for the SNI, but not for other extensions, such as ALPN. Another potential design would be to encrypt all of the extensions using the same basic structure as we use here for ESNI. That design has the following advantages:

- It protects all the extensions from ordinary eavesdroppers
- If the encrypted block has its own KeyShare, it does not necessarily require the client to use a single KeyShare, because the client’s share is bound to the SNI by the AEAD (analysis needed).

It also has the following disadvantages:

- The client-facing server can still see the other extensions. By contrast we could introduce another EncryptedExtensions block that was encrypted to the backend server and not the client-facing server.
- It requires a mechanism for the client-facing server to provide the extension-encryption key to the backend server (as in Appendix A and thus cannot be used with an unmodified backend server.
- A conformant middlebox will strip every extension, which might result in a ClientHello which is just unacceptable to the server (more analysis needed).

Appendix D. 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 ESNI key. Richard Barnes, Christian Huitema, Patrick McManus, Matthew Prince, Nick Sullivan, Martin Thomson, and David Benjamin also provided important ideas and contributions.

Authors’ Addresses

Eric Rescorla
RTFM, Inc.

Email: ekr@rtfm.com
Abstract

The organizational separation between the operator of a TLS endpoint 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 by which operators may delegate their own credentials for use in TLS, without breaking compatibility with peers that do not support this specification.

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 6, 2020.

Copyright Notice

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

Typically, a TLS server uses a certificate provided by some entity other than the operator of the server (a "Certification Authority" or CA) [RFC8446] [RFC5280]. This organizational separation makes the 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 server operator can only use TLS authentication schemes for which the CA will issue credentials.

These dependencies cause problems in practice. Server operators often want to create short-lived certificates for servers in low-trust zones such as Content Delivery Network (CDNs) or remote data centers. This allows server operators to limit the exposure of keys in cases that they do not realize a compromise has occurred. The risk inherent in cross-organizational transactions makes it operationally infeasible to rely on an external CA for such short-lived credentials. In Online Certificate Status Protocol (OCSP) stapling (i.e., using the Certificate Status extension types ocsp [RFC6066] or ocsp_multi [RFC6961]), if an operator chooses to talk frequently to the CA to obtain stapled responses, then failure to fetch an OCSP stapled response results only in degraded performance. On the other hand, failure to fetch a potentially large number of short lived certificates would result in the service not being available, which creates greater operational risk.

To remove these dependencies, this document proposes a limited delegation mechanism that allows a TLS peer to issue its own credentials within the scope of a certificate issued by an external CA. Because the above problems do not relate to the CA’s inherent function of validating possession of names, it is safe to make such delegations as long as they only enable the recipient of the delegation to speak for names that the CA has authorized. For clarity, we will refer 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".

1.1. Change Log

(*) indicates changes to the wire protocol.

draft-05
  o Removed support for PKCS 1.5 RSA signature algorithms.
  o Additional security considerations.

draft-04
  o Add support for client certificates.

draft-03
  o 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.

2. Solution Overview

A delegated credential 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 credential indicates a delegation from the certificate that is issued to the peer. The secret key used to sign a credential corresponds to the public key of the peer’s X.509 end-entity certificate [RFC5280].

A TLS handshake that uses delegated credentials differs from a normal 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 authenticating intitiator uses information from the peer’s certificate to verify the delegated credential and that the peer is asserting an expected identity.
- Peers accepting the delegated credential use it as the certificate’s working key for the TLS handshak

As detailed in Section 3, 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 TLS 1.3 or later; their use in prior versions of the protocol is not allowed.

Delegated credentials allow a peer to terminate 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 a delegated credential is compromised, peers may not issue credentials with a validity period longer than 7 days. This mechanism is described in detail in Section 3.1.

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 prevent this type of cross-protocol attack, we define a new DelegationUsage extension to X.509 that permits use of delegated credentials. (See Section 3.2.)

2.1. 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. 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 server 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 that may be used to sign the TLS handshake ([RFC8446] section 4.2.3). This prevents them from being used with other, perhaps unintended signature algorithms.
2.2. 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]). 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 TLS 1.3 [RFC8446].

Remote key signing:

```
<table>
<thead>
<tr>
<th>Client</th>
<th>Front-End</th>
<th>Back-End</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;----ClientHello--</td>
<td></td>
<td>&lt;---remote sign----&gt;</td>
</tr>
<tr>
<td>&lt;---ServerHello----</td>
<td></td>
<td>&lt;---Certificate----&lt;---remote sign----&gt;</td>
</tr>
<tr>
<td>&lt;---Certificate----</td>
<td></td>
<td>&lt;---CertVerify-----</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
```

Delegated credentials:

```
<table>
<thead>
<tr>
<th>Client</th>
<th>Front-End</th>
<th>Back-End</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;---ClientHello----</td>
<td>&lt;---DC distribution-&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;---ServerHello----</td>
<td></td>
<td>&lt;---Certificate----</td>
</tr>
<tr>
<td>&lt;---Certificate----</td>
<td></td>
<td>&lt;---CertVerify-----</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
```

These two mechanisms can be complementary. A server could use credentials for clients that support them, while using [KEYLESS] to support legacy clients.

It is possible to address the short-lived certificate concerns above by automating certificate issuance, e.g., with Automated Certificate Management Environment (ACME) [RFC8555]. In addition to requiring frequent operationally-critical interactions with an external party, this makes the server operator dependent on the CA’s willingness to issue certificates with sufficiently short lifetimes. It also fails to address the issues with algorithm support. Nonetheless, existing automated issuance APIs like ACME may be useful for provisioning credentials within an operator network.
3. Delegated Credentials

While X.509 forbids end-entity certificates from being used as issuers for other certificates, it is perfectly fine to use them to issue other signed objects as long as the certificate contains the digitalSignature KeyUsage ([RFC5280] section 4.2.1.3). 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:

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

`valid_time`: Relative time in seconds from the beginning of the delegation certificate’s `notBefore` value after which the delegated credential is no longer valid.

`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 `CertificateVerify.algorithm` sent by the server. 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 [X690] SubjectPublicKeyInfo as defined in [RFC5280].

The delegated credential has the following structure:

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

`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 servers and "TLS, client delegated credentials" for clients.
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 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. Minimizing their semantics in this way is intended to mitigate the risk of cross protocol attacks involving delegated credentials.

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

3.1. Client and Server behavior

This document defines the following TLS extension code point.

    enum {
        ...  
        delegated_credential(TBD),  
        (65535)  
    } ExtensionType;

3.1.1. Server authentication

A client which supports this specification SHALL send an empty "delegated_credential" extension in its ClientHello. If the client receives a delegated credential without indicating support, then the client MUST abort 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
3.1.2. Client authentication

A server which supports this specification SHALL send an empty "delegated_credential" extension in the CertificateRequest message when requesting client authentication. If the server receives a delegated credential without indicating 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 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 and expected_cert_verify_algorithm fields MUST be of a type advertised by the server in the "signature_algorithms" extension and are considered invalid otherwise. Servers that receive invalid delegated credentials MUST terminate the connection with an "illegal_parameter" alert.

3.1.3. Validating a Delegated Credential

On receiving a delegated credential and a certificate chain, the peer validates the certificate chain and matches the end-entity certificate to the peer's expected identity in the usual way. It also takes the following steps:

1. Verify that the current time is within the validity interval of the credential and that the credential’s time to live is no more
than 7 days. This is done by asserting that the current time is no more than the delegation certificate’s notBefore value plus DelegatedCredential.cred.valid_time.

2. 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.

3. Verify that the end-entity certificate satisfies the conditions in Section 3.2.

4. 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 the credential to verify the signature in the peer’s CertificateVerify message.

3.2. Certificate Requirements

We define 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 [X680] for the DelegationUsage certificate extension.

```asn1
ext-delegationUsage EXTENSION ::= {
    SYNTAX DelegationUsage IDENTIFIED BY id-ce-delegationUsage
}
DelegationUsage ::= NULL
id-ce-delegationUsage OBJECT IDENTIFIER ::=  { 1 3 6 1 4 1 44363 44 }
```

The extension MUST be marked non-critical. (See Section 4.2 of [RFC5280].) The client MUST NOT accept a delegated credential unless the server’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]).
4. IANA Considerations

This document registers the "delegated_credentials" extension in the "TLS ExtensionType Values" registry. The "delegated_credentials" extension has been assigned a code point of TBD. 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 TLS 1.3 [RFC8446].

This document also defines an ASN.1 module for the DelegationUsage certificate extension in Appendix A. IANA is requested to register an Object Identifier (OID) for the ASN.1 in "SMI Security for PKIX Module Identifier" arc. 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.

5. Security Considerations

5.1. Security of delegated private key

Delegated credentials limit the exposure of the TLS private key by limiting its validity. An attacker who compromises the private key of a delegated credential can act as a man-in-the-middle until the delegate 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 is meant to be used between parties that have some trust relationship with each other. The secrecy of the delegated private key is thus important and several access control mechanisms SHOULD be used to protect it, including file system controls, physical security, or hardware security modules.

5.2. Re-use of delegated credentials in multiple contexts

It is possible to use the same delegated credential for both client and server authentication if the Certificate allows it. This is safe because the context string used for delegated credentials is distinct in both contexts.

5.3. 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 would revoke the credential. Revocation of the long term private key that signs the delegated credential also implicitly revokes the delegated credential.
5.4. Interactions with session resumption

If a client decides to cache the certificate chain and re-validate it when resuming a connection, the client SHOULD also cache the associated delegated credential and re-validate it.

5.5. Privacy considerations

Delegated credentials can be valid for 7 days 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.

6. Acknowledgements

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

7. References

7.1. Normative References


7.2. Informative References


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 { joint-iso-itu-t(2) country(16) us(840) organization(1) gov(101) dod(2) infosec(1) modules(0) id-mod-delegate-credential-extn(TBD) }

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) } ;

- OIDS

id-cloudflare OBJECT IDENTIFIER ::= { 1 3 6 1 4 1 44363 }

- EXTENSION

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

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

DelegationUsage ::= NULL

END

Authors’ Addresses

Richard Barnes
Cisco

Email: rlb@ipv.sx

Subodh Iyengar
Facebook

Email: subodh@fb.com

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com
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

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


10.2. Informative References


Authors’ Addresses

Loganaden Velvindron
cyberstorm.mu
Rose Hill
MU

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

Kathleen Moriarty
Dell EMC

Alessandro Ghedini
Cloudflare Inc.
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 + the order of the last set bit divided by 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

Nir                     Expires January 23, 2020                [Page 1]
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:

- An extension that allows the server to tell the client that cross-SNI resumption is allowed: [I-D.sy-tls-resumption-group].
- 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:

```java
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.

```c
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:

- The Extension Name should be tls_flags
- The TLS 1.3 value should be CH, SH, EE
- The Recommended value should be Y
- 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:

- Value, which is a number between 0 and 63. All potential values are available for assignment.
- Flag Name, which is a string
- 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.
- Recommended, which is a Y/N value determined in the document defining the optional feature.
- 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


6.2. Informative References


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
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
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:

- 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
- 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.

- 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:

- an extensible data model for describing alternative network endpoints that are authoritative for an origin
- the "Alt-Svc Field Value", a text format for representing this information
- 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:

- 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:

- An "origin" is an information source as in [RFC6454].
The "origin server" is the server that the client would reach when accessing the origin in the absence of Alt-Svc.

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:

- a 1 octet flag field for SvcRecordType, interpreted as an unsigned numeric value (0 to 255, with only values "0" and "1" defined here)

- a 2 octet field for SvcFieldPriority as an integer in network byte order. If SvcRecordType is "0", SvcFieldPriority MUST be 0.

- the uncompressed SvcDomainName, represented as a sequence of length-prefixed labels as in Section 3.1 of [RFC1035].
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 RRSets, all RRs must have the same value for SvcRecordType. Clients and recursive servers MUST ignore HTTPSSVC resource records with other SvcRecordType values. If an RRSets 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
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:WRITEME

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 [HappyEybellsV2]
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 RRTyp could be extended to support schemes other than "https". Any such scheme MUST have an entry under the HTTPSSVC RRTyp 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:

<table>
<thead>
<tr>
<th>TYPE</th>
<th>Meaning</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTPSSVC</td>
<td>HTTPS Service Location</td>
<td>(This document)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>RR TYPE</th>
<th>_NODE NAME</th>
<th>Meaning</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTPSSVC</td>
<td>_https</td>
<td>Alt-Svc for HTTPS</td>
<td>(This document)</td>
</tr>
</tbody>
</table>

Schwartz, et al. Expires January 9, 2020
Per [AltSvc], please add the following entries to the HTTP Alt-Svc Parameter Registry:

<table>
<thead>
<tr>
<th>Alt-Svc Parameter</th>
<th>Meaning</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>esnikeys</td>
<td>Encrypted SNI keys</td>
<td>(This document)</td>
</tr>
</tbody>
</table>

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


11.2. Informative References


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:

- SRV records are typically mandatory, whereas clients will always continue to function correctly without making use of Alt-Svc or HTTPSSVC.
- SRV records cannot instruct the client to switch or upgrade protocols, whereas Alt-Svc can signal such an upgrade (e.g. to HTTP/2).
- 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.
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

- draft-nygren-httpbis-httpssvc-03
  * Change redirect type for HSTS-style behavior from 302 to 307 to reduce ambiguities.

- 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".

- draft-nygren-httpbis-httpssvc-01
  * Fixes of record name. Replace references to "HTTPSVC" with "HTTPSSVC".

- 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
Compact TLS 1.3
draft-rescorla-tls-ctls-03

Abstract

This document specifies a "compact" version of TLS 1.3. It is isomorphic to TLS 1.3 but saves space by trimming obsolete material, tighter encoding, and a template-based specialization technique. cTLS is not directly interoperable with TLS 1.3, but it should eventually be possible for a cTLS/TLS 1.3 server to exist and successfully interoperate.

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 7, 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
1. Introduction

DISCLAIMER: This is a work-in-progress draft of cTLS and has not yet seen significant security analysis, so could contain major errors. It should not be used as a basis for building production systems.
This document specifies a "compact" version of TLS 1.3 [RFC8446]. It is isomorphic to TLS 1.3 but designed to take up minimal bandwidth. The space reduction is achieved by four basic techniques:

- Omitting unnecessary values that are a holdover from previous versions of TLS.
- Omitting the fields and handshake messages required for preserving backwards-compatibility with earlier TLS versions.
- More compact encodings, omitting unnecessary values.
- A template-based specialization mechanism that allows for the creation of application specific versions of TLS that omit unnecessary values.

For the common (EC)DHE handshake with pre-established certificates, cTLS achieves an overhead of 45 bytes over the minimum required by the cryptovariables. For a PSK handshake, the overhead is 21 bytes. Annotated handshake transcripts for these cases can be found in Appendix A.

Because cTLS is semantically equivalent to TLS, it can be viewed either as a related protocol or as a compression mechanism. Specifically, it can be implemented by a layer between the TLS handshake state machine and the record layer.

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.

Structure definitions listed below override TLS 1.3 definitions; any PDU not internally defined is taken from TLS 1.3 except for replacing integers with varints.

3. Common Primitives

3.1. Varints

cTLS makes use of variable-length integers in order to allow a wide integer range while still providing for a minimal encoding. The width of the integer is encoded in the first two bits of the field as follows, with xs indicating bits that form part of the integer.
<table>
<thead>
<tr>
<th>Bit pattern</th>
<th>Length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxxxxxx</td>
<td>1</td>
</tr>
<tr>
<td>10xxxxx xxxxxxxx</td>
<td>2</td>
</tr>
<tr>
<td>11xxxxxx xxxxxxxxxx xxxxxxxx</td>
<td>3</td>
</tr>
</tbody>
</table>

Thus, one byte can be used to carry values up to 127.

In the TLS syntax variable integers are denoted as "varint" and a vector with a top range of a varint is denoted as:

```plaintext```
opaque foo<1..V>;
```

With a few exceptions, cTLS replaces every integer in TLS with a varint.

3.2. Record Layer

The cTLS Record Layer assumes that records are externally framed (i.e., that the length is already known because it is carried in a UDP datagram or the like). Depending on how this was carried, you might need another byte or two for that framing. Thus, only the type byte need be carried and TLSPlaintext becomes:

```c
struct {
    ContentType type;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

In addition, because the epoch is known in advance, the dummy content type is not needed for the ciphertext, so TLSCiphertext becomes:
struct {
    opaque content[TLSPlaintext.length];
    ContentType type;
    uint8 zeros[length_of_padding];
} TLSInnerPlaintext;

struct {
    opaque encrypted_record[TLSChiphertext.length];
} TLSCiphertext;

Note: The user is responsible for ensuring that the sequence numbers/nonces are handled in the usual fashion.

3.3. Handshake Layer

The cTLS handshake framing is same as the TLS 1.3 handshake framing, except for two changes:

1. The length field is omitted

2. The HelloRetryRequest message is a true handshake message instead of a specialization of ServerHello.

struct {
    HandshakeType msg_type;  /* handshake type */
    select (Handshake.msg_type) {
        case client_hello:      ClientHello;
        case server_hello:      ServerHello;
        case hello_retry_request: HelloRetryRequest;
        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;
    }
} Handshake;

3.4. Extensions

cTLS Extensions are the same as TLS 1.3 extensions, except varint length coded:
struct {
    varint extension_type;
    opaque extension_data<0..V>;
} Extension;

4. Handshake Messages

In general, we retain the basic structure of each individual TLS handshake message. However, the following handshake messages have been modified for space reduction and cleaned up to remove pre TLS 1.3 baggage.

4.1. ClientHello

The cTLS ClientHello is as follows.

opaque Random[RandomLength];      // variable length

struct {
    Random random;
    CipherSuite cipher_suites<1..V>;
    Extension extensions<1..V>;
} ClientHello;

4.1.1. KeyShare, SupportedGroups, and SignatureAlgorithms

KeyShare, SupportedGroups, and SignatureAlgorithms are identical to in TLS 1.3, except for the use of varints instead of integers. Note that because all of the EC DH groups are below 0x80, they will fit into a single byte. This is not true for signature algorithms.

[[OPEN ISSUE: Should we map signature algorithms into a smaller space?]]

4.1.2. PreSharedKeys

PreSharedKeys is the same as in TLS 1.3, except for the use of varints instead of integers.

[[OPEN ISSUE: Limiting this to one value would potentially save some bytes here, at the cost of generality.]]

4.2. ServerHello

We redefine ServerHello in a similar way:
struct {
    Random random;
    CipherSuite cipher_suite;
    Extension extensions<1..V>;
} ServerHello;

4.3. EncryptedExtensions

Likewise, EncryptedExtensions now uses a varint length field.

struct {
    Extension extensions<0..V>;
} EncryptedExtensions;

[[OPEN ISSUE: We could save 2 bytes in handshake header by omitting this value when it’s unneeded.]]

4.4. CertificateRequest

This message uses varint lengths and re-encodes the extensions.

struct {
    opaque certificate_request_context<0..V>
    Extension extensions<1..V>;
} CertificateRequest;

4.5. Certificate

We can slim down the Certificate message somewhat.
enum {
    X509(0),
    RawPublicKey(2),
    (255)
} CertificateType;

struct {
    select (certificate_type) {
        case RawPublicKey:
            /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
            opaque ASN1_subjectPublicKeyInfo<1..V>;
        case X509:
            opaque cert_data<1..V>;
    }
    Extension extensions<0..V>;
} CertificateEntry;

struct {
    opaque certificate_request_context<0..V>
    CertificateEntry certificate_list<1..V>;
} Certificate;

4.6. CertificateVerify

This just removes the length field. ˜˜˜˜ struct { SignatureScheme
algorithm; // TODO - define one byte schemes? opaque signature<1..V>;
} CertificateVerify; ˜˜˜˜

4.7. Finished

Unchanged.

4.8. HelloRetryRequest

The HelloRetryRequest has the following format:

struct {
    CipherSuite cipher_suite;
    Extension extensions<2..V>;
} HelloRetryRequest;

It is the same as the ServerHello above but without the unnecessary
sentinel Random value.
5. Template-Based Specialization

The protocol in the previous section is fully general and isomorphic to TLS 1.3; effectively it’s just a small cleanup of the wire encoding to match what we might have done starting from scratch. It achieves some compaction, but only a modest amount. cTLS also includes a mechanism for achieving very high compaction using template-based specialization.

The basic idea is that we start with the basic TLS 1.3 handshake, which is fully general and then remove degrees of freedom, eliding parts of the handshake which are used to express those degrees of freedom. For example, if we only support one version of TLS, then it is not necessary to have version negotiation and the supported_versions extension can be omitted.

Importantly, this process is performed only for the wire encoding but not for the handshake transcript. The result is that the transcript for a specialized cTLS handshake is the same as the transcript for a TLS 1.3 handshake with the same features used.

One way of thinking of this is as if specialization is a stateful compression layer between the handshake and the record layer:

```
+---------------+---------------+---------------+    +---------+
|   Handshake   |  Application  |     Alert     |<---| Profile |
+---------------+---------------+---------------+    +---------+
|               cTLS Compression Layer                            |
+---------------+---------------+---------------+    +---------+
|               cTLS Record Layer / Application                  |
```

Specializations are defined by a "compression profile" that specifies what features are to be optimized out of the handshake. In the following subsections, we define the structure of these profiles, and how they are used in compressing and decompressing handshake messages.

[[OPEN ISSUE: Do we want to have an explicit cTLS extension indicating that cTLS is in use and which specialization is in use? This goes back to whether we want the use of cTLS to be explicit.]]

5.1. Specifying a Specialization

A compression profile defining of a specialized version of TLS is defined using a JSON dictionary. Each axis of specialization is a key in the dictionary. [[OPEN ISSUE: If we ever want to serialize this, we'll want to use a list instead.]].

For example, the following specialization describes a protocol with a
single fixed version (TLS 1.3) and a single fixed cipher suite
(TLS_AES_128_GCM_SHA256). On the wire, ClientHello.cipher_suites,
ServerHello.cipher_suites, and the supported_versions extensions in
the ClientHello and ServerHello would be omitted.

```
{
  "version" : 772,
  "cipherSuite" : "TLS_AES_128_GCM_SHA256"
}
```

cTLS allows specialization along the following axes:

- **version** (integer): indicates that both sides agree to the single TLS
  version specified by the given integer value (772 == 0x0304 for
  TLS 1.3). The supported_versions extension is omitted from
  ClientHello.extensions and reconstructed in the transcript as a
  single-valued list with the specified value. The supported_versions
  extension is omitted from ClientHello.extensions and reconstructed in the transcript with
  the specified value.

- **cipherSuite** (string): indicates that both sides agree to the single
  named cipher suite, using the "TLS_AEAD_HASH" syntax defined in
  [RFC8446], Section 8.4. The ClientHello.cipher_suites field is
  omitted and reconstructed in the transcript as a single-valued
  list with the specified value. The server_hello.cipher_suite
  field is omitted and reconstructed in the transcript as the
  specified value.

- **dhGroup** (string): specifies a single DH group to use for key
  establishment. The group is listed by the code point name in
  [RFC8446], Section 4.2.7. (e.g., x25519). This implies a literal
  "supported_groups" extension consisting solely of this group.

- **signatureAlgorithm** (string): specifies a single signature scheme to
  use for authentication. The group is listed by the code point
  name in [RFC8446], Section 4.2.7. (e.g., ed25519). This implies
  a literal "signature_algorithms" extension consisting solely of
  this group.

- **randomSize** (integer): indicates that the ClientHello.Random and
  ServerHello.Random values are truncated to the given values. When
  the transcript is reconstructed, the Random is padded to the right
  with 0s and the anti-downgrade mechanism in ([RFC8446]),
  Section 4.1.3 is disabled. IMPORTANT: Using short Random values
  can lead to potential attacks. When Random values are shorter
  than 8 bytes, PSK-only modes MUST NOT be used, and each side MUST
use fresh DH ephemerals. The Random length MUST be less than or equal to 32 bytes.

clientHelloExtensions (predefined extensions): Predefined ClientHello extensions, see {predefined-extensions}

serverHelloExtensions (predefined extensions): Predefined ServerHello extensions, see {predefined-extensions}

encryptedExtensions (predefined extensions): Predefined EncryptedExtensions extensions, see {predefined-extensions}

certRequestExtensions (predefined extensions): Predefined CertificateRequest extensions, see {predefined-extensions}

knownCertificates (known certificates): A compression dictionary for the Certificate message, see {known-certs}

finishedSize (integer): indicates that the Finished value is to be truncated to the given length. When the transcript is reconstructed, the remainder of the Finished value is filled in by the receiving side. [[OPEN ISSUE: How short should we allow this to be? TLS 1.3 uses the native hash and TLS 1.2 used 12 bytes. More analysis is needed to know the minimum safe Finished size. See [RFC8446]; Section E.1 for more on this, as well as https://mailarchive.ietf.org/arch/msg/tls/TugB5ddJu3nYg7chcyeyUqW5bA.]]

5.1.1. Requirements on the TLS Implementation

To be compatible with the specializations described in this section, a TLS stack needs to provide two key features:

If specialization of extensions is to be used, then the TLS stack MUST order each vector of Extension values in ascending order according to the ExtensionType. This allows for a deterministic reconstruction of the extension list.

If truncated Random values are to be used, then the TLS stack MUST be configurable to set the remaining bytes of the random values to zero. This ensures that the reconstructed, padded random value matches the original.

If truncated Finished values are to be used, then the TLS stack MUST be configurable so that only the provided bytes of the Finished are verified, or so that the expected remaining values can be computed.
5.1.2. Predefined Extensions

Extensions used in the ClientHello, ServerHello, EncryptedExtensions, and CertificateRequest messages can be "predefined" in a compression profile, so that they do not have to be sent on the wire. A predefined extensions object is a dictionary whose keys are extension names specified in the TLS ExtensionTypeRegistry specified in [RFC8446]. The corresponding value is a hex-encoded value for the ExtensionData field of the extension.

When compressing a handshake message, the sender compares the extensions in the message being compressed to the predefined extensions object, applying the following rules:

- If the extensions list in the message is not sorted in ascending order by extension type, it is an error, because the decompressed message will not match.
- If there is no entry in the predefined extensions object for the type of the extension, then the extension is included in the compressed message.
- If there is an entry:
  * If the ExtensionData of the extension does not match the value in the dictionary, it is an error, because decompression will not produce the correct result.
  * If the ExtensionData matches, then the extension is removed, and not included in the compressed message.

When decompressing a handshake message the receiver reconstitutes the original extensions list using the predefined extensions:

- If there is an extension in the compressed message with a type that exists in the predefined extensions object, it is an error, because such an extension would not have been sent by a sender with a compatible compression profile.
- For each entry in the predefined extensions dictionary, an extension is added to the decompressed message with the specified type and value.
- The resulting vector of extensions MUST be sorted in ascending order by extension type.
Note that the "version", "dhGroup", and "signatureAlgorithm" fields in the compression profile are specific instances of this algorithm for the corresponding extensions.

[[OPEN ISSUE: Are there other extensions that would benefit from special treatment, as opposed to hex values.]]

5.1.3. Known Certificates

Certificates are a major contributor to the size of a TLS handshake. In order to avoid this overhead when the parties to a handshake have already exchanged certificates, a compression profile can specify a dictionary of "known certificates" that effectively acts as a compression dictionary on certificates.

A known certificates object is a JSON dictionary whose keys are strings containing hex-encoded compressed values. The corresponding values are hex-encoded strings representing the uncompressed values. For example:

```json
{
    "00": "3082...",
    "01": "3082...",
}
```

When compressing a Certificate message, the sender examines the cert_data field of each CertificateEntry. If the cert_data matches a value in the known certificates object, then the sender replaces the cert_data with the corresponding key. Decompression works the opposite way, replacing keys with values.

Note that in this scheme, there is no signaling on the wire for whether a given cert_data value is compressed or uncompressed. Known certificates objects SHOULD be constructed in such a way as to avoid an uncompressed object being mistaken for compressed one and erroneously decompressed. For X.509, it is sufficient for the first byte of the compressed value (key) to have a value other than 0x30, since every X.509 certificate starts with this byte.

6. Examples

The following section provides some example specializations.

TLS 1.3 only:

```json
{
    "Version" : 0x0304
}
```
TLS 1.3 with AES_GCM and X25519 and ALPN h2, short random values, and everything else is ordinary TLS 1.3.

{
"Version" : 772,
"Random": 16,
"CipherSuite" : "TLS_AES_128_GCM_SHA256",
"DHGroup" : "X25519",
"Extensions": {
    "named_groups": 29,
    "application_layer_protocol_negotiation" : "030016832",
    "..." : null
}
}

Version 772 corresponds to the hex representation 0x0304, named group "29" (0x001D) represents X25519.

[[OPEN ISSUE: Should we have a registry of well-known profiles?]]

7. Security Considerations

WARNING: This document is effectively brand new and has seen no analysis. The idea here is that cTLS is isomorphic to TLS 1.3, and therefore should provide equivalent security guarantees.

The use of key ids is a new feature introduced in this document, which requires some analysis, especially as it looks like a potential source of identity misbinding. This is, however, entirely separable from the rest of the specification.

Transcript expansion also needs some analysis and we need to determine whether we need an extension to indicate that cTLS is in use and with which profile.

8. IANA Considerations

This document has no IANA actions.

9. Normative References

Appendix A. Sample Transcripts

In this section, we provide annotated example transcripts generated using a draft implementation of this specification in the mint TLS library. The transcripts shown are with the revised message formats defined above, as well as specialization to the indicated cases, using the aggressive compression profiles noted below. The resulting byte counts are as follows:

<table>
<thead>
<tr>
<th></th>
<th>ECDHE</th>
<th>PSK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TLS</td>
<td>CTLS</td>
</tr>
<tr>
<td>ClientHello</td>
<td>132</td>
<td>50</td>
</tr>
<tr>
<td>ServerHello</td>
<td>90</td>
<td>48</td>
</tr>
<tr>
<td>ServerFlight</td>
<td>478</td>
<td>104</td>
</tr>
<tr>
<td>ClientFlight</td>
<td>458</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>1158</td>
</tr>
</tbody>
</table>

To increase legibility, we show the plaintext bytes of handshake messages that would be encrypted and shorten some of the cryptographic values (shown with "..."). The totals above include 9 bytes of encryption overhead for the client and server flights, which would otherwise be encrypted (with a one-byte content type and an 8-byte tag).

Obviously, these figures are very provisional, and as noted at several points above, there are additional opportunities to reduce overhead.

NOTE: We are using a shortened Finished message here. See Section 5.1 for notes on Finished size. However, the overhead is constant for all reasonable Finished sizes.]

A.1. ECDHE and Mutual Certificate-based Authentication

Compression Profile:
Internet-Draft                  cTLS 1.3                   November 2019

{
    "version": 772,
    "cipherSuite": "TLS_AES_128_CCM_8_SHA256",
    "dhGroup": "X25519",
    "signatureAlgorithm": "ECDSA_P256_SHA256",
    "randomSize": 8,
    "finishedSize": 8,
    "clientHelloExtensions": {
        "server_name": "000e00000b6578616d706c652e636f6d",
    },
    "certificateRequestExtensions": {
        "signature_algorithms": "00020403"
    },
    "knownCertificates": {
        "61": "3082...",
        "62": "3082..."
    }
}

ClientHello: 50 bytes = RANDOM(8) + DH(32) + Overhead(10)

01                    // ClientHello
2ef16120dd84a721      // Random
28                    // Extensions.length
33 26                 // KeyShare
0024                // client_shares.length
001d              // KeyShareEntry.group
0020 a690...af948 // KeyShareEntry.key_exchange

ServerHello: 48 = RANDOM(8) + DH(32) + Overhead(8)

02                 // ServerHello
962547bba5e00973   // Random
26                 // Extensions.length
33 24              // KeyShare
001d              // KeyShareEntry.group
0020 9fbc...0f49 // KeyShareEntry.key_exchange

Server Flight: 96 = SIG(71) + MAC(8) + CERTID(1) + Overhead(16)
08  // EncryptedExtensions
00  // Extensions.length
0d  // CertificateRequest
00  // CertificateRequestContext.length
00  // Extensions.length
0b  // Certificate
00  // CertificateRequestContext
03  // CertificateList
01  // CertData.length
61  // CertData = 'a'
00  // Extensions.length
0f  // CertificateVerify
0403  // SignatureAlgorithm
4047 3045...10ce // Signature
14  // Finished
bfc9d6671bb2b04 // VerifyData

Client Flight: 91 bytes = SIG(71) + MAC(8) + CERTID(1) + Overhead(11)

0b  // Certificate
00  // CertificateRequestContext
03  // CertificateList
01  // CertData.length
62  // CertData = 'b'
00  // Extensions.length
0f  // CertificateVerify
0403  // SignatureAlgorithm
4047 3045...f60e // Signature.length
14  // Finished
35e9c34ec2c5dc1 // VerifyData

A.2. PSK

Compression Profile:
{  "version": 772,
  "cipherSuite": "TLS_AES_128_CCM_8_SHA256",
  "signatureAlgorithm": "ECDSA_P256_SHA256",
  "randomSize": 16,
  "finishedSize": 0,
  "clientHelloExtensions": {
    "server_name": "000e00000b6578616d706c652e636f6d",
    "psk_key_exchange_modes": "0100"
  },
  "serverHelloExtensions": {
    "pre_shared_key": "0000"
  }
}

ClientHello: 67 bytes = RANDOM(16) + PSKID(4) + BINDER(32) + Overhead(15)

01 // ClientHello  
e230115e62d9a3b58f73e0f2896b2e35 // Random  
2d // Extensions.length  
29 2b // PreSharedKey  
000a // identities.length  
0004 00010203 // identity  
7bd05af6 // obfuscated_ticket_age  
0021 // binders.length  
20 2428...bb3f // binder

ServerHello: 18 bytes = RANDOM(16) + 2

02 // ServerHello  
7232e2d3e61e476b844d9c1f6a4c868f // Random  
00 // Extensions.length

Server Flight: 3 bytes = Overhead(3)

08 // EncryptedExtensions  
00 // Extensions.length  
14 // Finished

Client Flight: 1 byte = Overhead(3)

14 // Finished
Acknowledgments

We would like to thank Karthikeyan Bhargavan, Owen Friel, Sean Turner, Martin Thomson and Chris Wood.

Authors’ Addresses

Eric Rescorla
Mozilla

Email: ekr@rtfm.com

Richard Barnes
Cisco

Email: rlb@ipv.sx

Hannes Tschofenig
Arm Limited

Email: hannes.tschofenig@arm.com
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
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

- Enable TCP load balancers to forward metadata to the backend.
- Enable backends to reply.
- Reduce the need for TLS-terminating load balancers.
- Ensure confidentiality and integrity for all forwarded metadata.
- Enable split ESNI architectures.
- Prove to the backend that the load balancer intended to associate this metadata with this connection.
- Achieve good CPU and memory efficiency.
Don’t impose additional latency.

Support backends that receive a mixture of direct and load-balanced TLS.

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:

```
+-----------+ +-----------+ +-----------+
| Backend A | | Backend B | | Backend C |
+-----------+ +-----------+ +-----------+
```

4. EncryptedProxyData[ \\ / \\ 3. ClientHello (verbatim)
got SNI info] \\ / \\ 2. EncryptedProxyData[
5. ServerHello, etc. \\ / \\ SNI="secret.b", \\
\ / \ / client=2, etc.]
\\ / \ /

```
+---------------+
| Load balancer |
+---------------+
```

6. ServerHello, etc. \\ / \ / 1. ClientHello[
(verbatim) \\ / \ / ESNI=enc("secret.b")]
\\ / \ /

```
+-----------+ +-----------+ +-----------+
|  Client 1 | |  Client 2 | |  Client 3 |
+-----------+ +-----------+ +-----------+
```

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;

- "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.
- "nonce": Non-repeating initializer for the AEAD. This prevents an attacker from observing whether the same ClientHello is marked with different metadata over time.
- "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:

```c
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:

```c
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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
<th>DTLS-OK</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>XX</td>
<td>proxy_header</td>
<td>N</td>
<td>This document</td>
</tr>
</tbody>
</table>

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


10.2. Informative References


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
Design issues for hybrid key exchange in TLS 1.3
draft-stebila-tls-hybrid-design-01

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, and is motivated by transition to post-quantum cryptography. This document categorizes various design considerations for using hybrid key exchange in the Transport Layer Security (TLS) protocol version 1.3 and outlines two concrete instantiations for consideration.

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. Revision history ........................................ 3
   1.2. Terminology ........................................... 4
   1.3. Motivation for use of hybrid key exchange ............... 4
   1.4. Scope .................................................. 5
   1.5. Goals .................................................. 5
   1.6. Related work ........................................... 7

2. Overview ..................................................... 8

3. Design options ................................................. 10
   3.1. (Neg) How to negotiate hybridization and component algorithms? ........................................ 10
       3.1.1. Key exchange negotiation in TLS 1.3 ............... 10
       3.1.2. (Neg-Ind) Negotiating component algorithms individually ........................................ 10
       3.1.3. (Neg-Comb) Negotiating component algorithms as a combination ................................. 11
       3.1.4. Benefits and drawbacks .............................. 12
   3.2. (Num) How many component algorithms to combine? ....... 13
       3.2.1. (Num-2) Two ........................................ 13
       3.2.2. (Num-2+) Two or more ............................... 13
       3.2.3. Benefits and Drawbacks .............................. 13
   3.3. (Shares) How to convey key shares? ....................... 13
       3.3.1. (Shares-Concat) Concatenate key shares .......... 13
       3.3.2. (Shares-Multiple) Send multiple key shares ...... 14
       3.3.3. (Shares-Ext-Additional) Extension carrying additional key shares ......................... 14
       3.3.4. Benefits and Drawbacks .............................. 14
   3.4. (Comb) How to use keys? .................................. 14
       3.4.1. (Comb-Concat) Concatenate keys .................... 15
       3.4.2. (Comb-KDF-1) KDF keys ................................ 16
       3.4.3. (Comb-KDF-2) KDF keys ................................ 17
       3.4.4. (Comb-XOR) XOR keys ................................ 18
       3.4.5. (Comb-Chain) Chain of KDF applications for each key ............................................ 18
       3.4.6. (Comb-AltInput) Second shared secret in an alternate KDF input .................................. 19
       3.4.7. Benefits and Drawbacks .............................. 20
       3.4.8. Open questions ....................................... 21

4. Candidate instantiations ....................................... 21
   4.1. Candidate Instantiation 1 ............................... 21
       4.1.1. ClientHello extension supported_groups ........... 21
1. Introduction

This document categorizes various design decisions one could make when implementing hybrid key exchange in TLS 1.3, with the goal of fostering discussion, providing options for short-term prototypes/experiments, and serving as a basis for eventual standardization. This document also includes two concrete instantiations for consideration, following two different approaches; it is not our intention that both be standardized.

This document does not propose specific post-quantum mechanisms; see Section 1.4 for more on the scope of this document.

Comments are solicited and should be addressed to the TLS working group mailing list at tls@ietf.org and/or the author(s).

1.1. Revision history

- draft-00: Initial version.
- draft-01:
  - Add (Comb-KDF-1) (Section 3.4.2) and (Comb-KDF-2) (Section 3.4.3) options.
  - Add Candidate Instantiation 1 (Section 4.1).
* Add Candidate Instantiation 2 (Section 4.2).

1.2. Terminology

For the purposes of this document, it is helpful to be able to divide cryptographic algorithms into two classes:

- "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.

- "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 mechanisms based on different cryptographic assumptions (for example, 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 mechanisms remains unbroken. We use the term "component" algorithms to refer to the algorithms that are being 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.

1.3. Motivation for use of hybrid key exchange

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 (but not all) of the 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 / elliptic curve Diffie-Hellman, and thus we do not necessarily have as much confidence in
their fundamental security, or the concrete security level of specific parameterizations.

Early adopters eager for post-quantum security may want to use hybrid key exchange to have the potential of post-quantum security 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.)

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:

- Selecting which next-generation algorithms to use in TLS 1.3, nor algorithm identifiers nor encoding mechanisms for next-generation algorithms. (The outcomes of the NIST Post-Quantum Cryptography Standardization Project [NIST] will inform this choice.)

- 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. But 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 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:

- *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.

- **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.

- **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 / ciphertext sizes for post-quantum algorithms range from hundreds of bytes to over one hundred kilobytes, so this impact can be substantially. 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.

- **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.
*No duplicate information:* Attempting to negotiate hybrid key
exchange should not mean having to send multiple public keys of
the same type.

1.6. 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:

- Internet-Drafts for TLS 1.2: [WHYTE12]
- 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:

- Experimental implementations in TLS 1.2: [BCNS15], [CECPQ1],
  [FRODO], [OQS-102]
- 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.

2. Overview

We identify four distinct axes along which one can make choices 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 remainder of this document 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) (Section 3.1) How to negotiate the use of hybridization in general and component algorithms specifically?

   * (Neg-Ind) (Section 3.1.2) Negotiating component algorithms individually

      + (Neg-Ind-1) (Section 3.1.2.1) Traditional algorithms in "ClientHello" "supported_groups" extension, next-gen algorithms in another extension

      + (Neg-Ind-2) (Section 3.1.2.2) Both types of algorithms in "supported_groups" with external mapping to tradition/next-gen.

      + (Neg-Ind-3) (Section 3.1.2.3) Both types of algorithms in "supported_groups" separated by a delimiter.

   * (Neg-Comb) (Section 3.1.3) Negotiating component algorithms as a combination

      + (Neg-Comb-1) (Section 3.1.3.1) Standardize "NamedGroup" identifiers for each desired combination.
+ (Neg-Comb-2) (Section 3.1.3.2) Use placeholder identifiers in "supported_groups" with an extension defining the combination corresponding to each placeholder.

+ (Neg-Comb-3) (Section 3.1.3.3) List combinations by inserting grouping delimiters into "supported_groups" list.

2. (Num) (Section 3.2) How many component algorithms can be combined?

* (Num-2) (Section 3.2.1) Two.

* (Num-2+) (Section 3.2.2) Two or more.

3. (Shares) (Section 3.3) How should multiple key shares (public keys / ciphertexts) be conveyed?

* (Shares-Concat) (Section 3.3.1) Concatenate each combination of key shares.

* (Shares-Multiple) (Section 3.3.2) Send individual key shares for each algorithm.

* (Shares-Ext-Additional) (Section 3.3.3) Use an extension to convey key shares for component algorithms.

4. (Comb) (Section 3.4) How should multiple shared secrets be combined?

* (Comb-Concat) (Section 3.4.1) Concatenate the shared secrets then use directly in the TLS 1.3 key schedule.

* (Comb-KDF-1) (Section 3.4.2) and (Comb-KDF-2) (Section 3.4.3) KDF the shared secrets together, then use the output in the TLS 1.3 key schedule.

* (Comb-XOR) (Section 3.4.4) XOR the shared secrets then use directly in the TLS 1.3 key schedule.

* (Comb-Chain) (Section 3.4.5) Extend the TLS 1.3 key schedule so that there is a stage of the key schedule for each shared secret.

* (Comb-AltInput) (Section 3.4.6) Use the second shared secret in an alternate (otherwise unused) input in the TLS 1.3 key schedule.
3. Design options

3.1. (Neg) How to negotiate hybridization and component algorithms?

3.1.1. 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.

3.1.2. (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.

3.1.2.1. (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].)

3.1.2.2. (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.

3.1.2.3. (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".

3.1.3. (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.

3.1.3.1. (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].

3.1.3.2. (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.

3.1.3.3. (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.

3.1.4. Benefits and drawbacks

*Combinatorial explosion.* (Neg-Comb-1) (Section 3.1.3.1) requires new identifiers to be defined for each desired combination. The other 4 options in this section do not.

*Extensions.* (Neg-Ind-1) (Section 3.1.2.1) and (Neg-Comb-2) (Section 3.1.3.2) 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) (Section 3.1.3.1) require new logic to process negotiation.

*Matching security levels.* (Neg-Ind-1) (Section 3.1.2.1), (Neg-Ind-2) (Section 3.1.2.2), (Neg-Ind-3) (Section 3.1.2.3), and (Neg-Comb-2) (Section 3.1.3.2) 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) (Section 3.1.2.1), (Neg-Ind-2) (Section 3.1.2.2), and (Neg-Ind-3) (Section 3.1.2.3) give control over the combination to the server; (Neg-Comb-2) (Section 3.1.3.2) gives control over the combination to the client. (Neg-Comb-1) (Section 3.1.3.1) 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) (Section 3.1.2.2), (Neg-Ind-3) (Section 3.1.2.3), (Neg-Comb-1) (Section 3.1.3.1), (Neg-Comb-2) (Section 3.1.3.2) and unrecognized "ClientHello" extensions (Neg-Ind-1) (Section 3.1.2.1), (Neg-Comb-2) (Section 3.1.3.2). In (Neg-Ind-3) (Section 3.1.2.3) and (Neg-Comb-3) (Section 3.1.3.3), 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.

3.2. (Num) How many component algorithms to combine?

3.2.1. (Num-2) Two

Exactly two algorithms can be combined together in hybrid key exchange. This is the approach taken in [KIEFER] and [SCHANCK].

3.2.2. (Num-2+) Two or more

Two or more algorithms can be combined together in hybrid key exchange. This is the approach taken in [WHYTE13].

3.2.3. 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.

3.3. (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.

3.3.1. (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].
3.3.2. (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.)

3.3.3. (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].

3.3.4. Benefits and Drawbacks

*Backwards compatibility.* (Shares-Multiple) (Section 3.3.2) is fully backwards compatible with non-hybrid-aware servers. (Shares-Ext-Additional) (Section 3.3.3) is backwards compatible with non-hybrid-aware servers provided they ignore unrecognized extensions. (Shares-Concat) (Section 3.3.1) 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) (Section 3.3.3) and (Shares-Concat) (Section 3.3.1). (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) (Section 3.3.2) does not result in duplicate key shares.

3.4. (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.
3.4.1. (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:

```
0 | v
PSK -> HKDF-Extract = Early Secret
    +-----+ Derive-Secret(...)
    +-----+ Derive-Secret(...)
    +-----+ Derive-Secret(...)
    v
    Derive-Secret(., "derived", "")
    v
concatenated_shared_secret -> HKDF-Extract = Handshake Secret
    +-----+ Derive-Secret(...)
    +-----+ Derive-Secret(...)
    +-----+ Derive-Secret(...)
    v
    Derive-Secret(., "derived", "")
    v
0 -> HKDF-Extract = Master Secret
    +-----+ Derive-Secret(...)
    +-----+ Derive-Secret(...)
    +-----+ Derive-Secret(...)
    +-----+ Derive-Secret(...)
```

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.

3.4.2. (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:

```
0
  v
PSK -> HKDF-Extract = Early Secret
    +-----+ Derive-Secret(...) \\
    +-----+ Derive-Secret(...) \\
    +-----+ Derive-Secret(...)

Next-Gen
  v
(EC)DHE -> HKDF-Extract
    v
output +-----+ HKDF-Extract = Handshake Secret
           v
             +-----+ Derive-Secret(...) \\
             +-----+ Derive-Secret(...) \\
             +-----+ Derive-Secret(...)
             v
             Derive-Secret(., "derived", ")"
             v
0 -> HKDF-Extract = Master Secret
    +-----+ Derive-Secret(...) \\
    +-----+ Derive-Secret(...) \\
    +-----+ Derive-Secret(...) \\
    +-----+ Derive-Secret(...) 
```
3.4.3. (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) (Section 3.4.2), 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) (Section 3.4.1), 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.

```
  0  
   |  
PSK -> HKDF-Extract = Early Secret 
   +------> Derive-Secret(...) 
   +------> Derive-Secret(...) 
   +------> Derive-Secret(...) 
      
concatenated secret -> HKDF-Extract Derive-Secret(., "derived", "") 
    ^^^^^^^^^ 
      
output -----> HKDF-Extract = Handshake Secret 
    ^^^^^^^ 
      
      +------> Derive-Secret(...) 
      +------> Derive-Secret(...) 
      +------> Derive-Secret(...) 
         
Derive-Secret(., "derived", "") 
      
  0 -> HKDF-Extract = Master Secret 
    +------> Derive-Secret(...) 
    +------> Derive-Secret(...) 
    +------> Derive-Secret(...) 
    +------> Derive-Secret(...) 
```
3.4.4. (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).

3.4.5. (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(k1 || F(k2))". 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:
PSK $\rightarrow$ HKDF-Extract = Early Secret
  $\rightarrow$ Derive-Secret(...)
  $\rightarrow$ Derive-Secret(...)
  $\rightarrow$ Derive-Secret(...)
  Derive-Secret(., "derived", "")

traditional_shared_secret $\rightarrow$ HKDF-Extract
  Derive-Secret(., "derived", "")

next_gen_shared_secret $\rightarrow$ HKDF-Extract = Handshake Secret
  Derive-Secret(., "derived", "")

0 $\rightarrow$ HKDF-Extract = Master Secret
  Derive-Secret(...)
  Derive-Secret(...)
  Derive-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.

3.4.6. (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].

3.4.7. Benefits and Drawbacks

*New logic.* While (Comb-Concat) (Section 3.4.1), (Comb-KDF-1) (Section 3.4.2), and (Comb-KDF-2) (Section 3.4.3) 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) (Section 3.4.5) 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) (Section 3.4.5) continues that approach.

*Efficiency.* (Comb-KDF-1) (Section 3.4.2), (Comb-KDF-2) (Section 3.4.3), and (Comb-Chain) (Section 3.4.5) increase the number
of KDF applications for each component algorithm, whereas (Comb-Concat) (Section 3.4.1) and (Comb-AltInput) (Section 3.4.6) keep the number of KDF applications the same (though with potentially longer inputs).

*Extensibility.* (Comb-AltInput) (Section 3.4.6) 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) (Section 3.4.1) and (Comb-Chain) (Section 3.4.5) can naturally accommodate more than 2 component shared secrets since there is no distinction to how each shared secret is treated. (Comb-AltInput) (Section 3.4.6) 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.

3.4.8. Open questions

At this point, it is unclear which, if any, of the above methods preserve FIPS compliance: i.e., if one shared secret is from a FIPS-compliant method (e.g., ECDH), and another shared secret is from a non-approved method (e.g., post-quantum), is the result still considered FIPS compliant? Guidance from NIST on this question would be helpful. Specifically, are any of these approaches acceptable under either [NIST-SP-800-56C] or [NIST-SP-800-135]?

4. Candidate instantiations

In this section, we describe two candidate instantiations of hybrid key exchange in TLS 1.3, based on the design considerations framework above. It is not our intention that both of these instantiations be standardized; we are providing two for discussion and for comparing and contrasting the two approaches.

4.1. Candidate Instantiation 1

Candidate Instantiation 1 allows for two or more component algorithms to be combined (Num-2+) (Section 3.2.2), and negotiates the combination using markers in the "NamedGroup" list as pointers to an extension listing the algorithms comprising each possible combination (Neg-Comb-2) (Section 3.1.3.2) following the approach of [WHYTE13]. The client conveys its multiple key shares individually in the "client_shares" vector of the "ClientHello" "key_share" extension (Shares-Multiple) (Section 3.3.2). The server conveys its multiple key shares concatenated together in its "KeyShareServerHello" struct...
(Shares-Concat) (Section 3.3.1). The shared secrets are combined by concatenating them then feeding them through a KDF, then feeding the result into the TLS 1.3 key schedule (Comb-KDF-2) (Section 3.4.3).

4.1.1. ClientHello extension supported_groups

Following [WHYTE13] section 3.1, the "NamedGroup" enum used by the client to populate the "supported_groups" extension is extended to include new code points representing markers for hybrid combinations:

```c
enum {
    /* existing named groups */
    secp256r1 (23),
    ...
    /* new code points eventually defined for post-quantum algorithms */
    ...
    /* new code points reserved for hybrid markers */
    hybrid_marker00 (0xFD00),
    hybrid_marker01 (0xFD01),
    ...
    hybrid_markerFF (0xFDFF),
    /* existing reserved code points */
    ff dhe_private_use (0x01FC..0x01FF),
    ecdhe_private_use (0xFE00..0xFEFF),
    (0xFFFF)
} NamedGroup;
```

"hybrid_marker" code points do not a priori represent any fixed combination. Instead, during each session establishment, the client defines what it wants each "hybrid_marker" code point to represent using the following extension.

4.1.2. ClientHello extension hybrid_extension

Following [WHYTE13] section 3.2.4, a new "ClientHello" "hybrid_extension" extension is defined. It is defined as follows:

```c
struct {
    NamedGroup hybrid_marker;
    NamedGroup components<2..10>;
} HybridMapping;

struct {
    HybridMapping map<0..255>;
} HybridExtension;
```
The "HybridExtension" contains 0 or more "HybridMapping"s. Each "HybridMapping" corresponds to one of the "hybrid_marker" included in the "supported_groups" extension, and lists the component algorithms that are meant to comprise the this hybrid combination, which can be any of the existing named groups (elliptic curve or finite field), new code points eventually defined for post-quantum algorithms, or reserved code points for private use.

4.1.3. ClientHello extension key_share

No syntactical modifications are made to the "KeyShareEntry" or "KeyShareClientHello" data structures.

Semantically, the client does not send a "KeyShareEntry" corresponding to any of the "hybrid_marker" code points. Instead, the client sends "KeyShareEntry" for each of the component algorithms listed in the "HybridMapping"s.

For example, if the list of "supported_groups" is "secp256r1", "x25519", "hybrid_marker00", and "hybrid_marker01", where "hybrid_marker00" comprises "secp256r1" with a fictional post-quantum algorithm "PQ1", and "hybrid_marker01" comprises "x25519" with "PQ1", then the client could send three "KeyShareEntry" components: one for "secp256r1", one for "x25519", and one for "PQ1".

4.1.4. ServerHello extension KeyShareServerHello

The server responds with a "KeyShareServerHello" struct containing a single "KeyShareEntry", which contains a single "NamedGroup" value and an opaque "key_exchange" string.

To complete the negotiation of a hybrid algorithm, the server responds with the "NamedGroup" value being the "hybrid_marker" code point correspond to the combination that the server was willing to agree to.

The "key_exchange" string is the octet representation of the following struct:

```
struct {
    KeyShareEntry key_share<2..10>;

    HybridKeyShare;
}
```

where there is one "key_share" entry for each of the components of this hybrid combination.

Note that the "key_exchange" string has a maximum length of 2^16-1 octets, which may be insufficient for some post-quantum algorithms or
for some hybridizations of multiple post-quantum algorithms. It remains an open question as to whether this length can be increased without breaking existing TLS 1.3 implementations.

4.1.5. Key schedule

The component algorithm shared secrets are combined by concatenating them, then applying a key derivation function, the output of which is then used in the TLS 1.3 key schedule in place of the (EC)DHE shared secret. The component shared secrets are concatenated in the order that they appear in the "components" vector of the "HybridMapping" extension above.

We provide two options for concatenating the shared secrets, and would like feedback from the working group in which to proceed with.

Each component algorithm’s "shared_secret" is defined by the algorithm itself, for example the DHE or ECDHE shared secrets as defined in Section 7.4 of [TLS13], or as defined by post-quantum methods once standardized in their own documents.

*Option 1: Using data structures.* Option 1 uses a full-fledged TLS 1.3 data structure to represent the list of component shared secrets. As a result, lengths of each shared secret are unambiguously encoded.

```c
struct SharedSecret {
    opaque shared_secret<0..2^16-1>;
}

struct {
    SharedSecret component<2..10>;
} HybridSharedSecret;
```

The "concatenated_shared_secret" is then the octet representation of the "HybridSharedSecret" struct.

*Option 2: Direct concatenation.* Option 2 directly concatenates the shared secrets. Option 2 should only be considered if the shared secret for each algorithm is guarantees to be of a fixed length, which would imply that, once the component algorithms are fixed, concatenation is bijective.

```c
concatenated_shared_secret = shared_secret0 | shared_secret1 | ...
```

In either option, the "concatenated_shared_secret" octet string is used as the IKM argument of HKDF-Extract, with the zero-length string as the salt argument. The output of HKDF-Extract is used as the IKM...
argument for HKDF-Extract’s calculation of the handshake secret, as shown below.

```
0
  v
PSK -> HKDF-Extract = Early Secret
     +----- Derive-Secret(...)
     +----- Derive-Secret(...)
     +----- Derive-Secret(...)
     v
concatenated 0
shared secret -> HKDF-Extract Derive-Secret(., "derived", ")
     ^^^^^^^
     v     v
output ----> HKDF-Extract = Handshake Secret
     ^^^^^^^
     v
     v
Derive-Secret(., "derived", ")
     v
0 -> HKDF-Extract = Master Secret
     +----- Derive-Secret(...)
     +----- Derive-Secret(...)
     +----- Derive-Secret(...)
     +----- Derive-Secret(...)```

4.2. Candidate Instantiation 2

Candidate Instantiation 2 allows for exactly two component algorithms to be combined (Num-2) (Section 3.2.1), and uses code points standardized for each permissible combination. The client concatenates its multiple key shares together as a distinct entry in the "client_shares" vector of the "ClientHello" "key_share" extension (Shares-Concat) (Section 3.3.1). The server does the same. The shared secrets are combined by concatenating them then feeding them through a KDF, then feeding the result into the TLS 1.3 key schedule (Comb-KDF-2) (Section 3.4.3).
4.2.1. ClientHello extension supported_groups

The "NamedGroup" enum used by the client to populate the "supported_groups" extension is extended to include new code points representing each desired combination.

For example,

```
enum {
    /* existing named groups */
    secp256r1 (23),
    x25519 (0x001D),
    ...,

    /* new code points eventually defined for post-quantum algorithms */
    PQ1 (0x????),
    PQ2 (0x????),
    ...,

    /* new code points defined for hybrid combinations */
    secp256r1_PQ1 (0x????),
    secp256r1_PQ2 (0x????),
    x25519_PQ1 (0x????),
    x25519_PQ2 (0x????),

    /* existing reserved code points */
    ffdhe_private_use (0x01FC..0x01FF),
    ecdhe_private_use (0xFE00..0xFEFF),
    (0xFFFF)
} NamedGroup;
```

4.2.2. ClientHello extension KeyShareClientHello

The client sends a "KeyShareClientHello" struct containing multiple "KeyShareEntry" values, some of which may correspond to some of the hybrid combination code points it listed in the "supported_groups" extension above.

The "KeyShareEntry" for a hybrid combination code point contains an opaque "key_exchange" string which is the octet representation of the following struct:

```
struct {
    KeyShareEntry key_share<2..10>;
} HybridKeyShare;
```

where there is one "key_share" entry for each of the components of this hybrid combination.
Note that this approach may result in duplication of key shares being sent; for example, a client wanting to support either the combination "secp256r1_PQ1" or "x25519_PQ1" would send two "PQ1" key shares.

4.2.3. ServerHello extension KeyShareServerHello

The server responds with a "KeyShareServerHello" struct containing a single "KeyShareEntry", which contains a single "NamedGroup" value and an opaque "key_exchange" string. The "key_exchange" string is the octet representation of the "HybridKeyShare" struct defined above.

4.2.4. Key schedule

The key schedule is computed as in Candidate Instantiation 1 above.

4.3. Comparing Candidate Instantiation 1 and 2

CI2 requires much less change to negotiation routines – each hybrid combination is just a new key exchange method, and the concatenation of key shares and shared secrets can be handled internally to that method. This comes at the cost, however, of combinatorial explosion of code points: one code point needs to be standardized for each desired combination. We have also limited the number of hybrid algorithms to 2 in CI2 to somewhat limit the explosion of code points needing to be defined. Concatenating client key shares also risks sending duplicate key shares, increasing communication sizes.

CI1 requires more change to negotiation routines, since it introduces new data structures and has an indirect mapping between hybrid combinations and key shares. Benefits from this approach include avoiding sending duplicate key shares and not needing to standardize every possible supported combination. Implementers, however, must do the work of deciding which combinations of algorithms are meaningful / tolerable / desirable from a security perspective, potentially complicating interoperability.

5. IANA Considerations

If Candidate Instantiation 1 is selected, the TLS Supported Groups registry will have to be updated to include code points for hybrid markers.

6. Security Considerations

The majority of this document is about security considerations. As noted especially in Section 3.4, 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. While many natural approaches seem to achieve this, there can be subtleties (see for example the introduction of [GIACON]).

The rest of this section highlights a few unresolved questions related to security.

6.1. Active security

One security consideration that is not yet resolved is whether key encapsulation mechanisms used in TLS 1.3 must be secure against active attacks (IND-CCA), or whether security against passive attacks (IND-CPA) suffices. Existing security proofs of TLS 1.3 (such as [DFGS15], [DOWLING]) are formulated specifically around Diffie-Hellman and use an "actively secure" Diffie-Hellman assumption (PRF Oracle Diffie-Hellman (PRF-ODH)) rather than a "passively secure" DH assumption (e.g. decisional Diffie-Hellman (DDH)), but do not claim that the actively secure notion is required. In the context of TLS 1.2, [KPW13] show that, at least in one formalization, a passively secure assumption like DDH is insufficient (even when signatures are used for mutual authentication). Resolving this issue for TLS 1.3 is an open question.

6.2. Resumption

TLS 1.3 allows for session resumption via a pre-shared key. When a pre-shared key 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?

6.3. 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?

7. Acknowledgements

These ideas have grown from discussions with many colleagues, including Christopher Wood, Matt Campagna, and authors of the various hybrid Internet-Drafts and implementations cited in this document. 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) (Section 3.4.2) approach.

8. References

8.1. Normative References


8.2. Informative References


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
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.
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 clients 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


6.2. Informative References

Author’s Address

Martin Thomson
Mozilla

Email: mt@lowentropy.net
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 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.
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 content NAT rebinding is a common reason 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.

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.
An on-path adversary could intercept and modify the source IP address (and the source port). Even if receiver checks the authenticity and freshness of the packet, the recipient is fooled into changing the CID-to-IP/port association. This attack is possible because the network and transport layer identifiers, such as source IP address and source port numbers, are not integrity protected and authenticated by the DTLS record layer.

This attack makes strong assumptions on the attacker’s abilities, and moreover it only misleads the peer until the next message gets through un-intercepted.

A return routability check (RRC) 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 a certain degree of confidence to the receiving peer that the sending peer is reachable at the indicated address and port.

Without such a return routability check, an adversary can redirect traffic towards a third party or a black hole.

While an equivalent check can be performed at the application layer (modulo the DTLS API exposing the address update event to the calling application), it is advantageous to offer this functionality at the DTLS layer. Section 3 describes the application layer procedure and Section 4 specifies a new message to perform this return routability check.

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 solutions defined for DTLS 1.2 [I-D.ietf-tls-dtls-connection-id] and for DTLS 1.3 [I-D.ietf-tls-dtls13].

3. Application Layer Return Routability Check

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). This is to ensure that a man-on-the-middle attacker that sends a datagram
with a different source address/port on an existing CID session does not successfully manage to re-route any return traffic.

Furthermore, when using CID, anti-replay protection MUST be enabled. This is to ensure that a man-on-the-middle attacker sending a previously captured record with a modified source IP address and port will not be able to successfully pass the above check (since the datagram is very likely discarded on receipt - if it falls outside the replay window).

The two countermeasures cannot complete stop a man-in-the-middle attacker who performs a DoS on the sender or uses the receiver as as backscatter source for a DDoS attack. For a more generic protection, a return routability check is needed.

It is RECOMMENDED that implementations of the CID functionality described in [I-D.ietf-tls-dtls-connection-id] and in [I-D.ietf-tls-dtls13] added peer address update events to their APIs. Applications can then use these events as triggers to perform an application layer return routability check, for example one that is based on successful exchange of minimal amount of ping-pong traffic with the peer.

4. The Return Routability Check Message

```c
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;
```

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 endpoint that observes the peer’s address 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.

```
struct {
    opaque cookie<1..2^16-1>;
} Cookie;

struct {
    Cookie cookie;
} return_routability_check;
```

5. 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.
Figure 1: Return Routability Example
6. Security and Privacy Considerations

As all the datagrams in DTLS are authenticated, integrity and confidentiality protected there is no risk that an attacker undetectably modifies the contents of those packets. The IP addresses in the IP header and the port numbers of the transport layer are, however, not authenticated. With the introduction of the CID, care must be taken to test reachability of a peer at a given IP address and port.

Note that the return routability checks do not protect against third-party flooding if the attacker is along the path, as the attacker can forward the return routability checks to the real peer (even if those datagrams are cryptographically authenticated).

7. 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.

8. 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?

9. References

9.1. Normative References

[I-D.ietf-tls-dtls-connection-id]

[I-D.ietf-tls-dtls13]

9.2. URIs

[1] mailto:tls@ietf.org


Appendix A. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

- Initial version

Appendix B. Working Group Information

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

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

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

Appendix C. 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